Ensta Bretagne
2, rue François Verny
29806 Brest cedex
France
Tel +33 (0)2 98 34 88 00
www.ensta-bretagne.fr

Support de cours

FISE 1A

24 août 2018

Langage et algorithmique

Rodéric Moitié, Christophe Osswald
prénom.nom@ensta-bretagne.fr

Table des matières

Préambule

CERTAINES parties de ce cours sont tirées de cours et documents disponibles sur internet, Wikipedia [1] et commentcamarche.net [2].

Merci à Amandine, pour ses commentaires et son excellent travail de relecture [3]. Merci à Arnaud et Malek pour leurs participations à l'écriture de ce document. Merci à Irvin pour ses conseils avisés ainsi que son regard acéré sur les exemples présentés. Et surtout, merci à Douglas Adams de nous avoir donné la réponse [Ada82].

1. http://fr.wikipedia.org
2. http://www.commentcamarche.net/poo/
3. L'un des auteurs tient à s'excuser par avance au cas où certains lecteurs trouveraient excessif l'usage des notes de bas de page.

En ces jours, les esprits étaient braves, les enjeux étaient élevés, les hommes de vrais hommes, les femmes de vraies femmes, et les petites bestioles fourrées d'Alpha du Centaure de vraies petites bestioles fourrées d'Alpha du Centaure.

Douglas Adams

1 Un peu d'histoire

Sommaire

1.1 Algorithmique

L'ALGORITHMIQUE, ou étude des algorithmes, est une partie essentielle de l'informatique. Elle a été systématisée par le mathématicien arabe *Abû 'Abd Allah Muhammad ben Mūsā*

al-Khawārizmī — ابُو أَبد الَّه مُهَمَّد بِن مُوسَا الكهَوَارِزمِي — (780-850), auteur d'un ouvrage décrivant des méthodes de calculs algébriques (ainsi que d'un autre introduisant le zéro des Indiens). Son nom donna au Moyen Âge le mot « *algorisme* » et par la suite algorithme.

Voici une définition informelle de ce terme :

Définition 1.1 (Algorithme)**.** *Un algorithme*[1] *est une suite finie d'étapes, réalisées dans un ordre déterminé, appliquées à un nombre fini de données afin d'arriver avec certitude à un certain résultat, et ce, indépendamment des données de départ.*

Un algorithme peut être vu comme un outil permettant de résoudre un problème donné, ce problème étant défini en tant que relation désirée entre des données d'entrée et des données de sortie. Par exemple, un problème de tri peut être défini de la manière suivante : on se donne en entrée une suite de n nombres $a_1, a_2, ..., a_n$, et on souhaite obtenir en sortie une permutation de l'entrée $a'_1, ..., a'_n$ telle que $a'_1 \leqslant a'_2 \leqslant ... \leqslant a'_n$. Un algorithme de tri sera alors une suite d'opérations permettant de passer de l'entrée à la sortie.

1.1.1 Quelques dates importantes

Une étape importante dans l'histoire de l'algorithmique a été la conception — vers 1820 — par Charles Babbage (1792-1871) d'une machine appelée « difference engine », une machine à calculer selon les différences finies (qu'il n'achèvera jamais[2]). Alors qu'il travaillait sur ce projet, il rencontra Lady Ada Lovelace (1815-1852), fille de Lord Byron qui devint son assistante. De leur collaboration naquit l'idée d'une machine universelle, capable d'effectuer toutes sortes de calculs par simple changement de programme : il s'agit du précurseur de l'ordinateur. La description complète de cette machine fut publiée par Ada Lovelace. Hélas, les techniques de l'époque n'étaient pas suffisamment avancées pour leur permettre de la terminer...

Les notes d'Ada Lovelace sur la machine de Babbage contiennent une description très détaillée d'une méthode de calcul des nombres de Bernoulli à l'aide de cette machine. Elles sont considérées comme le premier programme informatique[3]. On peut également noter qu'Ada Lovelace fut à l'origine de l'utilisation du terme *algorithme*, et qu'un langage à haut niveau de fiabilité et de sécurité porte aujourd'hui son prénom.

Par la suite, il faudra attendre l'arrivée des tubes à vide (inventés en 1904 par John Flemming) pour voir les premiers ordinateurs. À partir de 1930, quelques modèles particulièrement spectaculaires ont été conçus :

- l'IBM 601 en 1935, un calculateur à relais permettant d'effectuer une multiplication par seconde ;
- les calculateurs *Robinson* et *Colossus* en 1940, qui ont servi à décoder les messages chiffrés de l'armée allemande[4] ;
- l'ENIAC[5] en 1946, qui se composait de 1900 tubes, pesait 30 tonnes, occupait une surface de $70m^2$ et consommait $140kW$. Il était cadencé à $100kHz$ et pouvait effectuer 330 multiplications par seconde.

En 1944, John Von Neumann rencontre le responsable du projet ENIAC. À la suite de cette rencontre, il propose un modèle de machine algorithmique universelle appelée *machine de Von Neumann*. Dans ce modèle, la machine est composée de quatre parties :

- l'UAL[6] qui permet d'effectuer les opérations de base ;
- l'unité de contrôle qui s'occupe de l'enchaînement des instructions ;

1. Définition extraite de wikipedia.
2. Mais le principe de fonctionnement était totalement correct, comme l'ont prouvé deux chercheurs Britanniques en réalisant une telle machine en 1991.
3. Bien que les biographes considèrent plutôt que ces programmes aient été écrits par *Babbage* lui-même.
4. Ces ordinateurs sont restés inconnus car classés secret défense jusqu'en 1975.
5. Electronic Numerical Integrator and Computer.
6. Unité Arithmétique et Logique

— la mémoire qui contient à la fois le programme et les données ; elle se divise en mémoire volatile et mémoire permanente ;
— les dispositifs d'entrée-sortie.

Pendant ce temps, d'énormes progrès théoriques ont été réalisés par Alan Turing (1912-1954). Turing est considéré comme le père de l'informatique moderne. Il est à l'origine du concept d'un modèle abstrait d'ordinateur : la *machine de Turing* (voir la section D.5). Il a également démontré qu'il était possible de créer une machine de Turing, appelée *machine de Turing universelle*, permettant de simuler le comportement de n'importe quelle autre *machine de Turing*.

Un résultat particulièrement intéressant de Turing est l'indécidabilité du problème d'arrêt : il est impossible de définir une machine de Turing (ou un programme) permettant de décider si une machine de Turing (ou un programme) quelconque se terminera en un temps fini. Ce résultat a des répercussions très importantes : il prouve qu'il existe une classe de problèmes qu'il est impossible de résoudre par un algorithme.

1.1.2 Importance de l'algorithmique

L'algorithmique est une étape fondamentale lors de la réalisation d'un programme. Face à un problème donné, la démarche permettant d'arriver à un programme est la suivante : il faut tout d'abord représenter correctement le problème, de manière non ambigüe. Puis, il faut trouver un algorithme permettant de résoudre ce problème [7]. Notons que la connaissance de certains algorithmes classiques est ici d'un grand secours. Il faut ensuite étudier l'algorithme : quelles devraient être ses performances *a priori*, répond-il bien à mon problème, suis-je sûr de sa terminaison ? Finalement, il faut traduire cet algorithme dans un langage de programmation.

L'objectif de ce cours est double : il s'agit d'une part d'apprendre à reconnaître certains problèmes classiques ainsi que les algorithmes permettant de les résoudre, et il s'agit d'autre part d'apprendre le langage Python, les spécificités de la programmation orientée objet, et à traduire des algorithmes en Python.

Ce document est découpé de la manière suivante : le chapitre 2 décrit la syntaxe générale et les constructions élémentaires du langage Python. Les chapitres 3 et 6 présentent quelques algorithmes et structures de données fondamentaux. Le chapitre 5 décrit les principes de la programmation orientée objet, dans le cas général, puis dans le cas spécifique de Python. Enfin, les chapitres 7 et 8 décrivent quelques bibliothèques et mécanismes spécifiques de Python.

1.2 Description d'un ordinateur

Texte tiré de Wikipedia, l'encyclopédie libre.

Le processeur, (ou en anglais, CPU, sigle de Central Processing Unit pour « unité centrale [de traitement] ») est le composant essentiel d'un ordinateur, où sont effectués les principaux calculs. Il ne s'agit pas nécessairement d'un circuit isolé, même si les progrès techniques depuis les premiers emplois du terme le permettent aujourd'hui, voire même de placer plusieurs CPU dans un même processeur.

Néanmoins, la distinction entre Central Processing Unit, CPU, processeur et microprocesseur est souvent abandonnée au profit d'une banalisation de ces termes.

1.2.1 Principe de fonctionnement

Le CPU est l'unité de traitement de données principale d'un ordinateur, ce qui veut dire qu'il va exécuter les programmes, ce qui peut inclure de déléguer une partie du traitement à d'autres

7. Ou démontrer que le problème posé n'est pas décidable...

processeurs périphériques. En plus de sa capacité de traitement, il a donc également une fonction de contrôle et de coordination de l'action de l'ensemble des composants d'un ordinateur. Un programme est un ensemble d'instructions situé dans la mémoire centrale de l'ordinateur, que le processeur va lire puis exécuter séquentiellement, à moins d'un saut dans le programme. Le temps d'exécution propre à chaque instruction est exprimé en cycles de l'horloge interne qui cadence l'activité du processeur.

1.2.2 Structure

Les parties essentielles d'un processeur sont :

— l'Unité Arithmétique et Logique (UAL, en anglais Arithmetic and Logical Unit - ALU), qui prend en charge les calculs arithmétiques élémentaires et les tests ;

— l'Unité de Contrôle ;

— les registres, qui sont des mémoires de petite taille (quelques octets), suffisamment rapides pour que l'UAL puisse manipuler leur contenu à chaque cycle de l'horloge. Un certain nombre de registres sont communs à la plupart des processeurs :

  compteur d'instructions : ce registre contient l'adresse mémoire de l'instruction en cours d'exécution ;

  accumulateur : ce registre est utilisé pour stocker les données en cours de traitement par l'UAL ;

  registre d'adresses : il contient toujours l'adresse de la prochaine information à lire par l'UAL, soit la suite de l'instruction en cours, soit la prochaine instruction ;

  registre d'instructions : il contient l'instruction en cours de traitement ;

  registre d'état : il sert à stocker le contexte du processeur, ce qui veut dire que les différents bits de ce registre sont des drapeaux (flags) servant à stocker des informations concernant le résultat de la dernière instruction exécutée ;

  pointeurs de pile : ce type de registre, dont le nombre varie en fonction du type de processeur, contient l'adresse du sommet de la pile (ou des piles) ;

  registres généraux : ces registres sont disponibles pour les calculs.

— le séquenceur, qui permet de synchroniser les différents éléments du processeur. En particulier, il initialise les registres lors du démarrage de la machine et il gère les interruptions ;

— l'unité d'entrée-sortie, qui prend en charge la communication avec la mémoire de l'ordinateur ou la transmission des ordres destinés à piloter ses processeurs spécialisés, permettant au processeur d'accéder aux périphériques de l'ordinateur.

Les processeurs actuels intègrent également des éléments plus complexes :

— plusieurs UAL, ce qui permet de traiter plusieurs instructions en même temps. L'architecture superscalaire, en particulier, permet de disposer des UAL en parallèle, chaque UAL pouvant exécuter une instruction indépendamment de l'autre ;

— l'architecture superpipeline permet de découper temporellement les traitements à effectuer. C'est une technique qui vient du monde des supercalculateurs ;

— une unité de prédiction de saut, qui permet au processeur d'anticiper un saut dans le déroulement d'un programme, permettant d'éviter d'attendre la valeur définitive d'adresse du saut. Cela permet de mieux remplir le pipeline ;

— une unité de calcul en virgule flottante (en anglais Floating Point Unit - FPU), qui permet d'accélérer les calculs sur des nombres réels codés en virgule flottante ;

— la mémoire cache, qui permet d'accélérer les traitements, en diminuant les accès à la RAM. Ces mémoires tampons sont en effet beaucoup plus rapides que la RAM et ralentissent moins le CPU. Le cache instructions reçoit les prochaines instructions à exécuter, le cache données manipule les données. Parfois, un autre cache unifié est utilisé. Dans les microprocesseurs évolués, des unités spéciales du processeur sont dévolues à la recherche, par des moyens statistiques et/ou prédictifs, des prochains accès en mémoire centrale.

1.2.3 Langage

Les instructions données au processeur sont exprimées en binaire (code machine). Elles sont généralement stockées dans la mémoire. Elles sont lues et l'UAL les interprète. L'ensemble de ces instructions constitue un programme.

Le langage le plus proche du code machine tout en restant lisible par des humains est le langage d'assemblage, aussi appelé langage assembleur (forme francisée du mot anglais « assembler »). Toutefois, l'informatique a développé toute une série de langages, dits de haut niveau, destinés à simplifier l'écriture des programmes.

Ainsi, alors que l'ordinateur reconnaîtra ce que l'instruction machine `10110000 01100001` signifie, pour le programmeur il est plus simple de se souvenir de son équivalent en langage assembleur :

```
mov $0x61, %al
```

(cela signifie de mettre la valeur hexadécimale 61 (97 en décimal) dans le registre 'AL'.)

Contrairement à un langage de haut niveau, il y a une correspondance 1-1 entre le code assembleur et le langage machine, ainsi il est possible de traduire le code dans les deux sens sans perdre d'information. La transformation du code assembleur en langage machine est accomplie par un programme nommé assembleur, dans l'autre sens par un programme désassembleur. Les opérations s'appellent respectivement assemblage et désassemblage. Dans un programme réel en assembleur, c'est un peu plus complexe que cela (on peut donner des noms aux routines, aux variables), et on n'a plus cette correspondance. Sur les premiers ordinateurs, la tâche d'assemblage était accomplie manuellement par le programmeur.

Chaque architecture d'ordinateurs a son propre langage machine, et donc son propre langage d'assemblage (l'exemple ci-dessus est pour le x86). Ces différents langages diffèrent par le nombre et le type d'opérations qu'ils ont à supporter. Ils peuvent avoir des tailles et des nombres de registres différents, et différentes représentations de type de données en mémoire. Tous les ordinateurs sont capables de faire les mêmes choses, ils peuvent les faire de manière différente.

De plus, plusieurs groupes de mnémoniques ou de syntaxe de langage assembleur peuvent exister pour un seul ensemble d'instructions. Dans ce cas, le plus populaire est habituellement celui de la documentation du fabricant. Notre exemple ci-dessus est donné en syntaxe AT&T. En syntaxe Intel, cela donnerait :

```
MOV AL,61h
```

C'est vraiment histoire de goût - les opérandes sont inversés - et des possibilités du programme d'assemblage (certains gèrent les deux syntaxes, d'autres une seule). Néanmoins cela ne facilite pas la maintenance des programmes !

Remarquons quand même que cette instruction se traduit, dans la plupart des langages évolués, par (ici en pseudo-C) :

```
AL = 0x61 ;
```

La syntaxe « Intel » semble la plus logique.

Remarquons aussi qu'en français[8] on dirait : « mettre 0x61 dans AL ». La syntaxe « AT&T » semble de ce point de vue être la plus logique.

1.2.4 Instructions machine

Des opérations de base sont disponibles dans la plupart des jeux d'instructions :

déplacement :

8. et aussi dans la plupart des autres langues pratiquées par les humains

— chargement d'une valeur dans un registre ;
— déplacement d'une valeur depuis un emplacement mémoire dans un registre, et inversement ;

calcul :
— addition, soustraction, multiplication ou division des valeurs de deux registres et chargement du résultat dans un registre ;
— combinaison de valeurs de deux registres avec un et/ou logique ;
— rendre négative la valeur d'un registre arithmétiquement (complément à 2) ou par un non logique (complément à 1) ;

modification du déroulement du programme :
— saut à un autre emplacement dans le programme (normalement, les instructions sont exécutées séquentiellement, les unes après les autres) ;
— saut à un autre emplacement, mais après avoir sauvegardé l'emplacement de l'instruction suivante afin de pouvoir y revenir (point de retour) ;
— retour au dernier point de retour ;

comparaison :
— comparer les valeurs de deux registres.

Et on trouve des instructions spécifiques avec une ou quelques instructions pour des opérations qui auraient dû en prendre beaucoup.

Exemples :
— déplacement de grands blocs de mémoire ;
— arithmétique lourde (sinus, cosinus, racine carrée, etc.) ;
— application d'une opération simple (par exemple, une addition) à un ensemble de données.

1.2.5 Directives du langage assembleur

En plus de coder les instructions machine, les langages assembleur ont des directives supplémentaires pour assembler des blocs de données et assigner des adresses aux instructions en définissant des étiquettes ou labels.

Ils sont capables de définir des expressions symboliques qui sont évaluées à chaque assemblage, rendant le code encore plus facile à lire et à comprendre.

Ils ont habituellement un langage macro intégré pour faciliter la génération de codes ou de blocs de données complexes.

1.2.6 Usage du langage assembleur

Il y a des débats sur l'utilité du langage assembleur. Dans beaucoup de cas, des compilateurs-optimiseurs peuvent transformer du langage de haut niveau dans un code qui tourne de façon plus efficace qu'un code assembleur écrit à la main, tout en restant beaucoup plus facile à lire et à « maintenir ».

Cependant,

1. quelques calculs complexes écrits directement en assembleur, en particulier sur des machines massivement parallèles, seront plus rapides;
2. certaines routines (drivers) sont parfois plus simples à écrire en langage de bas niveau ;
3. des tâches très dépendantes du système, exécutées dans l'espace mémoire du système d'exploitation sont parfois difficiles à écrire dans un langage de haut niveau.

Certains compilateurs transforment, lorsque leur option d'optimisation la plus haute n'est pas activée, des programmes écrits en langage de haut niveau en code assembleur, chaque instruction de haut niveau se traduisant en une série d'instructions assembleur rigoureusement équivalentes

et utilisant les mêmes symboles ; cela permet de voir le code dans une optique de débogage et de profilage [9]. En aucun cas ces techniques ne peuvent être conservées pour l'optimisation finale.

Certains systèmes embarqués sont aussi programmés en assembleur pour bénéficier du maximum des possibilités de ces systèmes, souvent limités en ressources. Comme les composants de ces systèmes sont de plus en plus puissants pour un coût identique, il devient plus courant d'y utiliser des langages de haut niveau.

Fin de l'extrait de wikipedia, l'encyclopédie libre.

1.2.7 Exemple de code en assembleur

Pour faire une addition en assembleur ARM et afficher le résultat, le code à taper sera :

Listing 1.1 – Programme assembleur

```
1  R0       RN 0                    ;Definition reg.0
2  R1       RN 1                    ;Definition reg.1
3  R2       RN 2                    ;Definition reg.2
4
5  LMAX     EQU 1                   ;1 chiffre max.
6
7           AREA PREMADD,CODE       ;Zone principale
8
9           ENTRY                   ;Point d'entree
10          MOV R0,#1               ;R0=1
11          MOV R1,#1               ;R1=1
12
13          ADD R0,R0,R1            ;R0=R0+R1
14          ADR R1,RESULT           ;Adresse de Result
15          MOV R2,#LMAX            ;nb max. chiffres
16          SWI &28                 ;OS_BinaryToDecimal
17
18          ADR R0,MOUT             ;Adresse message
19          SWI 2                   ;OS_Write0
20
21          MOV R0,R1               ;Adresse de Result
22          MOV R1,R2               ;R1=R2 (nb car.)
23          SWI &46                 ;OS_WriteN
24
25          SWI &11                 ;OS_Exit
26
27 MOUT     DCB "Resultat: ",0      ;A afficher
28 RESULT   % LMAX                  ;Res. Lmax octets
29
30          END                     ;Fin d'assemblage
```

Une fois ce programme traduit en langage machine, son exécution provoquera l'affichage suivant :

```
Resultat: 2
```

Le même exemple dans dans un langage de plus haut niveau, comme le C, donnerait :

```
1  #include <stdio.h> //chargement des définitions de fonctions usuelles
2  #include <stdlib.h>
3  int main(int argc, char* argv[]){  // Point d'entrée
```

9. Le profilage permet de gagner parfois beaucoup plus de temps en remaniant un algorithme

```
4           int R0, R1;                    // Définition des variables R0 et R1
5           R0 = 1;
6           R1 = 1;
7           R0 = R1 + R0;
8           printf("Resultat %d", R0); // Affichage
9           return EXIT_SUCCESS;       // Fin du programme
10  }
```

Le premier avantage évident de s'abstraire du code assembleur est la lisibilité et la concision du code à écrire. Le second est que même dans un langage relativement bas niveau comme le C ce code pourra être réutilisé partout sans avoir à l'écrire à nouveau, alors que l'exemple en assembleur est lié à un processeur particulier. Cependant cet exemple en C contient encore des lignes qui le lient fortement à la machine, par exemple l'instruction « printf » qui permet d'afficher le résultat à l'utilisateur pourra proposer plus ou moins d'options suivant le type d'ordinateur que l'on utilisera.

Python monte encore d'un cran dans l'abstraction en permettant d'écrire du code censé fonctionner partout sans avoir à se soucier de contraintes bas niveau. Il est important de comprendre que ces deux exemples font la même chose, quel que soit le langage utilisé. Seul le niveau d'abstraction change ; tôt ou tard il faut en revenir au code assembleur/machine par des outils automatiques appropriés.

1.3 Représentation des variables [10]

1.3.1 Rappel sur les bases

Nous recensons ici quelques bases qu'il est possible de rencontrer.

Base 10 est la base la plus couramment utilisée par les êtres humains (c'est la base naturelle lorsque l'on compte sur ses doigts). Elle est relativement inadaptée à l'informatique...

Base 2 ou *binaire* utilise uniquement les chiffres 0 et 1. C'est la base principale en informatique car les composants élémentaires des mémoires possèdent uniquement deux états. L'inconvénient principal de cette base est le nombre de chiffres nécessaires pour représenter une valeur. On appelle un *octet* un nombre binaire de 8 chiffres.

Base 16 ou *hexadécimal* utilise 16 chiffres : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E et F. Cette base est beaucoup plus compacte que le binaire, et il est très simple de passer du binaire à l'hexadécimal et réciproquement. En hexadécimal, un octet s'écrit avec 2 chiffres.

Base 8 ou *octal* utilise les huit chiffres de 0 à 7. En octal, un octet s'écrit avec 3 chiffres, de 000 à 377.

Base 1 est une base uniquement utile d'un point de vue théorique. Cette base contient un seul chiffre que l'on note 1. Un entier s'écrit par juxtaposition de 1. Par exemple, le nombre 5 est représenté par cinq fois le chiffre 1, c'est-à-dire 11111. Cette base peut être utilisée par des machines de Turing (voir par exemple chapitre D.5).

1.3.2 Représentations des entiers : complément à 2

Un entier relatif doit être codé de telle façon que l'on puisse savoir s'il s'agit d'un nombre positif ou d'un nombre négatif, et il faut de plus que les règles d'addition soient conservées. L'astuce consiste à utiliser un codage que l'on appelle complément à deux :

10. Ce paragraphe est écrit par Jean-François Pillou, extrait de « Comment ça marche » http://www.commentcamarche.net et disponible sous les termes de la licence « Creative Commons » http://www.commentcamarche.net/ccmguide/ccmlicence.php3.

Un entier relatif positif ou nul sera représenté en binaire (base 2) comme un entier naturel, à la seule différence que le bit de poids fort (le bit situé à l'extrême gauche) représente le signe. Il faut donc s'assurer pour un entier positif ou nul qu'il est à zéro (0 correspond à un signe positif, 1 à un signe négatif). Ainsi si on code un entier naturel sur 4 bits, le nombre le plus grand sera 0111 (c'est-à-dire 7 en base décimale).

D'une manière générale le plus grand entier relatif positif codé sur n bits sera $2^{n-1} - 1$.

Un entier relatif négatif sera représenté grâce au codage en complément à deux :
- prendre son opposé (son équivalent en positif) ;
- le représenter en base 2 sur $n - 1$ bits ;
- complémenter chaque bit (on inverse, c'est-à-dire que l'on remplace les zéros par des 1 et vice-versa), y compris le n^{e} chiffre, qui est le 0 le plus à gauche ;
- ajouter 1.

On remarquera qu'en ajoutant le nombre et son complément à deux on obtient 0...
Le plus petit entier codé sur 4 bits est -8, représenté par 1000 ; l'entier -1 est représenté par 1111.

Voyons maintenant cela sur un exemple. On désire coder la valeur -5 sur 8 bits. Il suffit :
- d'écrire 5 en binaire : 00000101 ;
- de complémenter à 1 : 11111010 ;
- d'ajouter 1 : 11111011 ;
- la représentation binaire de -5 sur 8 bits est 11111011.

Remarques:
- le bit de poids fort est 1, on a donc bien un nombre négatif ;
- si on ajoute 5 et -5 (00000101 et 11111011) on obtient 0 (avec une retenue de 1).

1.3.3 Représentation des réels : norme IEEE754

Il s'agit d'aller représenter un nombre binaire à virgule (par exemple 101.01 qui ne se lit pas cent un virgule zéro un puisque c'est un nombre binaire mais 5.25 en décimal) sous la forme $1.XXXXX... * 2^n$ (c'est-à-dire dans notre exemple $1.0101 * 2^2$). La norme IEEE définit la façon de coder un nombre réel.

Cette norme se propose de coder le nombre sur 32 bits (simple précision) ou 64 bits (double précision) et définit trois composantes :
- le signe est représenté par un seul bit, le bit de poids fort (celui le plus à gauche) ;
- l'exposant est codé sur les 8 bits consécutifs au signe, 11 en double précision ;
- la mantisse (les bits situés après la virgule) sur les 23 bits restants, 52 en double précision.

Ainsi le codage en simple précision se fait sous la forme suivante :
`seeeeeeeemmmmmmmmmmmmmmmmmmmmmmm`
- le s représente le bit relatif au signe ;
- les e représentent les bits relatifs à l'exposant ;
- les m représentent les bits relatifs à la mantisse.

Certaines conditions sont toutefois à respecter pour les exposants :
- l'exposant 00000000 est interdit ;
- l'exposant 11111111 est interdit. On s'en sert toutefois pour signaler des erreurs, on appelle alors cette configuration du nombre *NaN*, ce qui signifie *Not A Number* ;
- les exposants peuvent ainsi aller de -126 à 127.

Voyons ce codage sur un exemple : Soit à coder la valeur 525.5.
- 525.5 s'écrit en base 2 de la façon suivante : 1000001101.1 ;
- on veut l'écrire sous la forme $1.0000011011 * 2^9$;
- par conséquent :
  - le bit s vaut 0 ;
  - l'exposant vaut 9, et 127+9=136 soit 10001000 en binaire ;

— la mantisse est $00000110110\cdots0$.
— la représentation du nombre 525.5 en binaire avec la norme IEEE est : 01000100000000110110000000000000.

Le type de base de Python, *float* utilise une double précision, mais la bibliothèque *numpy* (section 4.3) propose d'autres formats.

1.3.4 Représentation des caractères : norme unicode

Les caractères utilisés par Python sont les caractères unicode 16 bits version 4.0. Pour plus d'informations sur la norme unicode, consulter http://www.unicode.org.

Les caractères sont codés sur deux octets et sont donc représentés par des entiers compris entre 0 et 65535 (0 et FFFF en hexadécimal).

1.4 Présentation du langage Python

Il existe de nombreux ouvrages présentant le langage Python. Les débutants pourront par exemple consulter [Swi12] ou [Le 12]. Ces deux ouvrages sont très didactiques et couvrent très bien les bases du langage.

Ceux qui utilisent Python pour résoudre des problèmes mathématiques pourront compléter leurs connaissances avec [CCC12] ; cet ouvrage présente des algorithmes classiques et des méthodes d'analyse numérique.

La première version de Python date de 1991, développée par Guido Van Rossum. Le nom du langage est un hommage au *Monty Python's Flying Circus*. Il utilise une syntaxe très légère, inspirée d'ABC. Le langage dispose de très peu de types de données, mais est riche en structures de données dynamiques, bien intégrées à la syntaxe.

Python est exécuté par une machine virtuelle, qui est le programme `/usr/bin/python` sur les installations linux usuelles, et le programme `python.exe` sur les installations Windows usuelles. Le nom de l'exécutable peut contenir un numéro de version, ce qui permet de faire coexister différentes versions sur un même système. Cette possibilité est particulièrement utile aux périodes de transition entre deux versions majeures : `/usr/bin/python2`, `/usr/bin/python3.4`.

La machine virtuelle permet de détacher le code exécutable des contraintes matérielles liées à la plate-forme. Un même programme python — le fichier texte en `.py` — pourra ainsi être exécuté aussi bien sur une machine Windows, Linux, MacOS ou Android. Par contre, un fichier Python 3 ne pourra pas[11] être exécuté par une machine virtuelle Python 2.

Dans d'autres langages, comme C, C++, Pascal, Delphi, Fortran, ..., le code source doit être recompilé sur chacune des architectures cible. Ceci est encore différent d'un programme assembleur qui doit être réécrit en cas de changement d'architecture matérielle. Tout ceci est représenté sur la figure 1.1, sur laquelle il convient de considérer que CPU1 et CPU2 sont deux ordinateurs distincts, le type de microprocesseur et/ou de système d'exploitation étant différents.

La machine virtuelle python procède au traitement du fichier source en deux temps. Elle commence par analyser syntaxiquement le fichier (elle vérifie donc que le code source est bien écrit en Python, c'est-à-dire qu'il en respecte la grammaire). Elle profite de cette vérification pour créer un *bytecode* équivalent au code d'origine, mais plus simple (comprendre : rapide) à interpréter. Ce bytecode peut être enregistré dans un fichier *.pyc* ou *.pyo*, ce qui accélérera une exécution ultérieure. Ce bytecode est lui-même indépendant de la machine physique. Dans un deuxième temps, la machine virtuelle interprète le *bytecode*, ce qui exécute le programme.

Le langage java procède de la même manière que Python, mais répartit ces fonctions sur deux programmes distincts : l'un génère le bytecode et l'autre l'interprète.

11. Sauf si on se force à n'utiliser que la syntaxe commune aux deux versions, ce qui empêche notamment d'utiliser `print`, et à se priver de plusieurs opérateurs de base... Cette option est à considérer comme un thème d'Oulipo informatique plutôt que comme un choix d'ingénierie.

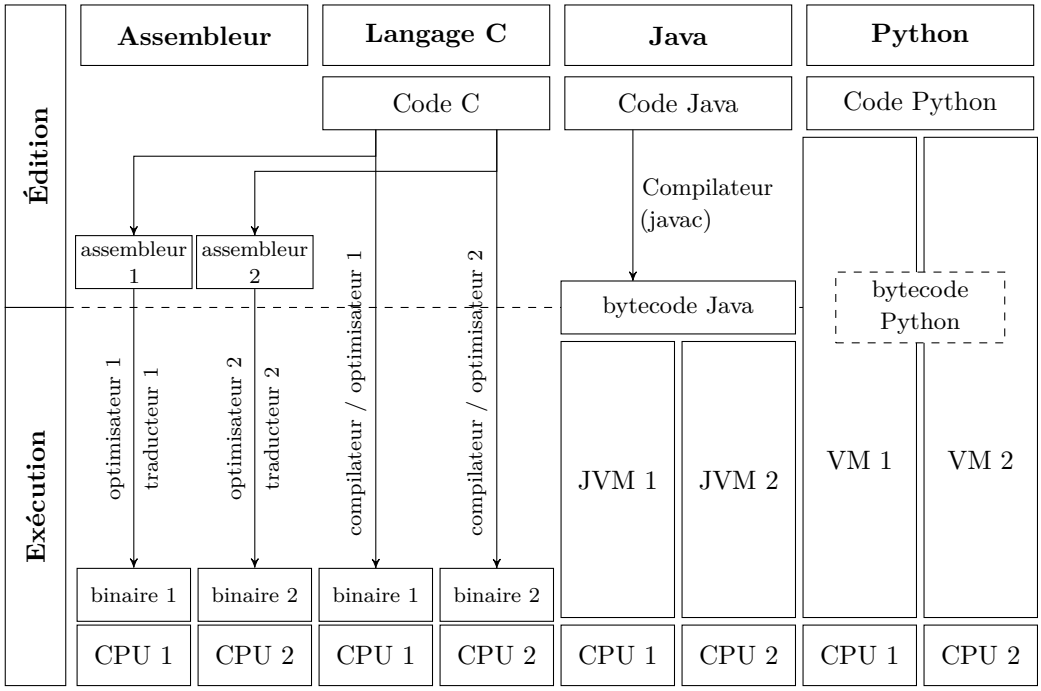


FIGURE 1.1 – Comparaison entre différents langages

1.4.1 Exécution d'un programme python

Il y a deux façons d'exécuter du code Python :

mode interactif : l'interpréteur `python3` est lancé dans un terminal, et exécute les instructions au fur et à mesure qu'elles sont saisies par l'utilisateur. Ce mode permet d'utiliser Python comme une calculatrice, ou de tester de petits morceaux de programme. L'invite de commande se fait par `>>>`. Si le résultat n'est pas stocké dans une variable, il est affiché dans la console.

```
>>> 6*7
42
>>> a,b = 6, 7
>>> a*b
42
```

L'interpréteur `ipython3` peut remplacer `python3` et offre un comportement similaire[12]. L'invite de commande se fait par `In[i]`. Si le résultat n'est pas stocké dans une variable, il est affiché dans la console après le `Out[i]`.

```
In [1]: 6*7
Out[1]: 42
In [2]: a, b = 6, 7
In [3]: a*b
Out[3]: 42
```

12. Il offre les mêmes possibilités que `python3`, mais il dispose d'une meilleure complétion, affiche des messages d'erreur plus lisibles, et est mieux intégré au shell linux.

exécution de script : un fichier Python est fourni à l'interpréteur. L'interpréteur lit l'intégralité du fichier, le vérifie syntaxiquement, génère puis exécute le *bytecode* associé. Si le programme à exécuter est défini dans le fichier *factorielle.py* et prend en compte un paramètre en ligne de commande, on pourra l'appeler par :

```
$ python3 factorielle.py 7
Résultat : 7! = 5040
```

Listing 1.2 – Calcul de factorielle. Fichier *factorielle.py*

```
1 import sys
2
3 f = 1
4 for n in range(1, int(sys.argv[1])+1):
5     f *= n
6 print("Résultat : {}! = {}".format(sys.argv[1], f))
```

Sur les systèmes Unix/Linux, il est possible de rendre le fichier exécutable [13] et de spécifier l'interpréteur associé en première ligne du script, par un *shebang* :

Listing 1.3 – Calcul de factorielle, avec shebang. Fichier *factorielle*

```
1 #!/usr/bin/python3
2 # -*- coding: utf-8 -*-
3 import sys
4
5 f = 1
6 for n in range(1, int(sys.argv[1])+1):
7     f *= n
8 print("Résultat : {}! = {}".format(sys.argv[1], f))
```

L'usage est de ne pas mettre d'extension `.py` au fichier dans ce cas. Le programme s'exécute en lançant simplement :

```
$ factorielle 7
Résultat : 7! = 5040
```

Remarque : La ligne 2 du programme 1.3 sert à indiquer à l'interpréteur Python que le code source est écrit avec l'encodage de caractères *latin-1*, alors qu'il est conseillé [14] par défaut d'utiliser *UTF-8*. Ici, l'encodage permet une intégration plus simple dans le polycopié écrit avec LaTeX. Les autres programmes-exemples du polycopié dissimulent cette astuce en n'affichant pas leur première ligne. Dans cette situation, la première ligne doit être le *shebang*, et la seconde doit renseigner l'encodage.

Il est fortement recommandé d'exécuter un script écrit dans un éditeur de texte approprié plutôt que d'utiliser le mode interactif dès que le programme à exécuter dépasse trois lignes...

1.4.2 Caractéristiques du langage

Python possède un certain nombre de caractéristiques qui ont largement contribué à son succès :

Le Python, c'est bon, et il est agréable à lire. Les blocs sont déterminés par l'indentation des lignes, et non par des signes de ponctuation spécifiques. Ceci garantit que les blocs apparaissent de façon visuelle au lecteur humain, et évite de surcharger le code par des

13. Commande `chmod`, à (re)consulter dans les débuts de l'UV 1.1.
14. Réellement, suivez ce conseil !

informations destinées uniquement au compilateur. La grammaire et les mots-clefs sont proches de l'anglais [15].

Python est interprété : Le code source est interprété par une machine virtuelle. Un même programme peut être exécuté sur des systèmes très différents, et produire les mêmes résultats.

Python est indépendant de la plate-forme : il est possible d'exécuter des programmes Python sur tous les environnements qui possèdent une machine virtuelle correspondant à cette version de Python. La compatibilité entre versions, même ascendante, n'est pas garantie. L'indépendance est assurée au niveau du code source grâce à Unicode, mais les questions d'encodage des caractères restent prégnantes au sein du langage.

Python est orienté objet : comme la plupart des langages récents, Python est orienté objet. Chaque variable est un objet, qui connaît sa propre classe, et contient des méthodes qui s'appliquent aux informations qu'elle contient. Il est possible de faire hériter ses propres classes de tous les types du langages, y compris des types « de base » comme les entiers, flottants, booléens et chaînes de caractères.
Toutefois, Python n'exige pas que chaque variable ou chaque type soit déclaré et utilisé comme un objet muni de méthode. Un programme exécutable n'est pas présenté comme l'instance d'une classe – même s'il l'est.
Python permet l'héritage multiple, quoique cette pratique soit à réserver à des programmeurs avertis.

Python permet une programmation impérative : il est possible d'utiliser une programmation orientée objet avec Python, mais ce n'est pas indispensable. Il est possible de se limiter à une programmation composée de tests, boucles et fonctions, sans méthodes ni héritage.

Python est fortement typé : toutes les variables sont typées et connaissent leur type. Il est possible d'interroger le type d'un paramètre reçu avant d'effectuer un traitement le concernant, ce qui permet soit d'adapter le traitement en question (en transformant un paramètre de type *str* reçu en type numérique, par exemple), soit de lever une exception s'il est incompatible.

Python est typé dynamiquement : le type d'une variable est déterminé par son contenu, et peut donc être modifié à chaque affectation dans cette variable. Il n'est pas possible de connaître le type d'une variable lors de la compilation du programme, c'est-à-dire l'analyse syntaxique qui produit le bytecode. Les incompatibilités de type entre opérateurs et opérandes ne sont donc détectées qu'à l'exécution [16]. Il n'est pas utile de déclarer les variables avant de les utiliser, ce qui allège le code.

Python permet la surcharge d'opérateurs : les opérateurs et certains mots-clefs du langage peuvent être enrichis pour admettre comme opérandes les types créés par le développeur.

Python assure la gestion de la mémoire : l'allocation de la mémoire pour un objet est automatique à sa création et Python récupère automatiquement la mémoire inutilisée grâce au ramasse-miettes qui restitue les zones de mémoire laissées libres suite à la destruction des objets.

Python sait interagir avec d'autres langages : il est possible de lier un programme python avec des programmes écrits dans d'autres langages. Ainsi les bibliothèques scientifiques

15. Il suffit d'écrire ce que doit faire le programme (en anglais), d'ajouter des « : » pour bien expliquer, et d'indenter correctement. Le programme fonctionne.

16. Il convient de lire attentivement les messages d'erreur, de localiser la ligne où se trouve l'incompatibilté et de *ne pas la modifier* tout de suite. La source probable de l'erreur se trouve sans doute à l'un des endroits où des valeurs sont affectées aux opérandes.

de Python contiennent du code Fortran compilé, et les bibliothèques graphiques sont souvent écrites en C ou C++. Pour fonctionner de façon indépendante de la plate-forme, les codes binaires des ces bibliothèques doivent être disponibles sur la machine cible, de la même manière que la machine virtuelle.

Il est aussi possible d'appeler directement des fonctions et programmes installés sur le système d'exploitation. Cela, évidemment, rend le programme python nettement moins portable : il convient de bien documenter quels sont les programmes requis. Il est ainsi possible d'utiliser Python dans un but d'administration système.

Il y a plus d'une manière de faire : Python permet de nombreux moyens d'atteindre un même objectif. Ces moyens se traduiront par des facilités d'écriture, de lecture, voire même comme marqueurs culturels... Ils peuvent naturellement induire des différences dans le temps d'exécution.

```
>>> tab = [0, 1, 3, 7, -1]
>>> tabr = []
>>> for x in tab:
...   tabr.append(x*(x-1))
...
>>> print(tabr)
[0, 0, 6, 42, 2]
>>> [x*(x-1) for x in tab]
[0, 0, 6, 42, 2]
>>> map(lambda x : x*(x-1), tab)
[0, 0, 6, 42, 2]
>>> import numpy as np
>>> list(np.array(tab)*(np.array(tab)-1))
[0, 0, 6, 42, 2]
```

Les quatre techniques ci-dessus permettent toutes de calculer $x(x-1)$ pour tous les éléments x d'une liste python, et de placer les résultats dans une liste. Aucune [17] n'est à comprendre comme « la bonne » méthode.

1.4.3 Développement d'un programme python

La mise en œuvre d'un programme python se fait en deux temps :

Écriture du programme : un *éditeur de texte*, comme par exemple *emacs*, *vi*, *notepad++*, ou *gedit* permet d'écrire le code source. Ces éditeurs enregistrent du texte brut (mais en permettant éventuellement d'en maîtriser l'encodage), et font bénéficier le développeur d'une aide à l'écriture dans le langage Python : couleurs spécifiques pour les mots-clefs, les commentaires, et certains types de constantes ; aide à la gestion de l'indentation des blocs de code. Il est fortement recommandé de sauvegarder le fichier avec l'extension *.py* pour son interprétation par la machine virtuelle python ; on pourra omettre l'extension si le fichier est un script exécutable en ligne de commande.

Exécution du programme : la machine virtuelle python, accessible via la commande `python` ou `python3`, permet d'exécuter un programme Python, en interprétant son contenu. Elle prend en paramètre le nom du fichier texte écrit en langage Python. Elle laisse en général, et selon les options utilisées, un fichier *.pyc* ou *.pyo* dans le même répertoire que le fichier *.py*.

17. Le mode interprété de la machine virtuelle se charge de l'affichage. Le premier choix est l'application la plus directe d'un pseudo-code traitant la création de la nouvelle liste à partir de l'ancienne. L'objectif du cours est que chacun soit à l'aise avec cette technique. La seconde fonctionne par compréhension de liste ; c'est la plus spécifique du langage Python. La troisième est issue du lambda-calcul, et peut être naturelle à ceux qui ont une expérience en O'Caml. La quatrième s'appuie sur la bibliothèque de calcul matriciel, pour une nouvelle version des opérateurs `*` et `-`.

Remarque : ne pas confondre la notion d'éditeur de texte et de traitement de texte. *Word* ou *Libreoffice* ne sont pas des éditeurs de texte ! Le traitement de texte ne produit pas du texte brut, mais un texte agrémenté d'informations de mise en page et de structure, parfaitement impossible à compiler ! Leurs fonctions d'exportation en texte brut sont en général peu robustes, et ils ne fournissent aucune aide à l'écriture dans un langage informatique. Ils sont par contre de bons supports à l'écriture en langage naturel.

Remarque : il est possible d'utiliser un éditeur dédié appelé IDE [18] pour éditer, exécuter et déboguer un programme python. Certains IDE sont libres et disponibles gratuitement, comme par exemple *eclipse* [19], qui est compétent pour de nombreux autres langages, ou *spyder* [20], particulièrement adapté pour les travaux scientifiques. D'autres environnements tels que PyCharm [21] se déclinent en version libre et gratuite [22] et en version fermée et payante [23]. L'IDE *PyCharm* sera utilisé dans une large part des travaux pratiques.

1.4.4 Les différentes éditions et versions de Python

La licence sous laquelle Python est distribué se stabilise en 2001 sur une licence libre compatible avec la GPL [24]. Il s'agit d'un logiciel libre : vous pouvez modifier votre interpréteur Python et distribuer cette version modifiée. Toutefois, cette licence n'entraîne aucune contrainte quant à la licence qui régit les programmes écrits en Python.

Le modèle objet est unifié avec la version 2.2 : les types de base (`int`, `float`, `bool`) deviennent des objets comme les autres. Ils ont des méthodes, et il est possible d'en hériter. La dernière version de cette branche est la 2.7.11 de décembre 2015.

La version 3.0 vise à éliminer des redondances dans les fonctionnalités de Python. Il s'agit de diminuer le nombre de façons de réaliser une même opération, et donc la complexité du langage comme de sa machine virtuelle. Une conséquence est que la version 3.0 n'est pas compatible avec les versions précédentes : elle ne peut pas exécuter du code Python 2.x. Certaines bibliothèques scientifiques n'ont pas encore achevé cette transition et *spyder3*, la version de l'IDE basée sur cette version, n'est pas encore disponible dans toutes les distributions Linux. Il existe un script, `2to3`, qui permet de transformer automatiquement du code de Python 2.x en Python 3.

18. Integrated Development Environment
19. http://www.eclipse.org
20. https://github.com/spyder-ide
21. https://www.jetbrains.com/pycharm/download
22. généralement appelée version communautaire.
23. généralement appelée version professionnelle.
24. La *GNU General Public License* est la plus importante des licences sous lesquelles sont distribués les logiciels libres ; elle est écrite et maintenue par la *Free Software Foundation*. La license de Python est la *Python Software Foundation License* (PSFL), inspirée de la licence BSD de Berkeley.

L'homme a toujours considéré qu'il était plus intelligent que les dauphins sous prétexte qu'il avait inventé toutes sortes de choses — la roue, New York, les guerres, etc. — tandis que les dauphins quant à eux, n'avaient jamais rien su faire d'autre que prendre du bon temps.
Mais, réciproquement, les dauphins s'étaient toujours crus bien plus intelligents que les hommes — et précisément pour les mêmes raisons.

Douglas Adams

2

Éléments de syntaxe du langage Python

Sommaire

2.1 Notion de variable

Dans un programme, il est constamment nécessaire de stocker provisoirement des valeurs. Il peut s'agir de données issues du disque dur, fournies par l'utilisateur (tapées au clavier), etc. Il peut aussi s'agir de résultats (intermédiaires ou définitifs) obtenus par le programme. Ces données peuvent être de plusieurs types : des entiers, des réels, du texte, etc. Dès qu'il est nécessaire de stocker une information au cours d'un programme, on utilise une variable.

Dans l'ordinateur, physiquement, il existe un emplacement de mémoire, repéré par une adresse binaire, pour chaque variable ; elle y est stockée sous forme d'octets. Une variable est donc une zone de la mémoire (définie par un nom aux yeux d'un programmeur) dans laquelle une valeur est stockée.

La plupart des langages actuels se chargent, entre autres rôles, d'épargner au programmeur la gestion fastidieuse des emplacements mémoire et de leurs adresses. Le programmeur se contente de décrire la variable qu'il veut utiliser, et le compilateur (ou l'interpréteur) se charge de sa représentation en mémoire.

2.1.1 Création de variable

En Python, il n'est pas nécessaire de déclarer (définir le type de) une variable avant de pouvoir l'utiliser. Il suffit d'affecter (assigner) une valeur au nom de la variable pour que celle-ci soit automatiquement créée avec le type qui correspond au mieux à la valeur fournie. En effet, la déclaration d'une variable et son initialisation (c'est-à-dire la première valeur que l'on souhaite stocker dedans) se fait en même temps. En Python, tout est objet et la variable n'est qu'une référence à l'objet (entier, réel, chaîne de caractères, ...).

Exemple :

```
1 >>> a = 42     #  a est un entier
2 >>> b = 33.5   #  b est un réel
3 >>> c = "une chaîne de caractères" # c est une chaîne de caractères
4 >>> a
5 42
```

Le nom de la variable est un identificateur qui doit respecter les contraintes suivantes :
— doit commencer par un caractère parmi : Lu, Ll, Lt, Lm, Lo, Nl, ou le tiret bas (_) ;
— peut contenir tous les caractères précédents ainsi que les caractères : Mn, Mc, Nd et Pc ;
— ne doit pas être un *mot-clé* parmi les 33 *mots clés* de Python 3 (voir la table 2.7).

Avec :

Lu les lettres majuscules (uppercase letters) ;

Ll les lettres minuscules (lowercase letters) ;

Lt les lettres de titre (titlecase letters) ;

Lm les lettres de modification comme les exposants (modifier letters) ;

Lo les autres lettres (other letters) ;

Nl les lettres représentant des nombres comme les chiffres romains (letter numbers) ;

Mn les caractères sans dimension comme les accents (nonspacing marks) ;

Mc des symboles additionnels (spacing combining marks) ;

Nd les chiffres (decimal numbers) ;

Pc des connecteurs (connector punctuations).

Le site http://www.fileformat.info/info/unicode/category/index.htm recense les caractères unicode par catégorie.

Remarques :

— il est conseillé de se limiter aux caractères directement accessibles sur tous les types de clavier ;
— un nom d'identificateur est sensible à la casse (les majuscules et les minuscules sont différentes) ;
— un nom d'identificateur peut contenir des lettres accentuées, ou des caractères non latins, mais ceci est déconseillé (permis par la syntaxe du langage, mais pas par les conventions de nommage, cf. section A.3) ;
— l'annexe A définit les conventions de nommage généralement utilisées ;
— il est fortement conseillé de donner des noms parlants aux variables, tout en évitant les noms excessivement longs [1] ;
— le nom composé d'un unique underscore _ référence la dernière valeur calculée en mode interactif. Il ne faut pas l'utiliser pour stocker de l'information, mais lire ce qui s'y trouve peut être intéressant. Elle peut aussi servir à recevoir temporairement de l'information que l'on ne souhaite pas conserver.

```
>>> 2 + 4
6
>>> _ * 7
42
>>> tab = [3, 9, 8]
>>> tab.sort()
>>> mintab, _, maxtab = tab
```

2.1.2 Typage des variables

Python est typé dynamiquement, par opposition à d'autres langages comme Java, C, C++ qui sont typés statiquement. Cela signifie qu'il identifie lui-même les types de données associés aux variables et que ces dernières peuvent changer de type au cours de l'exécution du programme.

Remarques :

— Pour connaître le type d'une variable ou expressions, on peut utiliser la fonction standard *type()* ;
— La fonction *print()* permet d'afficher la valeur d'une variable. Quelques exemples d'utilisation de cette fonction peuvent être consultés en section 2.5.2.

Exemple :

```
>>> i = 42
>>> type(i)
<class 'int'>
>>> i = 'indice'
>>> type(i)
<class 'str'>
>>> i = 42.0
>>> type(i)
<class 'float'>
>>>
```

1. Les exemples C.3 et C.4 p. 230 contiennent des noms excessivement longs.

2.1.3 Types numériques

Les types de base sont les types intégrés au langage. Ils sont directement compréhensibles par l'interpréteur et ne nécessitent aucune information supplémentaire pour être utilisables.

Python comme d'autres langages de programmation, dispose de deux types numériques : les entiers (*int*[2]) et les réels (*float*). À ces deux types numériques de base s'ajoutent les nombres complexes (*complex*) et les booléens (*bool*).

Les entiers sont représentés en complément à deux (voir paragraphe 1.3.2). Le type entier est du genre « bigint », dans lequel la représentation n'est limitée que par l'espace mémoire (pas de risque de débordement). Tant que l'entier reste suffisamment petit (32, 64 ou 128 bits selon l'architecture matérielle) il est traité de façon cohérente avec le microprocesseur. Au-delà, les calculs deviennent nettement plus lents.

```
>>> i = 42
>>> i
42
>>> math.factorial(i)
1405006117752879898543142606244511569936384000000000
```

Les réels sont représentés en virgule flottante (voir paragraphe 1.3.3). Le type réel utilisé en python 3 est en double précision codée en 64 bits. La simple précision (codée sur 32 bits) n'est implémentée que par `numpy` (section 4.3.1). Ce type reprend le stockage des nombres du type *double* du langage C ou du langage Java, et est cohérent avec le traitement de ces nombres par la plupart des microprocesseurs. Il offre une précision bien meilleure que la simple précision car le gain en terme d'utilisation de mémoire et de temps processeur est négligeable par rapport aux consommations d'un programme Python. En particulier, la précision numérique[3] est 2.22E-16 (2^{-52}) en double précision, et 1.19E-7 (2^{-23}) en simple précision.

```
>>> pi = 3.14
>>> pi
3.14
>>> freq = 10e9 # 10 milliards
>>> freq
10000000000.0
>>> eps = 2**(-52)
>>> eps
2.220446049250313e-16
>>> 1 + eps == 1
False
>>> 1 + eps/2 == 1
True
```

Les nombres complexes en Python sont formés d'un couple de réels qui accepte les mêmes opérateurs qu'un réel[4]. On utilise un suffixe j pour regrouper deux valeurs composant la partie réelle et la partie imaginaire du nombre complexe. Un nombre complexe possède deux attributs en lecture seule *.real* et *.imag* et une méthode *.conjugate()*. Le listing 2.1 illustre la manipulation de nombres complexes.

Listing 2.1 – Manipulation de nombres complexes

```
>>> nb = 10 + 5j
```

2. Le type *int* en Python 3 remplace les types *int* et *long* en Python 2.x.
3. le plus petit nombre qu'il est possible de distinguer de 1, noté ε.
4. L'ensemble $\mathbb{C}$ est un corps au même titre que $\mathbb{R}$.

```
>>> nb.real
10.0
>>> nb.conjugate()
(10-5j)
>>> nb = complex(10, 5)
>>> nb
(10+5j)
```

2.1.4 Types de base non numériques

Booléens

Une variable booléenne peut prendre deux valeurs : *True* ou *False*. Ces derniers sont l'équivalent des entiers 0 et 1 en algèbre de Boole. Les variables booléennes supportent toutes les opérations logiques de base : et, ou inclusif, ou exclusif, négation (voir la section 2.3.2). En Python, tout ce qui n'est pas faux est vrai (voir quelques exemples en page 45).

```
>>> b = True
>>> b and False
False
>>> not b
False
```

None

La valeur spéciale *None*, qui est l'unique valeur possible du type *NoneType*, représente l'absence de valeur. La plupart des opérations sont invalides sur *None*, mais elle peut se convertir en valeur booléenne *False* pour les opérateurs logiques. Il est possible d'affecter explicitement *None* à une variable, afin de s'assurer qu'elle ne contient plus rien, sans la détruire avec `del`. Une fonction qui ne renvoit rien (absence d'instruction `return`) renvoit en fait *None*.

```
>>> x = print(42)
42
>>> print(x, " : ", type(x))
None  :  <class 'NoneType'>
```

Chaînes de caractères

Une chaîne de caractères (type *str*) est une séquence de caractères délimitée par des guillemets simples `'` ou doubles `"` ou dans une série de trois guillemets simples `'''` ou doubles `"""`. On peut indifféremment utiliser l'un ou l'autre type de guillemets mais il faut utiliser le même délimiteur en début et fin de chaîne.

Exemple :

```
>>> ch = "Hello World"
>>> l  = """ Ma phrase est
en plusieurs
lignes"""
>>> print(ch, l)
Hello World Ma phrase est
en plusieurs
lignes
```

Remarque : Il n'existe pas de type caractère (type *char* dans d'autres langages) en Python, mais uniquement des chaînes contenant un seul caractère.

Accès aux éléments d'une chaîne : Pour accéder aux éléments d'une séquence, notamment une chaîne de caractères, il suffit de préciser l'indice du caractère (index) entre crochets. L'index dans une chaîne commence toujours à 0 et ne doit pas dépasser l'indice de dernier caractère.

Python permet d'utiliser des indices (positifs ou négatifs) pour accéder aux différents composants (caractères) d'une séquence (chaîne) ou pour en récupérer une partie (ou tranche : *slice*). Avec des indices négatifs, on accède aux éléments de la chaîne à partir de la fin.

Si dans une tranche (représentée par l'opérateur de *slicing ':'* une valeur manque d'un côté ou de l'autre de *':'*, elle désigne tout ce qui suit ou tout ce qui précède. Le dernier élément de la tranche est exclu. Par exemple `ch[:-1]` du listing 2.2 donne tous les caractères jusqu'à l'avant dernier.

Listing 2.2 – Accès aux éléments d'une chaîne de caractères

```
1 >>> ch = "Hello World"
2 >>> print(ch[0], ch[-1], ch[-2], ch[2:5], ch[6:], ch[:-1])
3 H d l llo World Hello Worl
4 >>> ch[11]
5 Traceback (most recent call last):
6   File "<pyshell#77>", line 1, in <module>
7     ch[11]
8 IndexError: string index out of range
```

Dans le listing 2.2, pour la chaîne de caractères ch, `ch[-1]` est le dernier caractère, `ch[-2]` l'avant dernier, etc. `ch[2:5]` correspond à la partie de ch depuis le caractère d'indice *2* inclus jusqu'au caractère d'indice *5* exclus. Dans l'instruction de la ligne 5, un message d'erreur nous signale que l'index *11* est en dehors de la plage d'index de la chaîne *ch*.

Remarque : une fois créée, une chaîne de caractères ne peut pas être modifiée. En Python, elle est une séquence non modifiable (*non-mutable*). Il est uniquement possible de créer une nouvelle chaîne à partir de l'ancienne comme dans la ligne 7 du listing 2.3.

Listing 2.3 – Modification d'une chaîne de caractères

```
1 >>> ch = "Hello World"
2 >>> ch[0] = 'h' # tentative de modifier le premier caractère
3 Traceback (most recent call last):
4  File "<pyshell#76>", line 1, in <module>
5    ch[0] = 'h' # tentative de modifier le premier caractère
6 TypeError: 'str' object does not support item assignment
7 >>> ch1 = "h" + ch[1:]
8 >>> ch1
9 'hello World'
```

Déterminer la taille d'une chaîne : grâce à la fonction standard *len()*.

```
1 >>> ch = "Hello World"
2 >>> ch[5:len(ch)]
3 ' World'
4 >>> len(ch)
5 11
```

Opérations sur les chaînes : Les chaînes de caractères peuvent êtres répétées avec l'opérateur '*' et concaténées avec l'opérateur '+'.

```
>>> ch = 'Hell' + 'o'
>>> ch * 2 +'World'
'HelloHelloWorld'
```

Les sections 8.1 et 8.6 décrivent des fonctionnalités avancées du type *str* et de traitement de chaînes de caractères.

2.1.5 Listes

En Python, une liste est représentée par le type *list*. Pour créer une liste, il faut placer un ensemble d'éléments entre crochets `[]` et les séparer par des virgules ','. Ces éléments peuvent être de types différents.

Le type liste de Python sert aussi à représenter ce que de nombreux langages (C, Java, Perl, ...) nomment tableau (*array*). La bibliothèque `numpy` (cf. section 4.3) dispose d'un type *array* plus contraint, dont toutes les cases sont de même type, et dont la taille est fixée.

Listing 2.4 – Utilisation de listes

```
>>> mois = ['janvier', 31, 'fevrier', 28, 'mars']
>>> mois
['janvier', 31, 'fevrier', 28, 'mars']
>>> type(mois)
<class 'list'>
```

Dans le listing 2.4, la valeur de la variable *mois* est une liste dont les éléments sont de types 'int' et 'str'. Par exemple, le premier, le troisième et le cinquième éléments sont des chaînes de caractères alors que le deuxième et le quatrième élément sont des entiers.

Accès aux éléments d'une liste

Comme dans le cas des chaînes de caractères, les indices des listes commencent par 0. On peut ainsi y accéder, les découper, les concaténer, les imbriquer, etc. Les opérateurs '+' et '*' présentés précédemment, ont le même effet sur les listes que sur les chaînes de caractères.

```
>>> mois = ['janvier', 31, 'fevrier', 28, 'mars']
>>> print(mois[0], mois[-1], mois[1 :-1])
janvier mars [31, 'fevrier', 28]
>>> mois[ :2] + ['octobre',  31, 'juin']
['janvier', 31, 'octobre', 31, 'juin']
>>> zeros = [0] * 5 #  Création d'une liste de 5 zéros
>>> zeros
[0, 0, 0, 0, 0]
```

Une liste, à la différence des chaînes de caractères, est un type mutable. L'affectation d'une tranche d'éléments est possible (cela peut changer la taille d'une liste). La fonction intégrée *len()*, que nous avons déjà présentée dans le cadre des chaînes de caractères, s'applique également aux listes.

```
>>> mois = ['janvier', 31, 'fevrier', 28, 'mars']
>>> len(mois)
5
>>> mois[2 :5] = ['fevrier', 29, 'avril', 30]
>>> mois
['janvier', 31, 'fevrier', 29, 'avril', 30]
```

```
>>> len(mois)
6
```

Remarque : Le concept de liste est assez différent du concept de « tableau » (*array*) ou d'autres types indicés que l'on rencontre dans d'autres langages de programmation dont les éléments sont de même type, et dont la taille est en général déterminée à la création.

Liste de listes

Il est possible d'emboîter des listes, c'est-à-dire de créer des listes contenant d'autres listes.

Remarque : il est important de noter que l'affectation d'une liste à partir d'une liste préexistante, ne crée pas une copie de la liste mais seulement une référence.

Listing 2.5 – Affectation de liste

```
1  >>> p = [2, 4, 6]
2  >>> q = [p, 8, 10]
3  >>> q
4  [[2, 4, 6], 8, 10]
5  >>>  len(q)
6  3
7  >>> q[0]
8  [2, 4, 6]
9  >>> q[0][1]
10 4
11 >>> q[0][1] = 'quatre'
12 >>> p
13 [2, 'quatre', 6]
```

Dans le listing 2.5, *p* et *q[0]* font référence au même objet. En ligne 9, *q[0][1]* se réfère au deuxième élément (indice 1) du premier élément (indice 0) de la liste référencée par la variable *q*.

Manipulation de liste

Il est possible via les méthodes de la classe *list* ou des instructions Python de modifier, d'interroger ou de trier une liste[5]. Citons :

append(el) permet d'ajouter l'élément *el* à la fin[6] de la liste.

extend(li) permet d'ajouter l'ensemble des éléments de la liste *li* à la suite de la liste.

insert(pos, el) permet d'insérer l'élément *el* à l'index *pos* spécifié.

remove(el) permet de supprimer la première occurrence de la valeur *el*.

pop([pos]) permet de renvoyer et de supprimer l'élément d'index *pos*, ou le dernier élément si aucune valeur n'est passée à *pop*.

del est une instruction intégrée qui permet de supprimer un élément dont on connaît la position.

sort() trie la liste. Elle ne renvoit pas de nouvelle liste.

reverse() inverse l'ordre des éléments de la liste.

Les complexités des différentes opérations sont décrites dans la table D.2 p. 242.

5. Pour connaître la totalité des méthodes du type *list*, utiliser l'instruction `help(list)`
6. Cette méthode est utile pour remplir une liste dans une boucle.

```
>>> mots = ['alice', 'livre', 'banane']
>>> mots.append('cahier')
>>> mots
['alice', 'livre', 'banane', 'cahier']
>>> mots.insert(1, 'python')
['alice', 'python', 'livre', 'banane', 'cahier']
>>> mots.remove('livre')
['alice', 'python', 'banane', 'cahier']
>>> del mots[1]
>>> mots
['alice', 'banane', 'cahier']
>>> mots.pop()
'cahier'
>>> mots.extend(['miroir', 'coeur'])
>>> mots
['alice', 'banane', 'miroir', 'coeur']
>>> mots.sort()
>>> mots
['alice', 'banane', 'coeur', 'miroir']
>>> mots.reverse()
>>> mots
['miroir', 'coeur', 'banane', 'alice']
```

Les opérateurs + (concaténation) et * (répétition) fonctionnent sur les listes comme sur les chaînes de caractères. Toutefois, il convient de prendre en compte le fait qu'une liste est définie par une référence, et que sa répétition ne crée pas de nouvelle liste. Le listing 2.6 illustre ce phénomène.

Listing 2.6 – Répétition de liste

```
>>> liste = [['Zaphod', 'President']] * 2
>>> liste
[['Zaphod', 'President'], ['Zaphod', 'President']]
>>> liste = liste + ['Whale']
>>> liste
[['Zaphod', 'President'], ['Zaphod', 'President'], 'Whale']
>>> liste[0][0] = 'Zappy'
>>> liste
[['Zappy', 'President'], ['Zappy', 'President'], 'Whale']
```

2.1.6 Tuples

Les tuples en Python ressemblent aux listes mais ils ne sont pas modifiables (*non-mutable*) et sont créés avec des parenthèses au lieu de crochets. La fonction **len()** est toujours présente, et le tuple peut se parcourir à l'aide d'une boucle *for* (section 2.4.4).

Listing 2.7 – Manipulation de tuples

```
>>> x = ('a', 2, 3.0)
>>> x
('a', 2, 3.0)
>>> x[2]
3.0
>>> x[0:2]
('a', 2)
>>> x[2] = 4
```

```
9  Traceback (innermost last):
10 File "<stdin>", line 1, in ?
11 TypeError: object doesn't support item assignment
```

Accès aux éléments d'un tuple

L'affectation et l'accès sont identiques aux opérations correspondantes dans les chaînes de caractères et les listes. Comme les tuples sont immuables, Python renvoie un message d'erreur (ligne 8 du listing 2.7) lors de toute tentative de modification d'un de ses éléments. Pour ajouter un élément (ou le modifier), il faut créer un autre tuple.

Listing 2.8 – Modification de tuple

```
1 >>> x = ('a', 2, 3.0)
2 >>> x + (2,)
3 ('a', 2, 3.0, 2)
4 >>> x = 5, 6, 7
5 >>> x
6 (5, 6, 7)
```

Dans la ligne 2 du listing 2.8, notons que la création d'un tuple d'un seul élément (de valeur *2*), nécessite l'utilisation d'une virgule. Il faut utiliser une syntaxe avec une virgule (*element,*) Ceci permet d'éviter une ambiguïté avec les parenthèses qui délimitent une expression. Il est également possible de créer des tuples sans utiliser des parenthèses, comme dans la ligne 4.

Recombinaison de listes et de tuples

La fonction intégrée `zip` permet de former des groupes constitués des éléments de même indice issus de différentes structures itérables, qui peuvent êtres des tuples ou des listes. Elle crée un itérateur qui renvoit successivement ces tuples (voir section 7.4). Cet itérateur peut être converti en liste comme en tuple.

```
>>> x = [1, 2, 3]
>>> y = (4, 5, 6)
>>> zipped1 = zip(x, y)
>>> list(zipped1)
[(1, 4), (2, 5), (3, 6)]
>>> zipped2 = zip(x, y)
>>> tuple(zipped2)
((1, 4), (2, 5), (3, 6))
```

2.1.7 Dictionnaires

Un dictionnaire [7] Python correspond à une collection non ordonnée d'éléments (objets). L'accès à un élément quelconque d'un dictionnaire se fait au travers d'un index spécifique appelé *clé*. Un dictionnaire fonctionne donc par couple (clé : valeur). Comme dans une liste, les éléments d'un dictionnaire peuvent être de n'importe quel type [8]. Par contre, les clés doivent obligatoirement être d'un type non mutable [9]. L'accès à la taille d'un dictionnaire se fait à l'aide de la fonction *len()* déjà vue. Les complexités des différentes opérations sont décrites dans la table D.4 p. 243.

7. Dans de nombreux langages, cette structure est appelée *table de hachage*, et implémentée dans un type baptisé *hashtable*.

8. Les éléments d'un dictionnaire peuvent être : des valeurs numériques, des chaînes, des listes, des tuples, des dictionnaires, et même aussi des fonctions, des classes ou des instances.

9. Il est donc impossible d'utiliser une liste ou un dictionnaire en tant que clé.

Création d'un dictionnaire

La création d'un dictionnaire se fait en utilisant une paire d'accolades `{}`. On peut également créer un dictionnaire à partir d'une liste de couples (clé, valeur) ou d'un autre dictionnaire passé en argument au constructeur *dict()*.

Listing 2.9 – Création d'un dictionnaire

```
1 >>> eleves = {'nom': 'Dupont', 'prenom' : 'Alice', 'option' : 'SPID'}
2 >>> eleves['age'] = 22
3 >>> eleves['prenom'] = 'Pierre'
4 >>> eleves
5 {'option': 'SPID', 'prenom': 'Pierre', 'age': 22, 'nom': 'Dupont'}
6 >>> eleves['prenom']
7 'Pierre'
```

L'opérateur `[]` en ligne 2 du listing 2.9 se trouvant à gauche d'une affectation sert à associer une valeur (l'entier *22*) à la clé représentée par la chaîne de caractères 'age'. Il est à noter que dans cette instruction, la clé 'age' n'était pas encore présente dans le dictionnaire *eleves*. Le couple ('age', 22) est alors ajouté et la taille de dictionnaire est modifiée. Dans le cas où la clé est déjà présente (comme pour l'instruction de la ligne 3), la valeur (*'Alice'*) précédemment liée à la clé *'prenom'* est remplacée par (*'Pierre'*). Pour récupérer une valeur associée à une clé, il suffit de préciser la clé concernée (cf. ligne 6).

Supprimer des clés d'un dictionnaire

Comme pour les listes, pour supprimer des éléments d'un dictionnaire, il est possible d'utiliser :
- l'instruction **del**;
- la méthode de dictionnaire **pop(clé)**. Contrairement aux listes, la clé est requise. Le dictionnaire n'ayant pas de relation d'ordre, il n'est pas possible d'agir sur un élément par défaut.

```
1 >>> eleves = {'nom': 'Dupont', 'prenom' : 'Alice', 'option' : 'SPID'}
2 >>> del eleves['nom']
3 >>> eleves
4 {'option': 'SPID', 'prenom': 'Alice'}
5 >>> eleves.pop('option')
6 'SPID'
7 >>> eleves
8 {'prenom': 'Alice'}
```

Accès aux éléments d'un dictionnaire

L'accès à la liste des clés, des valeurs et des couples (clé, valeur) est assuré par les méthodes :

keys() retourne dans un ordre quelconque[10] la liste de toutes les clés utilisées dans le dictionnaire. Pour tester si une clé quelconque est présente dans un dictionnaire, on peut utiliser l'expression `key in dico` qui retourne *True* si *key* est présente dans le dictionnaire *dico* et *False* sinon.

```
1 >>> eleves = {'nom': 'Dupont', 'prenom' : 'Alice', 'option' : 'SPID'
    }
2 >>> 'prenom' in eleves
3 True
4 >>> 'age' in eleves
5 False
```

10. Pour obtenir une liste ordonnée (triée), utiliser la méthode *sort()* à la liste des clés obtenue.

values() retourne la liste de toutes les valeurs stockées dans le dictionnaire.

items() renvoie la liste des couples (clé, valeur)

```
>>> eleves = {'nom': 'Dupont', 'prenom' : 'Alice', 'option' : 'SPID'
  }
>>> eleves.keys()
dict_keys(['option', 'prenom', 'nom'])
>>> eleves.values()
dict_values(['SPID', 'Alice', 'Dupont'])
>>> eleves.items()
dict_items([('option', 'SPID'), ('prenom', 'Alice'), ('nom', 'Dupont
  ')])
```

Remarque : il est possible d'utiliser les méthodes : *keys()*, *values()* et *items ()* avec une boucle *for* pour parcourir la liste renvoyée.

Counter

La classe *Counter* est un dictionnaire spécialisé qui permet de calculer de manière simple et efficace le nombre d'occurrences d'éléments. Le listing 2.10 contient un exemple d'utilisation de cette classe. La variable *pays* contient une liste de pays, et la variable *cpt* permet de compter le nombre d'occurrences de chaque pays.

Listing 2.10 – Utilisation de Counter

```
>>> from collections import Counter
>>> print(pays)
['Pays-Bas' 'Thailande' 'Italie' 'Pays-Bas' 'Espagne' 'Islande' 'Espagne'
 'Suède' 'Pays-Bas' 'Pays-Bas' 'France' 'France' 'Espagne' 'France'
 'Pays-Bas' 'Portugal' 'Belgique' 'Italie' 'Portugal' 'France' 'Belgique'
 'France' 'Espagne' 'Belgique']
>>> cpt = Counter(pays)
>>> cpt
Counter({'France': 5, 'Pays-Bas': 5, 'Espagne': 4, 'Belgique': 3, '
  Portugal': 2, 'Italie': 2, 'Islande': 1, 'Thailande': 1, 'Suède': 1})
>>> cpt['France']
5
```

2.1.8 Ensembles

Python propose deux classes permettant de manipuler les ensembles : *set* et *frozenset*. Ces classes sont des collections dans lesquelles chaque élément ne peut apparaître qu'une fois. Elles supportent les opérations d'appartenance (`in`), d'union (`|`), d'intersection (`&`), de différence (`-`) et de différence symétrique (`^`). Le type *frozenset* est immuable, et il faut donc lui affecter tous ses éléments lors de l'initialisation.

Comme dans le cas des dictionnaires, ces deux classes permettent un accès à leurs éléments en temps constant. La table D.5 p. 243 contient la complexité des différentes opérations sur les ensembles. Le listing 2.11 contient quelques exemples d'utilisation de ces classes.

Listing 2.11 – Exemple d'utilisation des classes *set* et *frozenset*

```
>>> s1 = set()
>>> s1.add(1)
>>> s1.add(2)
>>> print(s1)
```

```
{1, 2}
>>> s1.add(1)
>>> print(s1)
{1, 2}
>>> 1 in s1
True
>>> s2 = set([2, 3])
>>> print(s2)
{2, 3}
>>> s1 | s2
{1, 2, 3}
>>> s1 & s2
{2}
>>> s1 - s2
{1}
>>> s1 ^ s2
{1, 3}
>>> s3 = frozenset(s1)
>>> s3
frozenset({1, 2})
>>> s3.add(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

2.1.9 Transtypage

Dans certains cas, il est nécessaire de convertir une variable d'un certain type en variable d'un autre type. Cette opération s'appelle transtypage (ou *cast* en anglais). Pour transtyper une variable en Python, le principe est de confier la variable existante au constructeur du type souhaité, en utilisant le nom du type de destination comme une fonction (voir le chapitre 5 sur les objets et les constructeurs).

Utiliser *int()*, *float()*, *complex()*, *bool()* permet ainsi le transtypage vers les types *int*, *float*, *complex*, *bool*. La fonction standard *str* permet de convertir les différents objets en chaînes de caractères.

Remarques :

- lorsqu'on réalise un transtypage d'une chaîne de caractères vers un entier, il est possible de spécifier un second paramètre à *int()* pour préciser la *base* dans laquelle la valeur est calculée. Ceci n'est bien entendu possible que si la chaîne représente bien un entier dans la base demandée ;
- dans le cas des nombres complexes, il est possible d'utiliser deux réels (partie réelle et partie imaginaire) ;
- le booléen *False* (resp. *True*) est transtypé vers l'entier 0 (resp. 1) ;
- pour les conversions des réels vers des entiers, il est possible d'utiliser *int()*, mais également quelques fonctions du module math [11] : *ceil()*, *floor()*, *trunc()*. La conversion effectuée par *int(x)* correspond à *trunc(x)*, c'est-à-dire *floor(x)* si x est positif, et *ceil(x)* si x est négatif.
- la fonction *str()* permet de convertir la plupart des types vers des chaînes de caractères.

```
>>> int(12.5)
12
>>> int('125')
```

11. Le module *math* est décrit en section 4.1.

```
125
>>> float(125)
125.0
>>> complex(1, -5)
(1-5j)
>>> int("101010", 2)
42
>>> complex("12 + .5j")
(12+0.5j)
>>> int(True)
1
>>> bool(0)
False
>>> bool(124)
True
>>> str(3 * (1j + 4))
'(12+3j)'
```

2.2 Littéraux

Un littéral est une constante, qui peut être utilisé dans une expression ou pour initialiser une variable. Il peut être de plusieurs types :

— entier ;
— réel ;
— complexe ;
— booléen ;
— chaîne de caractères.

Chaque type de littéral peut posséder plusieurs écritures, qui définissent ce qui est lexicalement correct pour le langage Python :

les entiers peuvent s'écrire normalement, et ils sont alors en base 10 (exemple : `3`, `-42`).

Ils peuvent être précédés de *0x* et sont alors écrits en base 16 (*nombre hexadécimal*). Exemple : `0x9A3E`.

Ils peuvent être précédés de *0o* et sont alors écrits en base 8 (*nombre octal*)[12]. Exemple : `0o102`, qui vaut 66 en décimal.

Ils peuvent être précédés de *0b* et sont alors écrits en base 2 (*nombre binaire*). Exemple : `0b101010`, qui vaut 42 en décimal.

Il est autorisé de mélanger ces écritures dans une même expression (`42+0x42-0o42` vaut 74) mais c'est rarement une bonne idée.

les réels ou nombres à virgule flottante (*float*) peuvent s'écrire avec ou sans exposant. Exemple : `3.1416`, `6.022E23` (vaut 6.022×10^{23}), `1e-2` (vaut 0.01). S'il n'y a pas d'exposant, le caractère `'.'` est le marqueur du nombre à virgule flottante : `42` est un entier, mais `42.0` et `42.` sont des flottants.

les nombres complexes (*complex*) disposent d'une partie réelle et d'une partie imaginaire qui sont des *float*. Un nombre est complexe s'il dispose d'une partie imaginaire, fut-elle nulle. Elle peut être issue d'un calcul n'impliquant que des réels. La partie imaginaire est identifiée par le suffixe `j`. Exemples : `1j`, `-4+0j`, `3**0.5/2+0.5j`, `(-1)**0.5`.

les booléens s'écrivent `True` ou `False`.

12. De nombreux langages, comme C, Java ou Python 2, utilisent simplement le préfixe 0 plutôt que 0o pour identifier l'octal. Cette syntaxe a conduit et conduira de nombreux programmeurs à écrire en octal par inadvertance.

les chaînes de caractères s'écrivent entre guillemets ou entre apostrophes ; on choisira l'un ou l'autre pour la facilité d'écriture du contenu de la chaîne. Exemple : `"Une chaîne de caractères"`, `"C'en est une autre"`, `'L\'expression "dauphin" aussi'`. Il est possible de représenter les caractères par leur numéro unicode, écrit en hexadécimal. Exemple : `'\u0065'` (caractère e) ou `"1945 : de Stra\u00dfburg à Strasbourg"`, représentant `"1945 : de Straßburg à Strasbourg"`. Certains caractères spéciaux sont accessibles grâce à des séquences d'échappement (par ex. `'\n'` pour le retour chariot). Quelques séquences d'échappement sont fournies dans la table E.2 page 249.

2.3 Opérateurs

2.3.1 Affectation

L'affectation consiste à attribuer une valeur à une variable. Il s'agit d'une opération fondamentale sur les variables. Elle est représentée en Python par l'opérateur « = ».

```
>>> a = 2          # création d'une variable de type int de valeur 2
>>> a = a+1        # a reçoit le resultat du calcul a+1 (ici 3)
>>> b = (a > 2)    # b référence l'objet True si a>2 et False sinon
```

Réaffectation

La réaffectation consiste à associer un nouvel objet à un nom déjà utilisé. Le système de gestion de la mémoire se charge de mettre à jour les compteurs de références de l'ancien objet et du nouveau, et de la suppression de l'ancien objet si nécessaire.

```
>>> a = 2            # création d'une variable a
>>> a = 'bonjour'    # réaffectation de la variable a
```

Certaines affectations particulièrement courantes peuvent s'écrire sous forme abrégée :

```
>>> a = 1
>>> a += 2        # equivalent à : a = a + 2
```

Il est possible de condenser les affectations impliquant les autres opérations de la même manière : `+=`, `-=`, `*=`, `/=`, `//=`,`%=`, `**=`, `>>=`, `<<=`, `&=`, `^=`, `|=`.

Remarque : à la différence d'autres langages, Python n'accepte pas post-incrémentation, post-décrémentation, pré-incrémentation et pré-décrémentation. Ainsi, les expressions `i++`, `i--`, `++i`, et `--i` ne sont pas syntaxiquement correctes.

Affectation multiple

En Python, Il est possible d'affecter en une seule instruction la même valeur à plusieurs variables ou plusieurs valeurs à plusieurs variables. Il est ainsi possible d'interchanger les valeurs de plusieurs variables en une seule instruction.

```
>>>  x = y = z = 0    # affectation de la valeur 0 à toutes les
                      # variables x, y, et z
>>> m, n, p = 1 ,"deux",3.25
>>> m
1
>>> n
'deux'
>>> p
```

```
3.25
>>> m, n = n, m      # échanger le contenu de deux variables
>>> m
'deux'
>>> n
1
>> x, y, z = [1 ,"deux",3.25] # Affection depuis une liste
>>> x
1
>>> y
'deux'
>>> z
3.25
>>> lst = [1, "deux", 3.25]    # création d'une liste
>>> tete, *reste = lst         # l'étoile * désigne le
                               # reste des valeurs situées à droite
>>> tete
1
>>> reste
["deux",3.25]
```

Python effectue les affectations demandées de la gauche vers la droite. Si on affecte plusieurs valeurs à une même variable, celle-ci contiendra donc la dernière valeur.

```
>>> cat, cat = ('dead', 'alive')
>>> cat
'alive'
```

2.3.2 Comparaisons

Python propose les opérateurs de comparaisons classiques comme indiqué dans la table 2.1.

Remarque : le test d'égalité s'écrit « == » ; il faut le différentier de l'affectation qui s'écrit « = » ;

Les opérateurs logiques disponibles en Python sont référencés dans la table 2.2.

Remarque : le test d'égalité `==` concerne le contenu, le test `is` concerne les références (lorsque le type s'y prête), et n'est donc vrai que si les deux opérandes font référence à la même zone mémoire. La vérité du test `is` entraîne celle de `==`, mais l'inverse n'est pas vrai. Le listing 2.12 illustre cette différence.

Listing 2.12 – Différence entre `==` et `is`

```
>>> 2 is 2
True
>>> 2+2 == 4
True
>>> 2+2 is 4
True
>>> [2+2] == [4]
True
>>> [2+2] is [4]
False
```

Opérateur	Exemple	Signification
`>`	`a > 10`	strictement supérieur
`<`	`a < 10`	strictement inférieur
`>=`	`a >= 10`	supérieur ou égal
`<=`	`a <= 10`	inférieur ou égal
`==`	`a == 10`	égal à
`!=`	`a != 10`	différent de
`is`	`a is b`	a et b représentent le même objet
`is not`	`a is not b`	a et b ne représentent pas le même objet
`in`	`a is s`	a fait partie de la séquence s
`not in`	`a not in s`	a ne fait pas partie de la séquence s

TABLE 2.1 – Opérateurs Python

Remarque : le test d'égalité concerne l'intégralité du contenu des opérandes, et ne fait pas d'approximation. Ceci peut avoir des conséquences fâcheuses, les derniers bits des nombres en virgule flottante relevant souvent d'un bruit numérique. Il convient de s'assurer que l'écart entre les deux nombres est négligeable [13] comme illustré par le listing 2.13...

Listing 2.13 – Comparaison de nombres réels et bruit numérique

```
>>> 2.2*3 == 3.3*2
False
>>> abs(2.2*3 - 3.3*2) <= 1e-15 * max(abs(2.2*3), abs(3.3*2))
True
```

Opérateur	Exemple	Signification
`not`	`not a`	NON logique
`and`	`a and b`	ET logique : évaluation paresseuse
`or`	`a or b`	OU logique : évaluation paresseuse
`&`	`a & b`	ET logique
`^`	`a ^ b`	OU exclusif logique
`|`	`a | b`	OU logique

TABLE 2.2 – Opérateurs logiques Python

Définition 2.1 (Évaluation paresseuse)**.** *Une* évaluation paresseuse *est une évaluation qui s'arrête dès que le résultat de l'expression est connu. Par exemple, pour l'expression* `False and b`*, une évaluation paresseuse ne va pas déterminer la valeur de b (quelle que soit la valeur de b,* `False and b` *est faux), alors qu'une évaluation normale va déterminer la valeur de b puis conclure.*

Sauf cas exceptionnel, il est préférable d'utiliser l'évaluation paresseuse. Elle permet d'évaluer la validité de paramètres avant de les utiliser, dans une même expression :

Listing 2.14 – Évaluation paresseuse

```
if i>0 and i<len(tab)-1 and tab[i]>tab[i+1]:
    tab[i], tab[i+1] = tab[i+1], tab[i]
```

13. Le langage C dispose d'une telle fonction, nommée `fcmp`. Une fonction `is_close` devrait arriver dans les futures versions de Python (PEP 0485).

2.3.3 Opérations arithmétiques

Python propose bien entendu toutes les opérations arithmétiques de base : addition (`+`), soustraction (`-`), multiplication (`*`), division (`/`). On trouve aussi la division entière (`//`), le reste de division entière ou modulo (`%`), l'élévation à la puissance (`**`).

L'opérateur `@` est utilisé pour la multiplication matricielle [14]. Aucun type Python de base ne supporte cet opérateur. Voir section 4.3.5 pour son utilisation.

```
>>> 5/2
2.5
>>> 5//2
2
>>> 5%2
1
```

Remarque : en Python, comme le montre le listing 2.15, la division par zéro produit une exception. Il est cependant possible de manipuler les valeurs infinies ou une valeur spéciale nommée NaN [15]. Le listing 2.16 et la table 2.3 représentent les opérations disponibles avec ces deux valeurs.

Listing 2.15 – Division par zéro

```
>>> 5/0
Traceback (most recent call last):
  File "<pyshell#76>", line 1, in <module>
    5/0
ZeroDivisionError: division by zero
>>>
```

Listing 2.16 – Exemples de manipulation de valeurs infinies et de valeurs non définies

```
>>> 1 / float('inf')
0.0
>>> float('inf') - float('inf')
nan
>>> 0 + float('nan')
nan
>>> 1 * float('inf')
inf
>>> 0 * float('inf')
nan
```

X	**Y**	**X/Y**	**X%Y**	**X + Y**
x : valeur finie	$\pm\infty$	0	x	$\pm\infty$
$\pm\infty$	valeur finie	$\pm\infty$	NaN	$\pm\infty$
$+\infty$	$+\infty$	NaN	NaN	$+\infty$
$+\infty$	$-\infty$	NaN	NaN	NaN
Valeur finie ou infinie	NaN	NaN	NaN	NaN

TABLE 2.3 – Arithmétique infinie

14. À partir de Python 3.5.
15. Not A Number.

2.3.4 Opérateurs sur les bits

Il y a deux types d'opérateurs au niveau bits : opérateurs bit à bit et opérateurs de décalage. Pour comprendre les opérateurs sur les bits, il faut représenter les nombres en binaire. Par exemple, 3 s'écrit 11, et 5 s'écrit 101.

Un *ou bit à bit* entre ces deux nombres vaut donc 111 (en partant du poids fort, c'est-à-dire de la gauche, 1 ou 0 vaut 1, 0 ou 1 vaut 1 et 1 ou 1 vaut 1), c'est-à-dire 7. Et *et bit à bit* entre ces deux mêmes nombres vaut 001 en binaire, c'est-à-dire 1.

Les opérateurs de décalage permettent de décaler les chiffres binaires d'un nombre entier de bits vers la droite ou la gauche. Cela permet de multiplier ou de diviser par une puissance de 2.

La table 2.4 contient les opérateurs permettant d'agir sur les bits.

Opérateur	Exemple	Signification
`<<`	`a << n`	décalage à gauche de n bits (multiplication par 2^n)
`>>`	`a >> n`	décalage à droite de n bits (division par 2^n)
`~`	`~a`	NON binaire
`&`	`a & b`	ET binaire
`^`	`a ^ b`	OU exclusif binaire
`|`	`a | b`	OU binaire

TABLE 2.4 – Opérateurs sur les bits

Considérons une variable entière i. La table 2.5 décrit la valeur de i avec différents décalages.

Variable	Valeur entière	Valeur binaire
`i`	6	0b110
`i<<2`	24	0b11000
`i>>2`	1	0b1

TABLE 2.5 – Opérateurs binaires

Remarque : Les arrondis requis par >> se font par valeur inférieure. Il convient d'être prudent face aux nombres négatifs, même si le bit de signe du codage en complément à deux n'est pas décalé par cet opérateur.

```
>>> -(6>>2)
-1
>>> (-6)>>2
-2
```

2.3.5 Priorité des opérateurs

La table 2.6 définit les priorités des opérateurs de Python (du plus prioritaire au moins prioritaire). Les parenthèses ayant une forte priorité, elles peuvent servir à modifier l'ordre d'interprétation des opérateurs.

2.4 Instructions du langage Python

Quelques règles de base sur les instructions Python :

— Python est sensible à la casse ;

Nom de l'opérateur	**Symbole**
Parenthèses, crochets et accolades	`( )`, `{ }`, `[ ]`
Accès à des éléments d'une séquence	`x[i]`, `x[i:j]`, `x(i)`
Puissance	`**`
Plus, moins et NON unaire	`+x`, `-x`, `~x`
Multiplication, multiplication matricielle, division, division entière, et modulo	`*`, `@`, `/`, `//`, `%`
Addition et soustraction	`+`, `-`
Décalage	`<<`, `>>`
ET binaire	`&`
OU EXCLUSIF binaire	`^`
OU binaire	`|`
Comparaison	`<`, `>`, `<=`, `>=`, `==`, `!=`, `in`, `not in`, `is`, `is not`
NON booléen	`not`
ET booléen	`and`
OU booléen	`or`

TABLE 2.6 – Priorité des opérateurs

— un bloc de code commence par un caractère ':' et est indenté plus à droite que le bloc le contenant ; cette indentation est identique sur tout le bloc ;
— une instruction peut tenir sur plusieurs lignes. Il faut soit terminer chaque ligne par '\', soit que le saut de ligne soit entouré de délimiteurs comme `()` ou `[]`.

2.4.1 Commentaires

Les commentaires sont des instructions écrites en langage naturel par le programmeur pour ajouter des remarques concernant le code. Ils ne sont pas pris en compte par le compilateur.

Les caractères qui suivent un caractère '`#`' appartiennent au commentaire, sauf si ce caractère se trouve dans une chaîne de caractère, délimitée par des apostrophes ou des guillemets.

Listing 2.17 – Exemples de commentaires et d'instructions multi-lignes

```
# Ce calcul est un peu long : écrivons-le sur deux lignes
x = 12.345 * -21.879 + \
    451.25587 - 139.159615
  # On écrit le résultat (commentaire valide bien que mal indenté)
print(x)
print(12.345 * -21.879 +            # Pas besoin de \ en fin de ligne
      451.25587 - 139.159615)
print('Ce qui suit # est un commentaire') # mais pas s'il est dans une
                                          # chaîne de caractères
```

Commentaires de documentation automatique

Ce type de commentaire sert à générer une documentation HTML du code python. Les commentaires doivent commencer par « """ » et se terminer par « """ ». Voir l'annexe B pour plus d'informations. Exemple :

```
def foo(var1, var2):
    """ Explication brève de ce que fait la méthode

    Arguments :
    var1 : type, signification
    var2 : type, signification

    Retour :
    type, description

    """
    ...         # nombreuses instructions construisant une variable v
    return v
```

Un commentaire de ce type est une *docstring*. Elle peut être associée à une classe, une fonction, ou une méthode, en faisant commencer la docstring sur la ligne suivant `def` ou `class`. Si elle est associée à une méthode *foo*, l'instruction `help(foo)` fournit [16] la docstring appropriée.

Si elle est associée à une classe *A* et que *obj1* est une instance de cette classe [17] (créée par `obj1 = A()`), les instructions `help(A)` ou `help(obj1)` fournissent la docstring associée à la classe *A*. Cette docstring est composée du commentaire placé directement après la ligne de `class`, mais aussi de la liste des méthodes de la classe et de leurs *doctrings*.

2.4.2 Mots-clés

Un mot-clé est un identificateur ayant un sens défini par le langage Python. Ces mots-clés seront pour la plupart décrits au cours du polycopié (voir l'index page 261).

La liste des mots-clés du langage est disponible dans le module *keyword*.

and	as	assert	break	class	continue
def	del	elif	else	except	False
finally	for	from	global	if	import
in	is	lambda	None	nonlocal	not
or	pass	raise	return	True	try
while	with	yield			

TABLE 2.7 – Mots-clés en Python

Remarque : `print` n'est pas un mot-clé du langage Python dans sa version 3, mais une fonction intégrée. Il s'agissait d'un mot-clé en Python 2. C'est pour cela que les anciens codes écrivent `print x` là où il faut désormais écrire `print(x)`.

Si `x` est une chaîne de caractère, `print` l'écrit sur la sortie standard. Sinon, `print` en fait une chaîne de caractère en évaluant `str(x)`, avant de l'écrire. Si `print` reçoit plusieurs arguments, ils sont écrits successivement, sur une même ligne, séparés par un espace. Le paramètre optionnel

16. Au besoin, un certain nombre de lignes et de blancs inutiles sont retirés lors de la transformation du code source par l'instruction `help`.
17. Se référer au chapitre 5 ; ne pas s'inquiéter de ce paragraphe avant de traiter ce chapitre.

`end` vaut par défaut `'\n'`, ce qui provoque un retour à la ligne. Il est possible de l'empêcher en lui donnant une autre valeur.

Listing 2.18 – Utilisations de print

```
print(4)
print(2)
print(4, 2)
print(4, end="")
print(2)
```

Son exécution donne :

```
4
2
4 2
42
```

Remarque : les types de base, comme *int*, *float*, *str*, *bool* ou *list* ou *dict* ne sont pas des mots-clés. Il est donc *syntaxiquement* autorisé d'appeler une variable de cette manière. Le faire serait une *très* mauvaise idée.

2.4.3 Branchements conditionnels

Comme la quasi-totalité des langages de développement, Python propose un ensemble d'instructions qui permettent d'organiser et de structurer les traitements. Elles concernent les branchements conditionnels et les boucles. L'usage de ces instructions est similaire à celui rencontré dans leur équivalent dans d'autres langages.

Les branchements conditionnels permettent d'exécuter différentes parties du code en fonction de conditions booléennes. Python utilise les instructions *if*, *else* et *elif* pour gérer ces branchements.

L'instruction *if* permet de tester une condition, puis d'exécuter une partie du code si elle est vérifiée et une autre partie sinon. L'instruction *else* définit les instructions à exécuter en cas de condition fausse. Exemple :

Listing 2.19 – Exemple de `if`

```
if expressionBooleenne1:
    action1
else:
    if expressionBooleenne2:
        action2
    else:
        action3
```

Dans cet exemple, si l'expression booléenne *expressionBooleenne1* vaut `True`, alors *action1* est exécutée. Sinon, si *expressionBooleenne2* vaut `True`, alors *action2* est exécutée. Sinon, *action3* est exécutée.

Il est possible de condenser la partie `else: if:` du code 2.19 en `elif:`, ce qui permet de tester successivement des conditions incompatibles sans créer de nouveau niveau d'imbrication [18] :

18. Cette construction permet de retrouver un comportement proche d'une conditionnelle de type `switch ... case` populaire dans d'autres langages. Des alternatives sont d'associer les différentes actions à des instances d'une pattern (section 5.4) ou d'utiliser un dictionnaire de fonctions dont les clés sont fortement liées aux variables déterminant les expressions booléennes (section 2.1.7).

Listing 2.20 – Exemple de `elif`

```
if expressionBooleenne1:
    action1
elif expressionBooleenne2:
    action2
else:
    action3
```

Les conditions booléennes peuvent être des variables booléennes, des tests, ou des combinaisons logiques d'expressions booléennes.

Listing 2.21 – Exemple d'expression booléenne

```
b = fonction_test()
i = calcul_complique()
if b and (i>42):
    appel_fonction()
```

Dans le listing 2.21, *fonction_test()* renvoie un booléen [19]. *appel_fonction()* sera exécuté si b vaut `True` et si $i > 42$. Si une de ces conditions n'est pas remplie, rien n'est exécuté (il n'y a pas de *else*).

Il est possible de profiter de la paresse des évaluations des expressions booléennes. Dans le code 2.22, l'opérande de gauche de `and` est évalué en premier (c'est la simple lecture de b). S'il est `False`, l'expression totale vaut `False`, et l'opérande de droite n'est pas calculé. Le code suivant est donc plus rapide dans ce cas :

Listing 2.22 – Exemple d'expression booléenne paresseuse

```
b = fonction_test()
if b and (calcul_complique()>42):
    appel_fonction()
```

Le listing 2.23 se sert de la paresse de l'évaluation de la condition pour s'assurer que `2/i` soit une expression valide.

Listing 2.23 – Autre exemple d'expression booléenne paresseuse

```
if i!=0 and (2/i)==1:
    print('ok')
```

Il est possible d'utiliser une expression non booléenne dans le cadre d'une instruction *if*. Dans ce cas, Python va tester sa valeur de vérité, selon une procédure adaptée [20] au type de l'expression :

nombre : `False` si le nombre vaut 0, `True` sinon.

chaîne de caractères : `False` si la chaîne est vide (donc vaut `""`), `True` sinon, y compris [21] si la chaîne vaut `"0"`.

liste, tuple ou dictionnaire : `False` si la structure est vide (donc vaut `[]`, `()` ou `{}`), `True` sinon, y compris s'il s'agit de `[""]`, de `(0)` ou de `{False: None}`.

autre : `False` s'il s'agit de `None`, `True` sinon.

19. Quoique que toute variable puisse être associée à une condition booléenne.
20. En pratique, en confiant l'expression au constructeur de `bool`.
21. La mauvaise maîtrise du type des variables manipulées conduit facilement à ce genre de situation involontaire.

Remarques :
— une instruction *if* n'est pas obligatoirement suivie de l'instruction *else* ;
— le code suivant, bien que syntaxiquement correct, est à proscrire :

```
if condition:
    pass
else:
    action
```

La forme suivante sera préférée :

```
if not condition:
    action
```

Remarque : *pass* est une instruction vide qui ne fait rien.

Conditions multiples : Il est possible d'utiliser successivement des opérateurs de comparaison dans une même expression. Les deux écritures suivantes sont équivalentes :

Listing 2.24 – Condition multiple

```
if 0 <= a < 10:
if (0 <= a) and (a < 10):
```

En pratique, chaque opérateur de comparaison va construire sa valeur de vérité en fonction des deux opérandes l'entourant. L'expression totale est la conjonction de ces valeurs : il suffit d'un `False` pour que toute l'expression vaille `False`. Il n'y a pas d'obligation à respecter les relations de transitivité sur les opérateurs de comparaison, mais cela aide grandement le lecteur du code, voire son auteur.

Listing 2.25 – Condition multiple perverse

```
>>> 2 <= 5 >= 0 <= 3
True
```

2.4.4 Boucles

Une boucle est une structure de contrôle permettant d'exécuter certaines actions plusieurs fois, le nombre de fois étant déterminé par une condition. Il existe plusieurs types de boucles en Python, bien qu'il soit possible de tout écrire avec *while*. Dans de nombreuses situations, *for* conduit à un code plus pratique et plus simple.

Boucle while

La boucle la plus universelle est la boucle *while*, qui exécute une action tant qu'une condition est réalisée. Sa syntaxe générale est la suivante :

```
while condition:
    actions
```

Exemple 2.1 (boucle while)**.** *Cet exemple affiche successivement toutes les valeurs entières de 0 à 9 inclus. Remarquons qu'à la fin de la boucle* i *vaut 10, c'est-à-dire la première valeur pour laquelle la condition du* while *est fausse.*

Listing 2.26 – Exemple de boucle `while`

```
i=0
while i<10:                 # boucle tant que i<10
    print(i)                # affiche la valeur de i
    i = i+1                 # incrémente i
# maintenant i vaut 10
```

La condition de la boucle *while* est toujours évaluée avant la première itération. Certains langages prévoient un format de boucle spécifique dans lequel la condition n'est évaluée qu'après la première itération. Ce n'est pas le cas de Python, mais ce comportement peut facilement être reconstruit. L'évaluation des expressions booléennes étant paresseuse, *condition* n'est pas évaluée lors de la première itération du programme 2.27. Elle peut donc faire appel à des variables qui ne sont initialisées que lors de la première exécution de *actions*.

Listing 2.27 – Équivalent de boucle `do ...while`

```
flag = True
while flag or condition:
    flag = False
    actions
```

Sorties de boucle

Il est possible d'ajouter un bloc *else* après une boucle while, qui sera exécuté, une seule fois, lorsque la condition devient fausse.

L'instruction *continue* met fin à l'itération courante de la boucle et passe directement à l'itération suivante, en réévaluant la condition.

L'instruction *break* met fin à la boucle et en sort, sans traiter l'éventuel bloc *else* associé.

La fonction intégrée `exit` met fin à l'exécution du programme.

Listing 2.28 – Utilisations de `continue` et `break`

```
x, y = 10, 10
while (x, y) not in positions_bombes:
    x, y = demander_nouvelle_position()
    if not position_valide(x,y):
        continue
    if condition_victoire():
        break
else:
    print("Boum")
    exit()
print("Gagné !")
```

Leur utilisation permet éventuellement de limiter les niveaux d'imbrication du code lorsque de nombreuses conditions sont testées pour isoler des cas particuliers :

Listing 2.29 – Trois tests imbriqués

```
# Author : Lauritz V. Thaulow (CC BY-SA)

for x, y in zip(a, b):
    if x > y:
        z = calculate_z(x, y)
        if y - z < x:
            y = min(y, z)
```

```
            if x ** 2 - y ** 2 > 0:
                lots()
                of()
                code()
                here()
```

Listing 2.30 – Utilisation de `continue` pour les cas particuliers

```
# Author : Lauritz V. Thaulow (CC BY-SA)

for x, y in zip(a, b):
    if x <= y:
        continue
    z = calculate_z(x, y)
    if y - z >= x:
        continue
    y = min(y, z)
    if x ** 2 - y ** 2 <= 0:
        continue
    lots()
    of()
    code()
    here()
```

Boucle for

L'instruction `for` permet de parcourir une collection (une *list*, un *tuple*, un *dict* ou même un *set*) ou d'invoquer un itérateur (voir section 7.4) jusqu'à ce qu'il ait terminé son rôle.

Sa syntaxe est :

```
for variable in collection:
    actions
```

La *variable* prend successivement toutes les valeurs contenues dans la *collection*. Le bloc *actions* est exécuté pour chacune de ces valeurs ; il peut contenir des instructions `break` ou `continue`, qui ont les mêmes effets que dans une boucle `while`.

Listing 2.31 – Exemple de boucle for (avec utilisation maladroite de `continue`)

```
""" Affiche les éléments de la liste qui divisent 42 """
lst = [6, 7, 2, 8, 42, 14]
for x in lst:
    if 42%x != 0:
        continue
    print(x)
```

Si la collection est un dictionnaire, les valeurs prises sont les clés du dictionnaire. Pour travailler sur les valeurs associées, il convient de les chercher à chaque itération, ou d'utiliser `value()` :

Listing 2.32 – Exemple de boucle for sur un dictionnaire

```
seq_aa = {'muscle_1': 'AAPMQVT', 'souris_1':'ACMNQRVQNC', 'souris_2':'
  ACMNQTPDLC'}

# boucle sur les clés
for peptide in seq_aa:
    ARN_decode(seq_aa[peptide])
```

```
6
7  # boucle sur les valeurs
8  for code in seq_aa.values():
9      ARN_decode(code)
```

Itérateur range

L'itérateur `range()` fournit successivement des valeurs entières. Il est particulièrement utile pour parcourir les indices d'une liste, lorsque le contenu seul n'est pas suffisant.

Listing 2.33 – Utilisation de range pour extraire un indice

```
1  lst = [6, 7, 2, 8, 42, 14]
2  val_max = float('-inf')
3  ind_max = -1
4  for i in range(len(lst)):
5      if val_max < lst[i]:
6          val_max = lst[i]
7          ind_max = i
8  print("Le plus grand élément est {}, en position {}".format(val_max,
     ind_max))
```

Utilisé avec un seul paramètre, `range(n)` prend successivement les valeurs de 0 à $n-1$. Il fournit donc tous les indices d'une liste de taille n.

Utilisé avec deux paramètres, `range(deb, fin)` prend successivement les valeurs de *deb* à *fin-1*.

Utilisé avec trois paramètres, `range(deb, fin, inc)` prend successivement les valeurs de *deb* à *fin-1*, avec un incrément de *inc*. Cela permet aussi de parcourir une liste en sens inverse avec `for i in range(len(tab)-1, -1, -1)`. Les deux boucles suivantes sont équivalentes :

Listing 2.34 – Comparaison de range() et while

```
1  for i in range(2,45,5):
2      print(i)
3
4  i = 2
5  while i<45:
6      i += 5
7      print(i)
```

Remarque : les deux modes de fonctionnement, collection ou itérateur, n'en sont en fait qu'un seul : l'instruction `for` crée un itérateur autour de son argument si ce n'est pas un itérateur. Il est équivalent d'écrire :

```
1  for x in lst:
```

ou d'écrire :

```
1  for x in iter(lst):
```

Utilisation d'enumerate

La fonction `enumerate()` permet de parcourir une séquence en récupérant les indices et les valeurs des éléments de la séquence. Elle peut simplifier le parcours de séquences en évitant d'utiliser la fonction `range()`. Le listing 2.35 illustre le parcours de liste avec la fonction `range()` puis avec la fonction `enumerate()`.

Listing 2.35 – Exemple d'utilisation d'`enumerate`

```
>>> saisons = ['Printemps', 'Été', 'Automne', 'Hiver']
>>> for i in range(len(saisons)):
...     print(i, saisons[i])
0 Printemps
1 Été
2 Automne
3 Hiver
>>> for i, s in enumerate(saisons):
...     print(i, s)
0 Printemps
1 Été
2 Automne
3 Hiver
```

Le paramètre optionnel `start` permet de configurer la valeur initiale du compteur d'enumerate ; le listing 2.36 illustre son utilisation.

Listing 2.36 – Paramètre *start* de la fonction `enumerate`

```
>>> for i, s in enumerate(saisons, start=1):
...     print(i, s)
1 Printemps
2 Été
3 Automne
4 Hiver
```

Le listing 2.37 reprend l'exemple du listing 2.33 en remplaçant le `range` par un `enumerate`.

Listing 2.37 – Utilisation d'enumerate pour parcourir une liste

```
lst = [6, 7, 2, 8, 42, 14]
val_max = float('-inf')
ind_max = -1
for i, val in enumerate(lst):
    if val_max < val:
        val_max = val
        ind_max = i
print("Le plus grand élément est {}, en position {}".format(val_max,
  ind_max))
```

2.5 Premiers programmes python

Après avoir vu les notions élémentaires de langage permettant l'écriture d'un algorithme, il est important, pour ne pas travailler uniquement en mode interactif, d'appréhender la réalisation et l'exécution de programmes python en mode script. Cette partie présente quelques programmes rudimentaires illustrant l'utilisation de scripts. Et, comme il est d'usage dans beaucoup de manuels d'apprentissage de langage informatique, le premier programme sera le « Hello World ».

2.5.1 Hello World

Exemple 2.2 (Programme hello world)**.** *L'objectif de ce programme est uniquement d'afficher les deux mots* Hello World. *Cela se fait à l'aide de la fonction* print(). *Concrètement, il suffit, à l'aide de n'importe quel éditeur de texte, d'enregistrer dans un fichier que l'on nommera par exemple* hello.py *(le nom du fichier peut être quelconque mais le suffixe doit être* py*), le texte suivant :*

Listing 2.38 – Hello World

```
1 print("Hello, World!")
```

Un bon programmeur ayant la bonne habitude d'ajouter de nombreux commentaires à ses codes utilisera les symboles # ou """ pour inscrire des commentaires sur une ou plusieurs lignes.

Listing 2.39 – Hello World

```
1 # Ceci est mon premier programme avec un commentaire sur une ligne.
2 """ Ceci est mon premier programme
3     avec un commentaire fait sur plusieurs lignes. """
4 print("Hello, World!")
```

Gestion des accents

Les langues accentuées, en particulier le français, nous obligent souvent à utiliser des caractères ne faisant pas partie de la table ASCII[22]. Il faut donc faire appel à des tables de codage plus riches. L'encodage le plus répandu sur les ordinateurs récents est UTF-8. On ajoutera au début (et impérativement à cette place) du fichier *hello.py* une ligne supplémentaire.

Listing 2.40 – Hello World

```
1 # -*- coding: utf-8 -*-
2 # Ceci est mon premier programme avec un commentaire sur une ligne.
3 """ Ceci est mon premier programme
4     avec un commentaire fait sur plusieurs lignes. """
5 print("Bonjour, Le monde français avec des accents é, è, ê, â, ... !")
```

Attention, même si la ligne ainsi ajoutée commence par le signe #, il ne s'agit pas d'un commentaire. L'interpréteur python fait jouer aux premières lignes présentées sous la forme de commentaires un rôle particulier. Ces premières lignes s'appellent le *shebang*.

Exécution

Dans tous ces cas, il est alors possible d'exécuter ce premier programme par la commande `python3 hello` ou `python3 hello.py`.

Si vous utilisez un IDE, la démarche est différente et propre à l'IDE en question. Il suffit en général de cliquer sur un bouton pour exécuter le programme.

Les utilisateurs de système d'exploitation Linux et MacOS peuvent utiliser le *shebang* pour rendre le programme exécutable (tapez uniquement `./hello.py`) (cf. section 1.4.1).

2.5.2 Passage de paramètres

Dans le script *hello.py* précédent, le programme, une fois lancé, effectue systématiquement la même tâche (un simple affichage d'une chaîne de caractères déterminée à l'avance) ce qui en limite considérablement la portée. Il est en général plus intéressant de transmettre au programme un minimum d'information susceptible d'en modifier le comportement. Dans les exemples suivants, l'objectif est de transmettre un ou plusieurs paramètres au programme selon deux approches différentes.

22. Voir l'annexe E pour plus de détails sur ASCII et unicode.

Paramètres fournis au lancement du programme

Exemple 2.3 (Programme hello_first_names). *Le programme doit maintenant dire* Hello *à une liste de personnes dont les prénoms sont communiqués au lancement du script, par exemple en tapant* `python3 hello_first_names.py Claire Michel Sophie Paul`. *Pour cela, nous utilisons la bibliothèque* sys *que nous appelons par la commande* import. *Cette bibliothèque définit une liste* argv *d'arguments (liste de chaînes de caractères) dont le premier terme* argv[0] *contient le chemin et le nom du fichier de script. Les autres termes sont les paramètres saisis au lancement.*

Listing 2.41 – Programme `hello_first_names.py`

```
# Hello with a list of first names

import sys

print('Number of arguments :', len(sys.argv), 'arguments.')
print('Number of parameters :', len(sys.argv[1:]), 'parameters.')

for first_name in sys.argv[1:]:
    print('Hello ', first_name)

print('Job done by', sys.argv[0])
```

Ainsi dans notre cas, nous obtenons:

```
$ python3 hello_first_names.py Claire Michel Sophie Paul
Number of arguments: 5 arguments.
Number of parameters: 4 parameters.
Hello  Claire
Hello  Michel
Hello  Sophie
Hello  Paul
Job done by hello_first_names.py
$
```

Les paramètres sont systématiquement transmis sous forme de chaînes de caractères. Pour des applications numériques, il conviendra de convertir ces chaînes en des variables de type numérique, en utilisant *float()* ou *int()* par exemple (voir la section 2.1.9 sur le transtypage).

Remarques : Par défaut, la fonction *print()* introduit un retour à la ligne après chaque appel. En Python 3, la fonction *print()* possède un second argument qui définit les fins d'écriture. On peut par exemple écrire `print('.', end="")`, et quand la liste de termes à écrire est longue, on peut même préciser `print('.', end="", flush=True)` pour éviter les problèmes de buffer.

Paramètres fournis en mode interactif

Il peut être plus intéressant de saisir les différents paramètres au cours de l'exécution d'un programme.

Exemple 2.4 (Programme hello first names interactif). *Le programme doit maintenant demander un prénom et dire* Hello *à cette personne. Si la saisie contient une chaîne vide le programme s'arrête. Cette saisie interactive nécessite l'emploi de la fonction* input().

Listing 2.42 – Programme `hello_first_names_interactive.py`

```
# Hello to a given first name

```

```
while True:
    first_name=input("Who are you? ")
    if first_name:
        print('Hello ', first_name)
    else:
        break
```

L'exécution du code donnera alors:

```
$ python3 hello_first_names.py
Who are you? Claire
Hello  Claire
Who are you? Michel
Hello  Michel
Who are you? Sophie
Hello  Sophie
Who are you? Paul
Hello  Paul
Who are you?
$
```

2.6 Fonctions

En théorie, muni des éléments de langage python fondamentaux et après avoir mis en œuvre nos premiers scripts, il est possible de coder n'importe quel algorithme aussi complexe soit-il. Toutefois, concrètement, il est illusoire de chercher à programmer un algorithme élaboré sans structurer son code à l'aide de fonctions et de sous-programmes.

Les fonctions permettent de découper le code et de factoriser certains groupes d'instructions souvent appelés. Par exemple, si notre programme manipule des listes, il peut être intéressant d'écrire une fonction affichant une liste. Le programmeur se contentera alors d'appeler cette fonction au lieu de faire un copier-coller [23] des instructions permettant cet affichage. Ceci présente de nombreux avantages : le code écrit est mieux structuré, plus court et plus facile à faire évoluer.

Remarque : Dans cette section, nous n'utiliserons pas le terme de *méthode* pour parler des fonctions. Le terme *méthode* ne sera employé qu'après avoir introduit les notions de programmation objet que nous aborderons au chapitre 5.

2.6.1 Déclaration d'une fonction

Exemple 2.5 (fonction `hello`). *En revenant à nos premiers scripts, on peut définir une fonction* hello(), *et réécrire le code de l'exemple 2.2.*

Listing 2.43 – Fonction simple

```
def hello():
    print("Hello, World!")

hello()
```

Dans ce cas, le corps principal du programme se résume à l'appel de la fonction hello(). *L'exécution de ce nouveau code donnera le même résultat que pour l'ancien.*

23. En informatique, le copier-coller est considéré comme une très mauvaise pratique. Il reflète généralement une mauvaise conception du programme.

Exemple 2.6 (passage d'un paramètre à une fonction)**.** *On peut également réécrire l'exemple 2.3 en introduisant une fonction. Cette fonction devra alors gérer un paramètre appelé* paramètre formel.

Listing 2.44 – Fonction à un paramètre

```
import sys

def hello(name):
    print('Hello ', name)


print('Number of arguments :', len(sys.argv), 'arguments.')
print('Number of parameters :', len(sys.argv[1:]), 'parameters.')

for first_name in sys.argv[1:]:
    hello(first_name)
```

Les paramètres du prototype d'une fonction sont appelés paramètres formels.

Il est possible d'utiliser des *variables locales* dans les fonctions. La portée de ces variables est limitée à la fonction (voir section 2.6.3).

Exemple 2.7 (Fonction numérique)**.** *Dans les deux exemples précédents, les fonctions affichaient quelque chose mais ne renvoyaient aucune valeur au programme (on parle alors souvent de procédure plutôt que de fonction). Considérons la fonction f définie par $f(x, y, z) = x^2 + y - 3z$. Cette fonction renvoie une valeur et nécessite trois paramètres.*

Listing 2.45 – Fonction numérique à plusieurs paramètres

```
def func_num(x, y, z):
    return x**2 + y + 3*z


a = 2
b = 1.3
c = -2.5

print('The result is : ', func_num(a, b, c))
```

On notera que, dans cet exemple, le programme introduit les variables numériques a, b *et* c.

Le renvoi d'une valeur se fait par le mot-clé `return`. Cela met fin automatiquement à la fonction.

Lorsque plusieurs valeurs sont retournées simultanément, elles sont placées dans un tuple, lequel peut être utilisé pour affecter plusieurs variables. Il faut s'assurer que le tuple est suffisamment long pour affecter toutes les variables.

Listing 2.46 – Fonction renvoyant plusieurs valeurs

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

def order2(tab):
    if len(tab)==1:
        return (tab[0], )
    if len(tab)>2:
        return None
    if tab[0]<tab[1]:
```

```
10          return tab[0], tab[1]
11      else:
12          return tab[1], tab[0]
13
14  print(order2([42]))
15  print(order2([9,6]))
16  mini, maxi = order2([9,6])
17  print("Petit:",mini, "; Grand:", maxi)
18  print(order2([4,2,1]))
19  mini, maxi = order2([42])
```

L'exécution de ce programe donne :

```
$ python ch2_return.py
(42,)
(6, 9)
Petit: 6 ; Grand: 9
None
Traceback (most recent call last):
  File "return.py", line 19, in <module>
    mini, maxi = order2([42])
ValueError: need more than 1 value to unpack
```

2.6.2 Typage des paramètres d'une fonction

En Python, on ne précise pas le type des paramètres. On travaille en typage dynamique. Une même fonction peut donc s'appliquer sur des données de types très différents. Par exemple:

Listing 2.47 – Écrire 3 fois

```
1  def afficher3fois(arg):
2      print(arg, arg, arg)
```

À l'exécution, on pourra obtenir pour des arguments différents:

```
1  >>> afficher3fois(5)
2  5 5 5
3  >>> afficher3fois('zut')
4  zut zut zut
5  >>> afficher3fois([5, 7])
6  [5, 7] [5, 7] [5, 7]
7  >>> afficher3fois(6**2)
8  36 36 36
```

2.6.3 Portée des variables

La portée des variables est une notion fondamentale en programmation et très importante quand on manipule des fonctions, des classes ou lorsqu'on importe des modules. La notion de portée permet de définir l'accès et la visibilité aux variables utilisées dans un programme Python et ainsi introduire la notion de *variable globale* et *variable locale* définies dans un *espace des noms*.

Espace des noms

En Python on manipule des variables (références vers des objets puisque tout est objet en Python), qui sont définies dans une hiérarchie d'espaces de noms, et référencent des objets. Nous distinguons trois espaces des noms : espace des noms internes (`__builtins__`) espace des noms

locaux et espace des noms globaux. Cette hiérarchie d'espace des noms est centrale dans Python. Elle est constituée comme suit :

— l'espace des noms internes `__builtins__`, donne accès à toutes les fonctions de base disponibles en standard dans Python sans rien avoir à importer ;
— l'espace des noms globaux d'un module donne accès à toutes les variables, les fonctions, les classes définies dans ce module, ou importées ;
— l'espace des noms locaux d'un appel de fonction ou de méthode donne accès aux arguments et variables locales à cet appel.

Remarques :

— La fonction standard *dir()*, appelée sans paramètre dans une console Python, retourne la liste des noms définis dans l'espace courant de noms (noms locaux quand appelée dans une fonction).
— Lorsque Python rencontre une variable, il va traiter la résolution de son nom avec des priorités particulières : d'abord il cherche si la variable est locale, puis si elle n'existe pas localement, il vérifiera si elle est globale et enfin si elle n'est pas globale, il cherche si elle est interne.

2.6.4 Variables locales et globales

Quand nous définissons des variables à l'intérieur du corps d'une fonction par exemple, elles ne sont accessibles qu'à la fonction elle-même. On dit que ces variables sont des *variables locales* à la fonction et elles ne sont pas visibles à l'extérieur de celle-ci.

Par ailleurs, les variables définies à l'extérieur d'une fonction sont des variables globales. Leur contenu reste *visible* de l'intérieur d'une fonction mais non modifiable. Prenons un exemple pour illustrer ce fonctionnement.

```
>>> def maville():
        ville = 'Brest'
        print (ville)
>>> maville()
Brest
>>> print(ville)
Traceback (most recent call last):
File "<pyshell#52>", line 1, in <module>
  print(ville)
NameError: name 'ville' is not defined
```

On peut constater facilement que lorsque Python exécute le code de la fonction à partir de notre module principal (dans le cas de cet exemple, il s'agit de l'interpréteur Python), il connaît le contenu de la variable *ville*. Par contre, quand il y accède de l'extérieur de cette fonction, il ne la connaît plus d'où le message d'erreur. Il est à noter qu'une variable passée en argument est considérée comme *variable locale* pour la fonction.

Les variables définies à l'extérieur d'une fonction sont considérées comme des *variables globales*. Elles sont visibles de l'intérieur d'une fonction, mais leur donner une nouvelle valeur ne les modifie pas ; une variable locale portant le même nom est créée.

Listing 2.48 – Accès à une variable locale et à une variable globale

```
>>> def maville():
        ville = 'Brest'
        print (ville, code)
>>> ville, code = 'Quimper', '29000'
>>> maville()
Brest 29000
```

```
>>> print(ville, code)
Quimper 29000
```

Dans la ligne 1 à la ligne 3 du listing 2.48, on définit la fonction *maville()* dans laquelle on définit la variable *ville* avec *'Brest'* comme valeur. Cette variable est considérée comme *variable locale* à la fonction. En ligne 4, on revient au niveau principal pour y définir les deux variables *ville* et *code* auxquelles on attribue les contenus 'Quimper' et '29000'. Ces deux variables définies au niveau principal seront donc des *variables globales*. Le même nom de variable 'ville' est utilisée pour définir une variable locale (instruction de la ligne 2) et une variable globale (instruction de la ligne 4). Quand la fonction *maville()* est lancée, la variable globale *code* est accessible. Par contre, pour la variable *ville*, c'est la valeur localement attribuée qui est affichée. La dernière instruction, nous montre qu'à l'extérieur de la fonction, la variable globale *ville* conserve sa valeur.

Remarque : s'il n'est pas possible de réaffecter une variable globale dans une fonction, on peut modifier son contenu grâce à ses méthodes ou en accédant à ses éléments, lorsque l'objet est mutable.

Listing 2.49 – Modification locale d'une variable globale

```
>>> def noise():
...     i = random.randint(0, len(tab)-1)
...     j = random.randint(0, len(tab)-1)
...     tab[i], tab[j] = tab[j], tab[i]
>>> tab = list(range(10))
>>> print(tab)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> noise()
>>> print(tab)
[0, 1, 8, 3, 4, 5, 6, 7, 2, 9]
```

Forcer le caractère global d'une variable

Si on veut modifier une variable globale dans une fonction, il faut utiliser le mot-clé *global*. Cela interdit la création d'une variable locale la masquant en cas d'affectation, et permet sa modification dans la fonction. Ce principe est illustré dans le listing 2.50.

Listing 2.50 – Modification d'une variable globale grâce à `global`

```
>>> def maville():
        global code, ville
        ville = 'Brest'
        code ='29200'
        print (ville, code)
>>> ville, code = 'Quimper', '29000'
>>> print(ville, code)
Quimper 29000
>>> maville()
Brest 29200
>>> print(ville, code)
Brest 29200
```

Définition 2.2 (Effet de bord)**.** *On appelle effet de bord la modification d'une variable globale par une fonction. Étymologie de effet de bord : traduction mot à mot de l'expression anglaise « side effect » qui signifie en français « effet secondaire ».*

Remarque : le comportement d'un programme peut changer suivant l'ordre dans lequel sont appelées des fonctions ayant des effets de bord. Il est donc conseillé de les limiter le plus possible. Voir la partie C.1 pour des exemples de problèmes liés à des effets de bord.

Dans les cas où malgré tout le programme contient de tels effets de bord, il est fortement recommandé de bien les documenter.

Variable non locale

Dans le cas de fonctions imbriquées, il est possible d'augmenter la portée d'une variable sans la déclarer comme étant globale. Il faut pour cela utiliser le mot-clé *nonlocal*.

Listing 2.51 – Augmentation de la portée d'une variable grâce à `nonlocal`

```
def affiche():
    ville, code = 'Quimper', '29000'
    def maville():
        nonlocal ville, code
        ville = 'Brest'
        code = '29200'
    print(ville, code)
    maville()
    print(ville, code)

affiche()
```

Ainsi, l'exécution du listing 2.51 affichera le résultat présenté dans le listing 2.52.

Listing 2.52 – Résultat de l'exécution du programme 2.51

```
Quimper 29000
Brest 29200
```

2.6.5 Arguments par défaut d'une fonction

La définition d'une fonction peut être faite en considérant un grand nombre de paramètres. À l'appel de cette fonction, donner explicitement une valeur à chacun des paramètres peut s'avérer laborieux et dégrader la lisibilité d'un programme. En Python, il est possible de donner des valeurs par défaut aux paramètres formels d'une fonction.

Exemple 2.8 (function hello first name multilingual)**.** *Comme dans l'exemple 2.6, nous voulons créer une fonction qui affiche un* Hello *à quelqu'un, mais cette fois-ci nous voulons que cette fonction soit potentiellement multilingue et que l'on considère l'anglais comme langue par défaut.*

Listing 2.53 – Fonction avec un paramètre par défaut

```
def hello(name, lang='English'):
    if lang == 'English':
        print('Hello ', name)
    elif lang == 'French':
        print('Bonjour ', name)
```

À l'exécution, nous avons:

```
>>> hello('Claire','English')
Hello  Claire
>>> hello('Claire','French')
Bonjour  Claire
```

```
>>> hello('Claire')
Hello  Claire
>>>
```

Dans le troisième cas, on ne précise pas la langue d'affichage. La fonction *hello()* utilise alors la valeur par défaut donnée lors de la définition. Dans notre cas la langue par défaut est *English*.

2.6.6 Nature du passage de paramètres pour une fonction

Il est important de préciser qu'en Python le passage de paramètres n'est en réalité qu'un passage de valeurs. Ce point est essentiel dès lors que l'on envisage de modifier ces valeurs en utilisant une fonction.

Exemple 2.9 (function modif)**.** *Dans cet exemple, nous définissons une fonction qui modifie la valeur d'un argument et nous cherchons à l'appliquer, ce qui peut donner le programme suivant:*

Listing 2.54 – Fonction modification

```
def modif(a):
    a=0

b=1
print('la variable b vaut : ',b)
modif(b)
print('la variable b vaut : ',b)
```

À l'exécution, nous avons :

```
$ python function_modif.py
la variable b vaut :  1
la variable b vaut :  1
$
```

Apparemment, la fonction *modif()* n'a eu aucun effet sur la variable *b* qui se trouvait dans le corps principal du programme. En fait, à l'appel de la fonction *modif(b)*, nous communiquons la valeur contenue dans *b* à ce moment là. Cette valeur est "copiée" dans le paramètre formel *a* qui jouera le rôle d'une variable locale et dont la portée ne dépassera pas le corps de la fonction. Tout changement sur *a* restera sans effet sur la variable *b*.

Remarque : Quand on parle de passage par valeurs, encore faut-il savoir la nature exacte de la valeur qui est communiquée à la fonction.

Exemple 2.10 (function modif liste)**.** *Cet exemple ressemble beaucoup au précédent à l'exception du fait que la fonction est maintenant prévue pour traiter des listes. On prendra dans notre cas une fonction swap() qui permute les deux premiers éléments d'une liste. Le code sera donc:*

Listing 2.55 – Fonction modification d'une liste

```
def swap(a):
    v=a[0]
    a[0]=a[1]
    a[1]=v

b=[1,5]
print('la variable b vaut : ',b)
swap(b)
print('la variable b vaut : ',b)
```

L'exécution donne :

```
$ python function_modif_liste.py
la variable b vaut :  [1, 5]
la variable b vaut :  [5, 1]
$
```

Nous constatons qu'il y a maintenant une modification de la variable *b* appartenant au corps principal du programme.

Cela tient à la nature de ce qu'est la *valeur* d'une liste : le lieu de la mémoire où est stocké son contenu. On ne peut donc pas rediriger cette référence vers une autre zone mémoire, mais on peut agir sur son contenu. C'est aussi le cas pour les dictionnaires et les tuples, mais, ces derniers étant non-mutables, cela n'a guère de conséquence. Pour les types de base, numériques (section 2.1.3 ou non-numériques comme *str* ou *bool*), c'est bien la valeur au sens le plus intuitif qui est passée, et la variable citée n'est effectiviement pas modifiable.

Ainsi, quand la fonction *swap()* manipule la liste en interne, elle manipule les valeurs contenues à cette adresse, ce qui correspond aux valeurs contenues par *b*. Il y a donc bien une modification de la variable *b*.

2.7 Bibliothèques

De base, le langage Python ne dispose que d'un nombre limité de fonctions. En revanche, Python bénéficie d'un nombre considérable de fonctions préalablement définies dans des bibliothèques [24] préprogrammées (voir le chapitre 8 pour une sélection des plus courantes). Nous avons déjà vu dans les exemples de script précédents que l'appel d'une bibliothèque était effectuée par la commande *import*.

Ainsi, pour utiliser une fonction trigonométrique, il est possible de faire appel à la bibliothèque *numpy* et écrire le script suivant :

Listing 2.56 – Appel à *numpy*

```
import numpy

angle = 3.1415/2
print(numpy.sin(angle))
print(numpy.cos(angle))
```

Le nom d'une bibliothèque peut être long, et répéter, à chaque appel de fonction, le nom complet de la bibliothèque, un point et le nom de la fonction (dans notre cas: *numpy.sin()*, *numpy.cos()*, *numpy.log()*,...) peut s'avérer fastidieux. Il possible au moment de l'appel de la bibliothèque d'associer un nom plus court avec la structure *import ... as ...* :

Listing 2.57 – Appel à *numpy* simplifié

```
import numpy as np

angle = 3.1415/2
print(np.sin(angle))
print(np.cos(angle))
```

Cette façon de faire est en général recommandée.

Une autre façon de procéder est d'importer de la bibliothèque une ou plusieurs fonctions spécifiques. Ceci donnera dans ce cas :

24. En Python, il est courant de nommer ces bibliothèques des *modules*.

Listing 2.58 – Appel des fonction *sin()* et *cos()* de *numpy*

```
from numpy import sin, cos

angle = 3.1415/2
print(sin(angle))
print(cos(angle))
```

Il est enfin possible d'importer directement toutes les fonctions d'une bibliothèque avec le symbole * :

Listing 2.59 – Appel de toutes les fonctions de numpy

```
from numpy import *

angle = 3.1415/2
print(sin(angle))
print(cos(angle))
```

Cette méthode est apparemment plus simple puisqu'au début du programme, il n'est plus besoin d'identifier les fonctions que l'on utilisera par la suite. Néanmoins, cette façon de faire est à déconseiller pour deux raisons. La première est que l'on importe un nombre considérable de fonctions inutiles. La seconde raison est que cela peut engendrer de graves problèmes de confusion si on importe plusieurs bibliothèques qui possèdent des fonctions différentes mais portant des noms identiques. Par exemple, la bibliothèque *numpy* et la bibliothèque *math* possèdent toutes les deux des fonctions trigonométriques, mais celles de *numpy* peuvent travailler avec des listes de valeurs numériques alors que celles de *math* n'acceptent que des valeurs numériques simples.

Dans le programme suivant :

Listing 2.60 – Appel de toutes les fonctions math numpy

```
from math import *
from numpy import *

angles=[3.1415, 3.1415/2]
print(sin(angles))
print(cos(angles))
```

la bibliothèque *numpy* est appelée après *math*. Les fonctions trigonométriques seront celles de *numpy* et l'exécution ne posera aucun problème :

```
[  9.26535897e-05   9.99999999e-01]
[ -9.99999996e-01   4.63267949e-05]
```

Par contre, si nous écrivons :

Listing 2.61 – Appel de toutes les fonctions de *numpy* et de *math*

```
from numpy import *
from math import *

angles=[3.1415, 3.1415/2]
print(sin(angles))
print(cos(angles))
```

les fonctions trigonométriques seront celles de *math* et l'exécution générera une erreur :

```
TypeError: a float is required
```

Remarque : dans les exemples précédents, on pratique 3.1415 comme π. Hormis le fait que cette approximation est de qualité bien inférieure à 3.1416, elle omet surtout que cette constante est déjà définie dans les bibliothèques concernées, avec autant de chiffres significatifs que possible pour un *float* : `numpy.pi` et `math.pi`.

2.7.1 Espaces de nommage

Pour éviter que les noms issus de différents modules ne se masquent les uns les autres, chaque module dispose de son propre *espace de nommage*, qui est un dictionnaire associant chacun de ses identificateurs à sa valeur. Cette valeur peut être une variable ou une constante, mais aussi une fonction ou un type.

Le nom de l'espace de nommage par défaut est `__main__`, qui est attribué au programme que l'on exécute. Losqu'on importe un module, le nom du module sert à construire le nom de l'espace de nommage.

Listing 2.62 – Module nommé `outils.py`, fournissant une fonction `prod`

```
1  def prod(tab):
2      res = 1
3      for x in tab:
4          res *= x
5      return res
6
7  if __name__ == '__main__':
8      print('Ce test doit donner la valeur 42')
9      print(prod([2,3,7]))
10 else:
11     print('module', __name__, 'chargé')
```

Si on exécute directement le module, la procédure de test est déclenchée :

```
1  $ python3 outils.py
2  Ce test doit donner la valeur 42
3  42
4  $
```

Si on importe ce module depuis l'interpréteur, la condition de la ligne 7 n'est plus vérifiée :

```
1  >>> import outils
2  module outils chargé
3  >>> outils.prod([1,4,9])
4  36
```

Dans un code destiné à être réutilisé – c'est-à-dire dans tout code propre – il est recommandé de ne pas placer de code exécutable au niveau d'indentation le plus bas, qui serait exécuté quel que soit le contexte, mais d'écrire les parties exécutées par défaut dans un tel bloc "`if __name__ == '__main__':`".

Remarque : changer le nom du module avec `import ... as ...` change la façon d'appeler les fonctions du module, pas le nom qu'il se donne :

```
1  >>> import outils as o
2  module outils chargé
3  >>> o.prod([1,4,9])
4  36
```

Cela a comme conséquence que pour chercher des sous-bibliothèques, il faut utiliser le nom intrinsèque, et pas le nom attribué lors de `import` :

```
>>> import scipy as sp
>>> import scipy.stats as st
```

```
>>> import scipy as sp
>>> import sp.stats as st
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'sp'
```

Tout ce qui doit arriver arrivera.
Tout ce qui, en arrivant, entraîne l'arrivée d'autre chose, provoquera l'arrivée d'autre chose.
Tout ce qui, en arrivant, entraîne que ça arrive de nouveau, arrivera de nouveau.
Enfin, pas nécessairement dans l'ordre chronologique.

Douglas Adams

3 Algorithmique 1

Sommaire

Ce chapitre introduit quelques notions d'algorithmique fondamentales. Ces notions, assez générales, ne sont pas limitées à un langage de programmation particulier. Cependant, quelques-unes seront illustrées en Python.

3.1 Algorithmes de tris

Définition 3.1 (Algorithme)**.** *Un algorithme est une méthode de résolution de problème énoncée sous la forme d'une série d'opérations à effectuer.*

Un algorithme peut également être défini comme étant une suite d'opérations transformant un ensemble de données appelé entrée en un nouvel ensemble de données appelé sortie.

Le terme algorithme a été introduit par *Ada Lovelace*[1] en hommage au mathématicien perse *al-Khawārizmī* (780-850).

Nous nous intéressons dans cette partie aux algorithmes de tris.

Définition 3.2 (Problème de tri)**.** *Soit une suite de n nombres : $a_1, a_2, ..., a_n$. Trier cette suite de nombres revient à trouver une permutation $\sigma_1, ..., \sigma_n$ de $1, ..., n$ telle que $a_{\sigma_1} \leqslant a_{\sigma_2} \leqslant ... \leqslant a_{\sigma_n}$.*

Le tri est un algorithme de base nécessaire pour bien d'autres algorithmes. Il permet de plus d'introduire la plupart des notions d'algorithmique. Il existe de nombreux algorithmes de tris, plus ou moins simples à appréhender et plus ou moins efficaces. La plupart des ouvrages d'algorithmique, comme par exemple [Cor+94] décrivent les algorithmes de tris les plus classiques. Pour une analyse relativement détaillée des méthodes de tris les plus classiques, consulter [Knu73].

Dans tous les exemples suivants, l'algorithme de tri sera appliqué à un tableau ou à une liste d'entiers, mais il est possible de trier tout ensemble de données muni d'une relation d'ordre total (il est par exemple possible de trier un ensemble de mots suivant l'ordre lexicographique).

3.1.1 Tri par sélection

Le tri par sélection est un tri extrêmement intuitif mais particulièrement inefficace. L'idée de base de l'algorithme est la suivante : rechercher le plus petit élément du tableau et l'échanger avec l'élément en première position. Recommencer avec le deuxième plus petit élément, et continuer jusqu'au $n-1^{\text{e}}$ plus petit élément.

Exemple :

7*	2	1*	8	4
1	2**	7	8	4
1	2	7*	8	4*
1	2	4	8*	7*
1	2	4	7	8

Les éléments à permuter sont marqués d'un « * ». Remarquer que lors de la seconde étape l'élément 2 est permuté avec lui-même.

Algorithme 3.1 : Algorithme du tri par sélection

```
Entrées : tab : tableau d'entiers
Données : indMin, i, j, n : entier
n ← taille(tab);
pour i ∈ [0, n − 2] faire
    indMin ← i;
    pour j ∈ [i + 1, n − 1] faire
        si tab[j] <tab[indMin] alors
            indMin ← j ;
        fin
    fin
    inverser(tab[indMin],tab[i]);
fin
```

Exemple 3.1 (Tri par sélection)**.** *L'algorithme du tri par sélection est décrit par l'algorithme 3.1. Il se traduit en Python par le programme 3.1.*

1. Considérée comme la première algorithmicienne

Listing 3.1 – Tri par sélection

```
def trisel(tab):
    """Tri du tableau tab par sélection.

    Parameters
    ----------
    tab : numpy.array
        tableau à trier.
    """

    # Parcours du tableau
    for i in range(len(tab)-1):
        # Recherche de la position du plus petit élément
        i_min = i
        for j in range(i+1, len(tab)):
            if tab[j] < tab[i_min]:
                i_min = j
        # Permuter les cases i et i_min
        tab[i], tab[i_min] = tab[i_min], tab[i]
```

Exemple 3.2 (Tri par sélection, version *numpy*)**.** *Version optimisée du tri par sélection utilisant les fonctions du module* numpy*.*

Listing 3.2 – Tri par sélection optimisé pour numpy

```
def trisel(tab):
    """Tri du tableau tab par sélection.

    Parameters
    ----------
    tab : numpy.array
        tableau à trier.
    """

    # Parcours du tableau
    for i in range(len(tab)-1):
        # Recherche de la position du plus petit élément
        i_min = np.argmin(tab[i:]) + i
        # Permuter les cases i et i_min
        tab[i], tab[i_min] = tab[i_min], tab[i]
```

Étude du tri par sélection

Lors de l'étude d'un algorithme, il est intéressant de quantifier le nombre d'opérations élémentaires nécessaires pour résoudre le problème. Ce nombre d'opérations s'exprime en fonction de la taille des données en entrée de l'algorithme. Ce nombre d'opérations s'appelle *complexité* de l'algorithme. La notion de complexité est définie dans l'annexe D.

Cette complexité peut s'exprimer dans le pire des cas ou en moyenne, ce dernier cas étant plus difficile à étudier. Dans le cas du tri par sélection, remarquons que tous les cas sont identiques. Rechercher le plus petit élément force à parcourir le tableau et coûte donc $n-1$ comparaisons. La recherche du deuxième plus petit élément coûte $n-2$ comparaisons. Le coût total du tri par sélection est donc de $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$. Il est d'usage de ne garder que le terme prépondérant de la complexité et de ne pas prendre en compte les constantes multiplicatives. On dit alors que la complexité en moyenne et dans le pire des cas du tri par sélection est de $\Theta(n^2)$.

Remarque : la complexité de la plupart des algorithmes de tris intuitifs est de $\Theta(n^2)$. La meilleure complexité possible pour un tri est de $\Theta(n \log n)$ (voir la section D.6 pour une démonstration de cette proposition).

3.1.2 Tri bulles

Le tri bulles est un tri très populaire, très facile à mettre en œuvre. Son fonctionnement s'appuie sur des permutations répétées d'éléments contigus qui ne sont pas dans le bon ordre.

Algorithme 3.2 : Algorithme du tri bulles

```
Entrées : tab : tableau d'entiers
Données : i, j, n : entier
n ← taille(tab);
pour i ∈ [n − 1, 0] faire
    pour j ∈ [0, i − 1] faire
        si tab[j] >tab[j + 1] alors
            inverser(tab[j],tab[j+1]);
        fin
    fin
fin
```

L'idée de l'algorithme est de faire remonter le plus grand élément comme une bulle, puis de faire remonter l'élément suivant et ainsi de suite. Ainsi, lors de la première itération, chaque élément du tableau est comparé à son successeur, et permuté si nécessaire. À la fin de cette itération, le plus grand élément a été placé en fin de tableau. On recommence alors l'itération sur les $n-1$ premières cases, puis sur les $n-2$ premières, *etc.* Cet algorithme est décrit figure 3.2.

Étude du tri bulles

L'étude de la complexité du tri bulles est très simple, et similaire à celle du tri par sélection. Encore une fois, tous les cas sont équivalents, et les complexités en moyenne et dans le pire des cas valent $\Theta(n^2)$.

3.1.3 Tri par insertion

Le tri par insertion est une méthode de tri efficace dans le cas où le nombre d'éléments à trier est relativement faible ou dans le cas où la liste est pratiquement ordonnée. Le principe de l'algorithme (décrit dans la figure 3.3) est le suivant : la liste à trier est découpée en deux parties. La partie de gauche est triée, et la partie de droite non triée. On insère alors successivement les éléments de la partie non triée à leur place dans la partie triée. Initialement, le premier élément est seul dans la partie triée. L'élément à insérer dans la partie triée est nommé *clé*.

Afin d'insérer la clé à sa place dans la partie triée, il faut décaler tous les éléments qui lui sont supérieurs d'un cran vers la droite.

Remarque : bien noter qu'il s'agit ici d'un décalage d'éléments et pas d'une permutation.

Exemple d'insertion de clé :

1	3	4	2

Dans cet exemple, la clé vaut 2. Lors de son insertion, on passe par les étapes suivantes :

Algorithme 3.3 : Algorithme du tri par insertion

```
Entrées : tab : tableau d'entiers
Données : i, j, n, cle : entier
n ← taille(tab) ;
pour i ∈ [1, n − 1] faire
    cle ← tab[i]                          # valeur à insérer dans la partie triée
    j ← i;
    tant que j > 0 et tab[j−1] >cle faire
        tab[j] ← tab[j−1]                 # décaler l'élément j d'un cran à droite
        décrémenter j;
    fin
    tab[j] ← cle                                        # insérer cle à sa place
fin
```

1	3	4	2
1	3	4	4
1	3	3	4
1	2	3	4

L'élément 4 est supérieur à la clé, et il a donc été copié d'un cran à droite (à ce moment, il apparaît donc deux fois). Ensuite, l'élément 3 a été copié un cran à droite. Enfin, la clé a été insérée à sa place.

En utilisant la méthode d'insertion vu précédemment, voici un exemple d'étapes du tri par insertion. Considérons le tableau ci-dessous :

5	2	6	3	1

Dans ce tableau, 5 fait partie de la partie triée. La première étape consiste à insérer la clé 2 dans la partie triée. On obtient alors le tableau suivant, où les deux premières valeurs sont triées :

2	5	6	3	1

On poursuit alors les itérations avec comme clés successives 6, 3 et 1 :

2	5	6	3	1
2	3	5	6	1
1	2	3	5	6

Étude du tri par insertion

Le tri par insertion est plus rapide que les tris précédents, bien que sa complexité moyenne soit du même ordre.

Cependant, cette fois, tous les cas ne sont pas équivalents. Le pire des cas correspond à un tableau trié en ordre inverse. Dans ce cas, « insérer la clé à sa place » coûte $\Theta(i)$ avec i la taille du tableau trié à explorer. La complexité de l'algorithme est alors $\sum_{i=1}^{n-1} i = \Theta(n^2)$.

Le meilleur des cas est un tableau trié. Dans ce cas, l'insertion de la clé coûte 1, et la complexité de l'algorithme devient $\Theta(n)$.

Nous admettrons que sa complexité moyenne est de $\Theta(n^2)$ ($\frac{n^2}{4}$ comparaisons).

3.1.4 Tri shell

Ce tri, proposé en 1959 par Donald L. Shell, constitue une variante optimisée du tri par insertion.

Remarquons que le principal inconvénient du tri par insertion est le temps nécessaire pour déplacer une clé d'un côté à l'autre du tableau : il faut n déplacements d'une case. Cette remarque est à l'origine de l'algorithme du tri shell.

Avant de trier le tableau, le tri shell va réaliser un pré-tri, c'est-à-dire qu'il va placer approximativement les éléments à leur place. Ce pré-tri va être réalisé sous la forme de tris par insertion avec différents pas h c'est-à-dire que chaque élément est décalé de h cases vers la droite (remarquons que pour $h = 1$, il s'agit d'un vrai tri par insertion). Un premier pré-tri avec un pas h grand permet de placer grossièrement les éléments. Ensuite, le pré-tri est affiné en diminuant la valeur de h. Finalement, un tri avec $h = 1$ (donc un tri par insertion) est effectué. Remarquons toutefois que le tri par insertion est effectué sur un tableau *pratiquement trié* et qu'il sera donc très efficace.

Le choix de la suite de h est important et conditionne la rapidité du tri.

Exemple de choix :
Considérons la suite $u_0 = 1; u_{n+1} = 3u_n + 1$. Soit N la taille du tableau à trier. La suite de pas sera définie par :

— $h_0 =$ le plus grand u_n tel que $u_n < N$
— $h_{n+1} = \dfrac{h_n - 1}{3}$

Algorithme 3.4 : Algorithme du tri Shell

```
Entrées : tab : tableau d'entiers
Données : i, j, n, cle, h : entiers
n ← taille(tab);
h ← 1;
tant que 3h + 1 < n faire                                   # calcul de h_0
    h ← 3h+1;
fin
tant que h > 0 faire
    pour i ∈ [h, n − 1] faire
        cle ← tab[i]                          # valeur à insérer dans la partie triée
        j ← i;
        tant que j ⩾ h et tab[j − h] >cle faire
            tab[j] ← tab[j-h]                 # décaler l'élément j de h crans à droite
            j ← j-h;
        fin
        tab[j] ← cle                                        # insérer cle à sa place
    fin
    h ← h/3                                                 # équivalent à (h-1)/3
fin
```

Étude du tri shell

Le tri shell est plus complexe à étudier que les tris vus jusqu'ici. Aussi, nous nous contenterons de donner les complexités sans les calculer. Pour une étude complète du tri shell, consulter [Knu73].

La complexité dans le pire des cas du tri shell dépend de la suite h_i choisie. La meilleure suite est inconnue, mais il existe des suites [2] pour lesquelles la complexité est de $\Theta(n^{\frac{3}{2}})$ et d'autres [3] pour lesquelles elle est de $\Theta(n^{\frac{4}{3}})$.

La complexité en moyenne pour la meilleure suite de h est elle aussi inconnue (car la meilleure suite de h est inconnue). Pour $h_i = 2^{i+1} - 1$, elle est de $O(n^{\frac{3}{2}})$. Pour $h_i = 2^p 3^q$, elle est de $O(n \log^2 n)$.

3.1.5 Autres tris

Les tris présentés jusqu'ici sont rarement utilisés en raison de leur manque d'efficacité. Quelques tris plus performants sont présentés brièvement ici, mais ils ne sont pas détaillés car ils utilisent des notions d'algorithmique vues ultérieurement.

Tri par segmentation, ou quicksort

Le tri par segmentation est un des tris les plus performants connus pour les tableaux de grande dimension. Couplé au tri par insertion, il devient le *quicksort* qui, comme son nom l'indique, est un tri particulièrement rapide et qui possède l'avantage [4] d'avoir une complexité en espace de $\Theta(1)$ (*i.e.* indépendant de la taille des données).

L'idée du tri par segmentation est la suivante : étant donné un tableau, on choisit un élément pivot, puis on place tous les éléments inférieurs au pivot à gauche et tous les éléments supérieurs à droite. On recommence ensuite l'opération sur les deux sous tableaux délimités par l'élément pivot.

Le *quicksort* utilise de plus la contrainte suivante : si la taille du sous-tableau est inférieure à un seuil n, on le trie en utilisant un tri par insertion. Les implantations les plus simples de cet algorithme utilisent la notion de récursivité vue en section 3.2.

Dans le pire des cas, le *quicksort* a une complexité de $\Theta(n^2)$. Par contre, sa complexité moyenne est de $\Theta(n \log n)$ qui est la meilleure complexité atteignable par un algorithme de tri. De plus, les constantes multiplicatives (qui n'apparaissent pas dans l'expression de la complexité) sont très faibles et font de ce tri l'un des plus performants connus.

Tri par partition/fusion

Ce tri s'appuie sur la notion de liste vue en section 6.1. La particularité de ce tri est que sa complexité dans le pire des cas est de $\Theta(n \log n)$. Par contre, son temps d'exécution est en général moins bon que celui du *quicksort*. De plus, il utilise plus de mémoire que le *quicksort* (sa complexité en espace est de $\Theta(n)$).

L'idée de l'algorithme est la suivante : les éléments à trier sont représentés sous la forme d'une liste de listes de un élément. Puis, les listes sont fusionnées entre elles deux à deux, de manière triée. On obtient alors une liste de listes de deux éléments. On poursuit l'opération de fusion jusqu'à obtenir une liste contenant une seule liste de n éléments. La performance de cet algorithme vient du fait qu'il est très rapide de fusionner deux listes triées entre elles.

Il est également possible d'écrire cet algorithme en utilisant des tableaux, et on obtient alors un tri très efficace en temps mais qui ne s'exécute pas *en place* (i.e. il utilise un espace mémoire supplémentaire dépendant de la taille des données).

2. $h_i = 2^{i+1} - 1$
3. $h_i = \begin{cases} 9.2^i - 9.2^{\frac{i}{2}} + 1 \text{ si i est pair} \\ 8.2^i - 6.2^{\frac{i+1}{2}} + 1 \text{ si i est impair} \end{cases}$
4. On néglige l'espace requis pour les appels récursifs, cf. section 3.2.2.

Tri par arbre binaire de recherche équilibré

Ce tri s'appuie sur la notion d'arbre vue en section 6.2. La complexité dans le pire des cas de cet algorithme est de $\Theta(n \log n)$, mais il est moins rapide que le tri par partition/fusion.

L'idée de l'algorithme est la suivante : les éléments à trier sont insérés un à un dans un arbre binaire de recherche. Ensuite, un simple parcours infixe de l'arbre fournit les éléments triés. Si l'arbre est maintenu équilibré lors de sa construction (*cf.* section 6.2.7), alors l'insertion d'un élément se fait en $\Theta(\log n)$ et le tri se fait donc en $\Theta(n \log n)$.

3.1.6 Mélange d'un tableau

Il existe différentes méthodes pour mélanger aléatoirement un tableau de nombres. La méthode abordée ici est extraite de [Knu69]. Elle permet d'obtenir une permutation aléatoire d'un tableau de taille n en n étapes (en générant uniquement n nombres aléatoires).

Algorithme 3.5 : Algorithme de mélange d'un tableau

Entrées : tab : entier[]
Données : i, n : entiers
n ← taille(tab)-1 ;
tant que $n > 0$ **faire**
    i ← nombre aléatoire entre 0 et n inclus ;
    permuter tab[n] et tab[i] ;
    décrémenter n ;
fin

L'algorithme 3.5 s'écrit en Python de la manière suivante :

Listing 3.3 – Mélange d'un tableau

```
def shuffle(tab):
    n = len(tab) - 1
    while n > 0:
        # Générer un nombre aléatoire entre 0 et n inclus
        i = np.random.randint(0, n + 1)
        tab[i], tab[n] = tab[n], tab[i]
        n -= 1
```

Il est également possible d'utiliser la fonction *shuffle* de la bibliothèque *numpy* (listing 3.4) :

Listing 3.4 – Mélange d'un tableau avec shuffle

```
import numpy as np

# création d'un tableau de type numpy.array de taille nb
tab = np.array(range(nb))
# mélange du tableau grâce à la fonction shuffle de numpy
np.random.shuffle(tab)
```

3.2 Récursivité

Définition 3.3 (Fonction récursive)**.** *Une fonction ou une méthode est dite récursive si elle se définit à partir d'elle-même, c'est-à-dire si elle comporte un appel à elle-même dans son corps.*

Le récursivité permet une traduction immédiate des fonctions mathématiques définies par récurrence.

3.2.1 Premier exemple

Exemple 3.3 (Factorielle). *La fonction factorielle se définit par récurrence de la manière suivante :*

$$\begin{cases} 0! = 1 \\ n! = n(n-1)! \end{cases}$$

Ceci se traduit en python par le listing 3.5.

Listing 3.5 – Factorielle récursive

```
1 def fact(n):
2     """Fonction récursive de calcul de n!
3     """
4
5     # Condition d'arrêt
6     if n == 0:
7         return 1
8     else:
9         return n * fact(n-1)
```

On remarque que le corps de la fonction récursive est découpé en deux parties :
— une condition d'arrêt (ligne 6) ;
— un appel récursif (ligne 9).

Toute fonction récursive doit comporter au moins une condition d'arrêt : il s'agit d'une condition pour laquelle il n'y a pas d'appel récursif. Dans l'exemple 3.3, il s'agit de la valeur initiale de la suite factorielle. De manière générale, dans le cas où la fonction récursive traduit une suite définie par récurrence, la condition d'arrêt de la fonction correspond à la condition initiale de la suite.

Remarque : une fonction récursive peut comporter plusieurs conditions d'arrêt et plusieurs appels récursifs. Par exemple, la fonction *factorielle* aurait pu s'écrire :

```
1 def fact(n):
2     """Fonction récursive de calcul de n!
3     """
4
5     # Condition d'arrêt
6     if n == 0:
7         return 1
8     # Autre condition d'arrêt
9     elif n == 1:
10        return 1
11    else:
12        return n * fact(n-1)
```

3.2.2 Notion de pile d'appels

La notion de pile d'appels est indispensable pour comprendre le fonctionnement de la récursivité. L'idée fondamentale est qu'une fonction récursive empile les appels récursifs, les traite, les dépile, puis continue son traitement.

Par exemple lors du calcul de 2! par la méthode de l'exemple 3.3 :
— la méthode *fact* est appelée avec le paramètre 2 ;
— elle calcule $2 * fact(1) \Rightarrow$

- la méthode *fact* est appelée avec le paramètre 1 ;
- elle calcule $1 * fact(0) \Rightarrow$
  - la méthode *fact* est appelée avec le paramètre 0 ;
  - elle retourne 1
- elle retourne $1 * 1 = 1$

— elle retourne $1 * 2 = 2$

Il y a donc à un certain moment les appels de *fact(2)*, *fact(1)* et *fact(0)* en attente dans la pile.

Remarque : la pile d'appels est une zone de la mémoire de taille finie. Lorsqu'elle contient trop d'appels, elle déborde et génère une exception : *python.lang.StackOverflowError* (*cf.* section 8.2.5 pour modifier sa taille).

3.2.3 Quelques exemples

Nous présentons ici quelques problèmes pouvant être traités par des algorithmes récursifs.

Fibonacci

La suite de Fibonacci est un exemple classique de suite pouvant être calculée de manière récursive, pour laquelle une traduction immédiate de la suite mathématique fournit un programme particulièrement inefficace.

Exemple 3.4 (Fibonacci)**.** *La suite de Fibonacci se définit ainsi :*

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ \forall n \geqslant 2, F_n = F_{n-1} + F_{n-2} \end{cases}$$

Elle se traduit en Python de la manière suivante :

Listing 3.6 – Suite de Fibonacci récursive

```
1  def fibo(n):
2      """Fonction récursive de calcul de la suite de Fibonacci.
3      """
4
5      if n == 0:
6          return 0
7      elif n == 1:
8          return 1
9      else:
10         return fibo(n-1) + fibo(n-2)
```

Examinons la pile d'appel dans le cas du calcul de *fibo(3)* :

- appel à *fibo(3)* $\Rightarrow$
  - appelle *fibo(2)* $\Rightarrow$
    - appelle *fibo(1)*
      - retourne 1
    - *+ fibo(0)*
      - retourne 0
    - retourne 1
  - *+ fibo(1)*
    - retourne 1
- retourne 2

On constate que le nombre d'appels à la méthode *fibo* augmente de manière exponentielle par rapport à n : chaque calcul de fibo fait deux appels récursifs. De plus, certaines valeurs (comme par exemple F_1) sont calculées plusieurs fois.

Donc, l'implantation récursive de la suite de fibonacci est très simple à mettre en œuvre, très proche de la formule mathématique, mais très inefficace. Elle ne sera jamais utilisée. Un exemple d'implantation récursive correcte sera abordé en TD.

Recherche d'un élément dans un tableau trié

Considérons un tableau trié de n entiers. Nous voulons rechercher si un entier est présent dans le tableau grâce à une fonction récursive. Afin d'être efficace, nous allons procéder par dichotomie. L'idée de l'algorithme est la suivante :

- tester l'élément au milieu du tableau ;
  - s'il a la valeur recherchée, l'algorithme se termine ;
  - si la partie du tableau à traiter n'a plus qu'un élément, alors la valeur recherchée n'est pas dans le tableau. Fin de l'algorithme ;
  - si l'élément est supérieur à l'élément recherché, poursuivre la recherche dans la partie gauche du tableau ;
  - sinon poursuivre la recherche dans la partie droite du tableau.

Exemple 3.5 (Recherche récursive)**.** *La méthode de recherche récursive d'un élément dans un tableau sera une méthode retournant un booléen*[5] *et possédant quatre paramètres : le tableau, l'élément à chercher, et les bornes entre lesquelles chercher.*

On commence par les conditions d'arrêt : est-ce que l'élément est trouvé, ou est-ce qu'il n'est pas dans le tableau (zone de recherche limitée à une case).

Sinon, il faut faire les appels récursifs : si l'élément est supérieur à la valeur recherchée, alors le résultat de la recherche est égal au résultat de la recherche dans la première partie du tableau. Sinon, il est égal au résultat de la recherche dans la seconde partie du tableau.

Remarquer la méthode recherche *qui amorce la méthode récursive sur tout le tableau, évitant à l'utilisateur de renseigner les quatre paramètres de* recherche_rec*.*

Listing 3.7 – Recherche récursive dans un tableau

```
1  def recherche_rec(tab, val, deb, fin):
2      """Recherche récursive d'un élément dans un tableau.
3
4      Parameters
5      ----------
6      tab : numpy.array
7          Tableau contenant les éléments
8      val : int
9          Valeur à chercher
10     deb, fin : int, int
11         Intervalle dans lequel rechercher
12
13     Returns
14     -------
15     bool
16         True si trouvé, False sinon.
17     """
18     # Milieu de la zone de recherche
19     m = (deb + fin) // 2
20     # Élément trouvé
21     if tab[m] == val:
```

5. indiquant si l'élément est trouvé ou non

```
22          return True
23      # deb >= fin  ->  élément non trouvé
24      elif deb >= fin:
25          return False
26      # Recherche dans la partie gauche
27      elif tab[m] > val:
28          return recherche_rec(tab, val, deb, m - 1)
29      # Recherche dans la partie droite
30      else:
31          return recherche_rec(tab, val, m + 1, fin)
32
33
34  def recherche(tab, val):
35      """Recherche d'une valeur dans un tableau trié.
36      Utilise la fonction recherche_rec.
37      """
38      return recherche_rec(tab, val, 0, len(tab)-1)
```

Tours de Hanoï

La figure 3.1 représente le jeu des tours de Hanoï. Le principe est très simple : plusieurs disques sont empilés par taille décroissante sur l'emplacement 1 et forment une tour. L'objectif est de déplacer cette tour vers l'emplacement 2 sachant qu'un disque ne peut être empilé que sur un disque plus grand et qu'il n'est possible que de déplacer un seul disque à la fois.

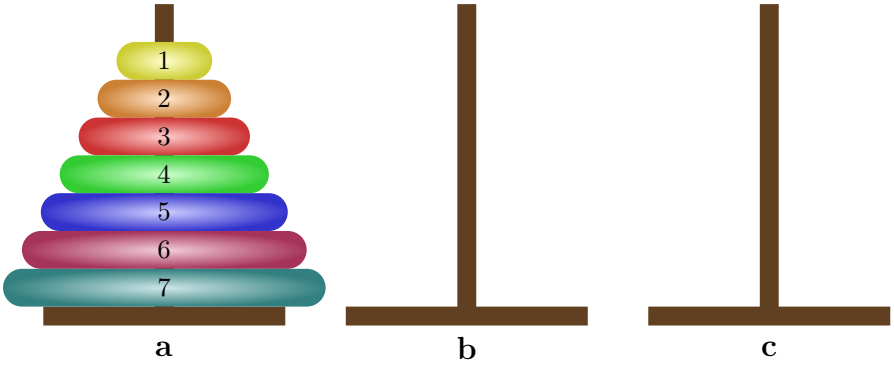


FIGURE 3.1 – Tours de Hanoï

Ce problème est tiré de la légende des tours de Bramah : dans le grand temple de Bénarès, sous le dôme qui marque le centre du monde, se trouve une plaque de bronze où sont fixées trois aiguilles de diamant, hautes chacune d'une coudée et fines comme la taille d'une guêpe. Sur une de ces aiguilles, lors de la création du monde, Dieu a placé 64 disques d'or pur, le plus large reposant sur la plaque de bronze, et les autres allant en décroissant jusqu'au plus petit. C'est la Tour de Bramah. Jour et nuit, sans arrêt, les prêtres transfèrent les disques d'une aiguille de diamant à une autre, en suivant les lois immuables de Bramah qui veulent que le prêtre de service ne prenne qu'un disque à la fois, et qu'il le place sur une aiguille de telle manière qu'il ne se trouve jamais sous lui de disque plus petit. Lorsque les soixante-quatre disques auront été transférés de l'aiguille sur laquelle Dieu les a mis lors de la création du monde, à une des autres aiguilles, la Tour, le Temple et les Brahmanes s'écrouleront en poussière, et dans un coup de tonnerre, le Monde s'évanouira.

Un algorithme permettant de résoudre ce problème s'écrit très simplement en utilisant la récursivité. L'algorithme récursif se construit de manière semblable à une démonstration par

Procédure hanoi(entier n, entier source, entier dest, entier tmp)

```
/* n : hauteur de la tour                                              */
/* source, dest, tmp : position d'origine, finale et intermédiaire de la
   tour à déplacer                                                     */
si n > 0 alors
    hanoi (n-1, source, tmp, dest) ;
    déplacer(source, dest) ;
    hanoi (n-1, tmp, dest, source) ;
fin
```

récurrence : la solution pour une tour de hauteur 1 est triviale. On suppose disposer d'un algorithme permettant de résoudre le problème pour une tour de hauteur h. Comment résoudre le problème pour une tour de hauteur $h+1$?

La réponse est la suivante : pour déplacer une tour de hauteur $h+1$ de a vers b, utiliser l'appel récursif pour déplacer les h premiers éléments de a vers c, déplacer le dernier élément de a vers b, puis déplacer les h premiers éléments de c vers b (nouvel appel récursif). Cet algorithme est décrit par la procédure hanoi. On peut remarquer qu'il n'y a pas d'action si $n \leqslant 0$: il s'agit de la condition d'arrêt. La traduction en python de cet algorithme est immédiate et laissée à titre d'exercice.

Étude de l'algorithme : quelle est la complexité de cet algorithme en fonction du nombre n de disques ?

Cette complexité se calcule facilement par récurrence :

— déplacer une tour de 1 disque se fait en 1 étape

— si on sait déplacer une tour de n disques en $c(n)$ étapes, alors on sait déplacer une tour de hauteur $n+1$ en $c(n+1) = 2c(n)+1$ étapes

— donc, $\forall n \in \mathbb{N}, c(n) = 2^n - 1$

La complexité de l'algorithme est donc de $\Theta(2^n)$ (complexité non polynomiale). Pour en revenir à la légende de la tour de Bramah, déplacer une tour de 64 disques nécessite $2^{64}-1$ déplacements élémentaires, c'est-à-dire 18 446 744 073 709 551 615. À raison d'un déplacement par seconde, il faudra 584,54 milliards d'années aux prêtres pour déplacer la tour...

Tri par segmentation

L'idée générale du tri par segmentation est décrite au paragraphe 3.1.5. L'algorithme est représenté par la fonction triSeg.

Cet algorithme se traduit, après quelques optimisations, par le listing 3.8.

Listing 3.8 – Quicksort récursif

```
1  def tri_seg(tab):
2      """Tri du tableau tab par segmentation.
3
4      Parameters
5      ----------
6      tab : tableau ou numpy.array
7          Valeurs à trier.
8      """
9
10     return tri_seg_rec(tab, 0, len(tab) - 1)
11
12
```

```python
def tri_seg_rec(tab, a, b):
    """Fonction récursive de tri par segmentation.

    Parameters
    ----------
    tab : tableau ou numpy.array
        Valeurs à trier.
    a, b : int, int
        Bornes entre lesquelles trier
    """
    cpt = 0  # compteur d'opérations
    x, y = a, b
    if x < y:
        # Clé de tri : tout ce qui est inférieur est placé à gauche
        # tout ce qui est supérieur à droite. C'est le choix du pivot.
        cle = tab[(x + y) // 2]
        while x <= y:
            # Se décaler tant que tab[x] est du bon côté de la clé
            while x < b and tab[x] < cle:
                x += 1
            # Se décaler tant que tab[y] est du bon côté de la clé
            while y > a and tab[y] > cle:
                y -= 1
            if x <= y:
                # Ici tab[x] et tab[y] sont du mauvais côté de la clé
                # Il faut donc les permuter
                tab[x], tab[y] = tab[y], tab[x]
                cpt += 2
                x += 1
                y -= 1
        # Appels récursifs sur les deux sous-tableaux
        if x < b:
            cpt += tri_seg_rec(tab, x, b)
        if a < y:
            cpt += tri_seg_rec(tab, a, y)
    return cpt
```

Procédure triSeg(tableau_entier tab, entier debut, entier fin)

si *debut*<*fin* **alors**
  si *fin* - *debut* $\leqslant 1$ **alors** # cas particulier : 2 éléments à trier
    si *tab*[*debut*] > *tab*[*fin*] **alors**
      inverser(tab, debut, fin) ;
    fin
  sinon
    choisir un élément pivot p entre debut et fin ;
    placer les éléments inférieurs au pivot avant et les éléments supérieurs après (le pivot est en *i*) ;
    `triSeg` (tab, deb, i-1) ;
    `triSeg` (tab, i+1, fin) ;
  fin
fin

3.2.4 Récursivité terminale

Définition 3.4 (Récursivité terminale)**.** *Une fonction est dite récursive terminale si elle ne contient aucun traitement après un appel récursif.*

L'exemple 3.3 p. 73 représente un calcul de factorielle de manière *non* récursive terminale : en effet, lors du *return*, un calcul est effectué (`n * fact(n-1)`).

Certains langages — dont Python ne fait malheureusement pas partie — permettent d'optimiser l'exécution des fonctions récursives terminales. Un appel à une fonction récursive terminale ne se traduit alors pas par des empilements dans la pile d'appel (voir 3.2.2 p. 73), et il n'y a alors plus de risque de débordement de pile.

Le passage d'une fonction récursive à une fonction récursive terminale se traduit généralement par l'ajout d'un paramètre traditionnellement appelé *accumulateur*. Ce paramètre sert à « accumuler » les résultats intermédiaires.

Exemple 3.6 (Factorielle récursive terminale)**.** *La fonction factorielle s'écrit de manière récursive terminale ainsi :*

Listing 3.9 – Factorielle récursive terminale

```
def fact_rec_term(n, acc):
    """Fonction factorielle récursive terminale.
    """
    # Condition d'arrêt : le résultat est dans l'accumulateur
    if n == 0:
        return acc
    # Appel récursif : modifier l'indice et l'accumulateur
    else:
        return fact_rec_term(n - 1, acc * n)


def fact(n):
    """Appel de la fonction récursive terminale.
    """

    # Initialiser l'accumulateur à l'élément neutre de la multiplication
    return fact_rec_term(n, 1)
```

Remarque 1 : le passage sous forme récursive terminale peut permettre de dépasser une traduction naïve d'une définition de suite mathématique par récurrence. On peut ainsi transformer un algorithme de complexité exponentielle en algorithme linéaire.

Remarque 2 : lorqu'il est possible d'écrire une fonction sous forme récursive terminale, il est aussi possible de l'écrire à l'aide d'une boucle, qui met à jour l'accumulateur à chaque itération, et incrémente ou décrémente l'indice.

3.2.5 Avantages, inconvénients de la récursivité

Remarque : tout algorithme récursif peut s'écrire de manière itérative, et réciproquement. Church, Turing et Kleene montrent l'équivalence de la calculabilité entre les fonctions récursives générales, le λ-calcul et les machines de Turing (1936). L'équivalence entre les machines de Turing et le pseudo-code utilisé en algorithmique est bien connue.

Cependant, certains algorithmes sont plus faciles à écrire de manière récursive, même si cette facilité de programmation se paye par un coût d'exécution plus important. Si la pénalité due à la

programmation récursive n'est pas trop importante et que le gain en facilité de programmation l'est, alors ne pas hésiter à choisir l'algorithme récursif.

En particulier dans le cas des parcours d'arbres (décrits paragraphe 6.2), il est conseillé d'utiliser des méthodes récursives (un parcours infixé d'un arbre s'écrit en 5 lignes de manière récursive et en plusieurs dizaines de lignes de manière itérative). De même, le tri par segmentation est beaucoup plus facile à écrire en récursif.

3.2.6 Mémoisation

Le terme anglais *memoization* (traduit en français par mémoisation) a été introduit par Donald Michie en 1968. Il vient du latin *memorandum* qui signifie « qui doit être rapplelé ». Il représente une technique d'optimisation de programmes.

Le principe de la mémoisation est d'utiliser un dictionnaire pour mémoriser les résultats intermédiaires d'une fonction (généralement une fonction récursive). Il s'agit donc de diminuer le temps de calcul au prix d'une utilisation de mémoire plus importante.

Exemple de mémoisation

La mémoisation peut s'appliquer à une fonction de plusieurs manières. Considérons comme cas d'étude la fonction factorielle récursive du listing 3.5.

Une première possibilité consiste à ajouter un dictionnaire en paramètre de la fonction ou en variable globale.

Listing 3.10 – Factorielle récursive avec mémoisation, première version

```
# Dictionnaire de mémoisation
fact_memo = dict()

def fact(n):
    global fact_memo
    # Test de mémoisation
    if n not in fact_memo:
        if n == 0:
            return 1
        else:
            fact_memo[n] = n * fact(n-1)
    return fact_memo[n]
```

Le listing 3.10 contient la fonction *factorielle* à laquelle le principe de mémoisation a été appliqué. L'inconvénient de cette version est qu'elle nous oblige à modifier la fonction. Pour résoudre ce problème, il est possible de créer un décorateur à appliquer aux fonctions à mémoiser, comme dans le listing 3.11. Dans ce listing, on remarque que la fonction *fact* est inchangée. Le décorateur est la fonction *memoize* qui s'applique en ajoutant `@memoize` avant la fonction *fact*.

Listing 3.11 – Factorielle récursive avec mémoisation, deuxième version

```
def memoize(f):
    """Fonction de décoration pour la mémoisation.
    Utiliser en ajoutant @memoize avant une fonction.
    """
    dict_memo = {}  # Dictionnaire de mémoisation
    def annexe(x):
        """Fonction de mémoization. Calcule f(x) en utilisant le
          dictionnaire.
        """
        if x not in dict_memo:
```

```
            dict_memo[x] = f(x)
        return dict_memo[x]
    return annexe

@memoize
def fact(n):
    """La fonction factorielle est inchangée. L'application du décorateur
    @memoize suffit à appliquer la mémoisation.
    """
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

3.3 Type abstrait de données

Un type abstrait de données (TAD) est une structure qui n'est pas définie en terme d'implantation en mémoire ou par la simple définition de ses composantes, mais plutôt en termes d'opérations et des propriétés d'application de ces opérations sur les données.

Ces propriétés se divisent en trois catégories :

— créateurs (création de la donnée) ;
— sélecteurs (sélection d'une partie de la donnée) ;
— transformateurs (modification de la donnée).

Un TAD permet de disposer d'une brique de base réutilisable dans d'autres contextes. Il est possible de modifier la représentation interne des données sans avoir à modifier le reste du programme. La programmation orientée objet nous dirige naturellement vers la définition de types abstraits de données (un TAD est une classe que l'on utilise en créant des objets).

3.3.1 Nombre rationnel

Considérons le type abstrait nombre rationnel. Il doit être possible de créer un rationnel, de multiplier deux rationnels, de tester si deux rationnels sont égaux, ...

Un nombre rationnel pourra, par exemple, être représenté par deux entiers (il s'agit de la représentation la plus logique mais pas la seule possible). Il est alors possible d'ajouter la contrainte suivante : le dénominateur est toujours strictement positif. Une représentation possible du type abstrait est :

Listing 3.12 – Type abstrait rationnel

```
class Rationnel(object):
    """Type abstrait nombre rationnel.
    """

    def __init__(self, num, den=1):
        """Creation d'un rationnel.
        Si le dénominateur est nul, alors le rationnel est non défini.
        On s'assure lors de la création que le dénominateur soit positif.
        """
        if den == 0:
            raise Exception('Invalid number')
        elif den < 0:
            self.__num = -num
            self.__den = -den
        else:
```

```
16              self.__num = num
17              self.__den = den
18
19      def __eq__(self, other):
20          """Test d'égalité entre deux rationnels.
21          """
22          return self.__num * other.__den == other.__num * self.__den
23
24      def __add__(self, other):
25          """Calcule la somme de deux rationnels.
26          Le résultat est un rationnel.
27          """
28          n = self.__num * other.__den + self.__den * other.__num
29          d = self.__den * other.__den
30          return Rationnel(n, d)
31
32      def __mul__(self, other):
33          """Calcule le produit de deux rationnels.
34          Le résultat est un rationnel.
35          """
36          n = self.__num * other.__num
37          d = self.__den * other.__den
38          return Rationnel(n, d)
39
40      def __str__(self):
41          """Représentation textuelle d'un rationnel.
42          Permet de les afficher simplement.
43          """
44          return '{0}/{1}'.format(self.__num, self.__den)
45
46  if __name__ == '__main__':
47      # Test : création de rationnels
48      r1 = Rationnel(1, -2)
49      r2 = Rationnel(3, 5)
50      # Affichage des rationnels et de leur produit
51      print(r1, r2, r1*r2)
52      # Test d'égalité de rationnels
53      print(r1 == Rationnel(-2, 4))
```

Remarques :

— dans le listing 3.12, la contrainte de positivité du dénominateur n'est garantie qu'à la création de l'objet. Pour la garantir durant toute la vie de l'objet[6], il faut utiliser des décorateurs, comme illustré dans le listing 5.5 p. 110.

— une exception est levée dans le cas où le dénominateur est nul ; le fonctionnement des exceptions est décrit section 7.2 p. 159.

3.3.2 Pile

Une pile (LIFO[7]) est un type abstrait contenant plusieurs éléments, dans lequel le premier élément à sortir est le dernier entré. La figure 3.2 représente une pile.

6. Les opérations présentées dans le listing ne peuvent certes pas casser cette propriété de dénominateur positif. L'ajout des opérateurs - (soustraction, `__sub__`) et/ou - (changement de signe, `__neg__`) y conduirait, ainsi que certaines implémentations de la division (`__floordiv__` : `//` ou `__truediv__` : `/`)

7. Last In First Out

Une pile possède une capacité de n éléments. Il est possible d'empiler ou de dépiler des éléments, de tester si la pile est pleine ou si elle est vide. La représentation interne utilisée sera un tableau de taille n pour les éléments et un entier pour la position du sommet de la pile (dernier élément empilé).

Le code Python représentant le type abstrait pile est défini dans le listing 3.13.

Listing 3.13 – Type abstrait pile

```
import numpy as np


class Pile():
    """Type abstrait pile d'entiers.
    La pile est créée avec une taille maximale autorisée.
    Ce type s'appuie sur le type numpy.array.
    """

    def __init__(self, taille=10):
        """Creation d'une pile : initialisation du tableau
        et de la position du sommet.
        """
        self.__contenu = np.zeros(taille, dtype=np.int32)
        self.__sommet = -1

    def est_vide(self):
        """Teste si la pile est vide.
        """
        return self.__sommet == -1

    def est_pleine(self):
        """Teste si la pile est pleine.
        """
        return self.__sommet == len(self.__contenu) - 1

    def empiler(self, val):
        """Ajoute un élément au sommet de la pile.
        La pile ne doit pas être pleine.
        """
        if not self.est_pleine():
            self.__sommet += 1
            self.__contenu[self.__sommet] = val

    def depiler(self):
        """Retire l'élément du sommet de la pile.
        Retourne None si la pile est vide.
        """
        if not self.est_vide():
            self.__sommet -= 1
            return self.__contenu[self.__sommet + 1]
        else:
            return None

    def __str__(self):
        """Représentation textuelle de la pile.
        """
        s = ''
        for i in range(self.__sommet, -1, -1):
```

```
50              s += '{0}\n'.format(self.__contenu[i])
51          return s
52
53  if __name__ == '__main__':
54      # Test de la classe Pile
55      p = Pile(3)
56      i = 0
57      while not p.est_pleine():
58          p.empiler(i)
59          i += 1
60          print(p)
61          while not p.est_vide():
62              x = p.depiler()
63              print('Dépile : ', x)
```

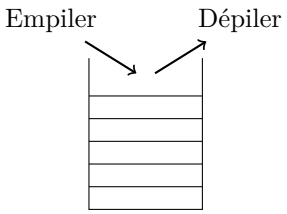


FIGURE 3.2 – Pile

3.3.3 File

Une file (FIFO [8]) est un type abstrait contenant plusieurs éléments dans lequel le premier élément à sortir est le premier entré. Son fonctionnement ressemble à celui d'une file d'attente : les premières personnes à arriver sont les premières personnes à sortir de la file.

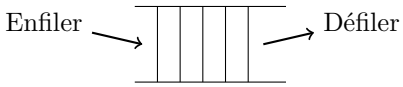


FIGURE 3.3 – File

Une file possède les fonctionnalités suivantes : il est possible d'enfiler ou de défiler des éléments, d'examiner sa tête de file (sans le défiler), de tester si la file est pleine ou si elle est vide. Il est possible de la représenter de manière semblable à la pile. Un exemple de file est représenté figure 3.3.

8. First In First Out

Là, dans le fouillis de lettres éparpillées, se trouvait un alignement parfaitement net. Un alignement qui formait deux mots. Et ces deux mots étaient les suivants :

QUARANTE DEUX

Arthur ferma les yeux et plongea la main dans la serviette *de cailloux. Les secoua, en tira quatre et les tendit à Ford.*
– Q, . . . , U, E, L. *Quel !*
– E, S, T, L, . . . , E, P, R, ... *Ça ne veut rien dire du tout, j'en ai peur.*

QUEL EST LE PRODUIT DE SIX PAR NEUF

– *C'est tout. Il n'y a plus d'autre lettre.*

Douglas Adams

4 Calcul numérique

Sommaire

Ce qui fait de Python une alternative confortable [1] à des logiciels commerciaux de calcul matriciel [2] est essentiellement le trio de bibliothèques `numpy` `scipy` et `matplotlib`. Le succès de `numpy` et de sa méthode `dot()` de produit matriciel a même conduit la version 3.5 de Python à introduire un opérateur dédié, `@`.

Les fonctionnalités de calcul matriciel, d'analyse numérique et d'affichage de courbes et de surfaces sont réparties entre trois modules :

numpy : définit les types et opérateurs permettant le calcul matriciel, redéfinit les fonctions mathématiques de *math* pour les rendre compatibles avec ces types.

scipy : définit les fonctions complexes de manipulation de matrices, ainsi que des schémas d'intégration pour les fonctions et équations différentielles. Utilise les types *numpy*.

matplotlib : définit les fonctions d'affichage et d'exportation vers de nombreux formats de fichier pour les types *numpy*, le plus souvent lorsqu'ils sont issus d'un calcul.

Cet ensemble de modules constitue une alternative raisonnable à des logiciels spécialisés tels que *Matlab*, *Mathematica*, *octave* ou *Scilab*. De plus, il permet l'intégration de l'analyse numérique et du calcul matriciel dans un langage universel.

1. C'est-à-dire gratuite, pratique, riche d'une large communauté, mais souffrant de quelques lenteurs à l'exécution
2. Par exemple Matlab

Il faut bien admettre, toutefois, que pour de l'analyse numérique pure, matlab ou octave permettront un développement ou une exécution significativement plus rapide qu'avec Python, même muni de numpy. Et que, comme pour le reste du langage, un code écrit dans un langage compilé permettra une exécution plus rapide et/ou une consommation électrique plus modeste.

Pour les usages plus simples, le module *math* contient les fonctions et constantes mathématiques standards. Le module *cmath* contient l'équivalent pour les nombres complexes. La documentation complète de *math* est disponible à l'adresse `https://docs.python.org/3/library/math.html`, et la documentation de *cmath* à l'adresse `https://docs.python.org/3/library/cmath.html`.

4.1 math

Les tables 4.1, 4.3, 4.4, 4.5 et 4.6 regroupent les principales fonctions et constantes de *math*.

Table 4.1: Arrondis, fonctions classiques

Fonction	Description
`ceil(x)`	Arrondi à la valeur entière supérieure
`floor(x)`	Arrondi à la valeur entière inférieure
`trunc(x)`	Retourne un réel égal à la valeur tronquée de x
`fabs(x)`	Valeur absolue de x
`fmod(x, y)`	Retourne x module y
`isfinite(x)`	Retourne vrai sauf si x est infini ou si x vaut NaN
`isinf(x)`	Retourne vrai si x vaut $+\infty$ ou $-\infty$

Table 4.2: Fonctions puissance et logarithme

Fonction	Description
`exp(x)`	e^x
`expm1(x)`	$e^x - 1$; plus précis pour les petites valeurs de x
`log(x)`	Logarithme naturel de x
`log2(x)`	Logarithme en base 2 de x
`log10(x)`	Logarithme en base 10 de x
`sqrt(x)`	$\sqrt{x}$

Table 4.3: Fonctions trigonométriques

Fonction	Description
`sin(x)`	Sinus
`cos(x)`	Cosinus
`tan(x)`	Tangente
`arcsin(x)`	Arcsinus
`arccos(x)`	Arccosinus
`atan(x)`	Arctangente
`hypot(x1, x2)`	Hypoténuse en fonction des deux autres cotés
`atan2(x1, x2)`	Arctangente du rapport x1/x2 (quadrant selon x1 et x2)
`degrees(x)`	Conversion de radians vers degrés
`radians(x)`	Conversion de degrés vers radians

Table 4.4: Fonctions trigonométriques hyperboliques

Fonction	Description
`sinh(x)`	Sinus hyperbolique
`cosh(x)`	Cosinus hyperbolique
`tanh(x)`	Tangente hyperbolique
`asinh(x)`	Arcsin hyperbolique
`acosh(x)`	Arccosinus hyperbolique
`atanh(x)`	Arcsinus hyperbolique

Table 4.5: Fonctions spéciales

Fonction	Description
`erf(x)`	Fonction d'erreur [3]
`erfc(x)`	Complémentaire de la fonction d'erreur [4]
`gamma(x)`	Fonction Gamma [5]
`gammac(x)`	Complémentaire de la fonction Gamma [6]

Table 4.6: Constantes

Fonction	Description
`pi`	π
`e`	e

4.2 cmath

Toutes les fonctions du module *cmath* supportent des arguments de type entier, réel ou complexe. La représentation interne des nombres complexes est sous forme rectangulaire (partie réelle et partie imaginaire). Il est possible de représenter des complexes sous forme polaire grâce aux fonctions de la table 4.7.

Les fonctions exponentielle, logarithme, trigonométriques, et trigonométriques hyperboliques sont définies dans *cmath*. Leur utilisation est identique aux fonctions de même nom du module *math* (tables 4.2, 4.3 et 4.4).

Table 4.7: Conversion de format

Fonction	Description
`phase(z)`	Phase ou argument du complexe z
`polar(z)`	Retourne z sous forme de tuple (r, φ)
`rect(r, phi)`	Retourne le complexe décrit en coordonnées polaires par (r, φ)

3. $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$
4. 1 - erf(x)
5. $\Gamma(x) = \int_0^{+\infty} t^{x-1} e^{-t} dt$
6. $1 - \Gamma(x)$

4.3 numpy

4.3.1 Les types sous Numpy

L'importation de la bibliothèque Numpy augmente le nombre des types possibles. En plus des entiers classiques (int), on peut considérer les entiers positifs (uint). On peut aussi spécifier le nombre de bits utilisé pour le codage. La liste des types devient alors :

bool : `True` ou `False` ; toute valeur numérique différente de 0 est assimilée à `True` ;

int : le type entier par défaut de Python ; sa taille n'est pas définie par la norme, et sa taille par défaut correspond à `int32` ou `int64` selon les architectures. Il est automatiquement étendu pour s'adapter à la taille de l'entier représenté.

int8 : de -128 à 127,

int16 : de -32 768 à 32 767,

int32 : de -2 147 483 648 à 2 147 483 647,

int64 -9 223 372 036 854 775 808 à 9 223 372 036 854 775 807,

uint8 : de 0 à 255,

uint16 : de 0 à 65 535,

uint32 : de 0 à 4 294 967 295,

uint64 : de 0 à 18 446 744 073 709 551 615,

float64, float : nombre à virgule flottante sur 8 octets ; correspond au type `double` du C.

float32 : nombre à virgule flottante sur 4 octets ; correspond au type `float` du C.

float16 : nombre à virgule flottante sur 2 octets ; il ne correspond à aucun type du microprocesseur : il permet de gagner de l'espace de stockage, mais pas de vitesse de calcul ;

complex128, complex : nombre complexe dont la partie réelle et la partie imaginaire sont des `float64`, c'est-à-dire des `float`.

complex64 : nombre complexe dont la partie réelle et la partie imaginaire sont des `float32`.

4.3.2 Quelques fonctions numériques sous Numpy

La listes des fonctions numériques définies dans `numpy` est extrêmement importante. On peut ici donner une liste non exhaustive des fonctions les plus classiques.

Remarque : les termes entre [et] sont des paramètres optionnels. Notamment, le paramètre optionnel *out*, fréquemment cité, conduit la fonction à écrire le résultat dans le tableau *out* fourni plutôt que d'en créer un nouveau et le renvoyer à l'appelant. Si x et *res* sont des tableaux de même taille, on pourra écrire `cos(x, res)` plutôt que `res=cos(x)` pour gagner le temps de l'affectation d'un nouveau tableau, et la place mémoire de l'ancien *res* le temps que le ramasse-miettes s'en occupe.

Table 4.8: Trigonométrie

Fonction	Description
`sin(x[, out])`	Sinus
`cos(x[, out])`	Cosinus
`tan(x[, out])`	Tangente
`arcsin(x[, out])`	Arcsinus
...	

...	
`arccos(x[, out])`	Arccosinus
`arctan(x[, out])`	Arctangente
`hypot(x1, x2[, out])`	Hypoténuse en fonction des deux autres cotés
`arctan2(x1, x2[, out])`	Arctangente du rapport x1/x2 (quadrant selon x1 et x2)
`degrees(x[, out])`	Conversion de radians vers degrés
`radians(x[, out])`	Conversion de degrés vers radians
`unwrap(p[, discont, axis])`	Déroulement de la phase
`deg2rad(x[, out])`	Conversion de degrés vers radians
`rad2deg(x[, out])`	Conversion de radians vers degrés

TABLE 4.9: Trigonométrie hyperbolique

Fonction	**Description**
`sinh(x[, out])`	Sinus hyperbolique
`cosh(x[, out])`	Cosinus hyperbolique
`tanh(x[, out])`	Tangente hyperbolique
`arcsinh(x[, out])`	Arcsin hyperbolique
`arccosh(x[, out])`	Arccosinus hyperbolique
`arctanh(x[, out])`	Arcsinus hyperbolique

TABLE 4.10: Arrondis

Fonction	**Description**
`around(a[, decimals, out])`	Arrondi à une décimale près
`rint(x[, out])`	Entier le plus proche
`fix(x[, y])`	Arrondi à 0 décimale
`floor(x[, out])`	Arrondi à la valeur inférieure
`ceil(x[, out])`	Arrondi à la valeur supérieure
`trunc(x[, out])`	Arrondi à l'entier le plus proche de zéro

TABLE 4.11: Exponentielles et logarithmes

Fonction	**Description**
`exp(x[, out])`	Exponentielle
`expm1(x[, out])`	Exponentielle moins 1
`exp2(x[, out])`	Exponentielle de base 2
`log(x[, out])`	Logarithme naturel
`log10(x[, out])`	Logarithme de base 10
`log2(x[, out])`	Logarithme de base 2
`log1p(x[, out])`	Logarithme de $x + 1$

Table 4.12: Fonctions spéciales

Fonction	**Description**
`i0(x)`	Bessel modifiée de première espèce d'ordre 0
`sinc(x)`	Sinus cardinal
De nombreuses autres fonctions spéciales existent dans Numpy mais elles appartiennent à des sous-bibliothèques.	

Table 4.13: Arithmétique

Fonction	**Description**
`add(x1, x2[, out])`	Addition
`reciprocal(x[, out])`	Inversion (Attention au cas entier)
`negative(x[, out])`	Opposé
`multiply(x1, x2[, out])`	Multiplication
`divide(x1, x2[, out])`	Division
`power(x1, x2[, out])`	Puissance $x1^{x2}$
`subtract(x1, x2[, out])`	Soustraction
`true_divide(x1, x2[, out])`	Vraie division (même entiers)
`floor_divide(x1, x2[, out])`	Quotient division entière (même flottants)
`fmod(x1, x2[, out])`	Reste division entière
`mod(x1, x2[, out])`	Reste division entière
`remainder(x1, x2[, out])`	Reste division entière

Table 4.14: Autres fonctions

Fonction	**Description**
`signbit(x[, out])`	Indication de la présence d'un bit de signe
`copysign(x1, x2[, out])`	Copie du signe de x2 sur x1
`modf(x[, out1, out2])`	Partie décimale et partie entière
`frexp(x[, out1, out2])`	Décomposition $x = \text{out1} * 2^{\text{out2}}$
`ldexp(x1, x2[, out])`	Calcul $\text{x1} * 2^{\text{x2}}$

Table 4.15: Nombres complexes

Fonction	**Description**
`angle(z[, deg])`	Argument
`real(val)`	Partie réelle
`imag(val)`	Partie imaginaire
`conj(x[, out])`	Conjugué

Table 4.16: Divers

Fonction	**Description**
`sqrt(x[, out])`	Racine carrée
`square(x[, out])`	Mise au Carré
`absolute(x[, out])`	Valeur absolue
...	

...	
`fabs(x[, out])`	Valeur absolue
`sign(x[, out])`	Signe (-1,0,1)
`maximum(x1, x2[, out])`	Maximum
`minimum(x1, x2[, out])`	Minimum
`nan_to_num(x)`	Remplacement de nan et de inf
`real_if_close(a[, tol])`	Élimination des petites parties imaginaires

Les constantes `numpy.nan` (Not A Number) et `numpy.inf` (Infini) sont définies dans la bibliothèque Numpy.

4.3.3 Définir une matrice en Python basique

En partant des types fondamentaux, il est possible de créer des matrices et des vecteurs en utilisant le type *list* usuel (défini en section 2.1.5). Une liste se crée en utilisant `[ ]`. On rappelle que pour une liste de taille n, l'indice varie de 0 à $n-1$.

```
>>> a=[1.3,2.0,7.6]
>>> a[0]
1.3
>>> a[2]
7.6
```

Pour créer une matrice, il est possible de créer une liste de liste :

```
>>> b=[[1,2,3],[4,5,6],[7,8,9]]
>>> print(b)
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> b[1]
[4, 5, 6]
>>> b[1][2]
6
```

Les fonctions `len` et `range` permettent de créer et de manipuler des listes comme des listes de listes.

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(6,9))
[6, 7, 8]
>>> range(1.2,5)
Traceback (most recent call last)~:
  File "<stdin>", line 1, in <module>
TypeError: 'float' object cannot be interpreted as an integer
>>> list(range(1,10,2))
[1, 3, 5, 7, 9]
>>> list(range(10,1,-2))
[10, 8, 6, 4, 2]
>>> a=range(4,8)
>>> len(a)
4
>>> b=[[1,2,3,4],[5,6,7,8],[9,10,11,12]]
>>> print(b)
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
>>> len(b)
```

```
3
```

Quoi qu'il en soit, il ne sera pas possible de travailler sur des listes ou des listes de listes de façon algébrique. Par exemple, le produit direct de listes conduit à une erreur :

```
>>> a=[1.3,2.0,7.6]
>>> b=[[1,2,3],[4,5,6],[7,8,9]]
>>> b*a
Traceback (most recent call last)~:
  File "<stdin>", line 1, in <module>
TypeError~: can't multiply sequence by non-int of type 'list'
```

Pour réaliser des opérations algébriques, nous sommes condamnés à reprogrammer les opérations fondamentales de l'algèbre linéaire. En ajoutant le fait qu'en Python de base les opérations arithmétiques et fonctions numériques sont limitées, l'emploi de bibliothèques mathématiques est indispensable pour des applications scientifiques.

4.3.4 Le type matrix sous Numpy

En plus des types de nombre et des fonctions numériques vues précédemment, la bibliothèque `numpy` apporte également de nouvelles structures. En particulier, il existe la structure `matrix` qui n'est pas une simple liste. Ainsi, le calcul algébrique suivant :

$$\vec{v} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \quad A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 2 \end{pmatrix} \quad \vec{u} = A \cdot \vec{v}$$

peut se traduire sous Python par la séquence :

```
>>> v = np.matrix([1,2,3])
>>> print(v)
[[1 2 3]]
>>> A = np.matrix([[1,0,4],[0,-1,0],[0,0,2]])
>>> print(A)
[[ 1  0  4]
 [ 0 -1  0]
 [ 0  0  2]]
>>> B = np.matrix('1, 0, 4 ; 0, -1, 0 ; 0, 0, 2')
>>> print(B)
[[ 1  0  4]
 [ 0 -1  0]
 [ 0  0  2]]
>>> A*v
raceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3/dist-packages/numpy/matrixlib/defmatrix.py",
    line 341, in __mul__
    return N.dot(self, asmatrix(other))
ValueError: objects are not aligned
>>> vt = np.transpose(v)
>>> print(vt)
[[1]
 [2]
 [3]]
>>> A*vt
[[13]
 [-2]
 [ 6]]
```

L'avantage de la structure `matrix` de `numpy` est que l'opérateur `*` a le même sens que la multiplication au sens des matrices et que les opérations algébriques deviennent transparentes.

Par ailleurs, il existe un grand nombre de fonctions applicables au type matrix : `transpose`, `trace`, `diagonal`, `reshape`, ...

Par exemple, pour transposer une matrice, on peut écrire sous Python :

```
>>> A=np.matrix([[1,2,3],[4,5,6],[7,8,9]])
>>> np.transpose(A)
matrix([[1, 4, 7],
        [2, 5, 8],
        [3, 6, 9]])
>>> A.T
matrix([[1, 4, 7],
        [2, 5, 8],
        [3, 6, 9]])
```

Toutefois, le type `matrix` de `numpy` possède de sérieuses limitations. En particulier, il n'est pas possible de travailler en dimension supérieure à 2.

4.3.5 Le type array sous Numpy

Le type `array` est plus général. Il permet de travailler en dimension quelconque et *c'est la structure qui est privilégiée sous* `numpy`. Avec le type `array`, il est possible de réaliser quasiment les mêmes opérations qu'avec le type `matrix`. Toutefois, la séquence précédente réécrite en utilisant le type `array`, présente certaines différences. Notez bien les simples ou doubles crochets qui marquent les dimensions des tableaux.

```
>>> import numpy as np
>>> v=np.array([1,2,3])
>>> print(v)
[1 2 3]
>>> v
array([1, 2, 3])
>>> A = np.array([[1,0,4],[0,-1,0],[0,0,2]])
>>> print(A)
[[ 1  0  4]
 [ 0 -1  0]
 [ 0  0  2]]
>>> A
array([[ 1,  0,  4],
       [ 0, -1,  0],
       [ 0,  0,  2]])
>>> A*v
array([[ 1,  0,  12],
       [ 0, -2,  0],
       [ 0,  0,  6]])
>>> A@v
array([13, -2,  6])
>>> vt = np.transpose(v)
>>> A*vt
array([[ 1,  0,  12],
       [ 0, -2,  0],
       [ 0,  0,  6]])
>>> A@vt
array([13, -2,  6])
>>> A*A
array([[ 1,  0, 16],
```

```
       [ 0,  1,  0],
       [ 0,  0,  4]])
>>> A@A
array([[ 1,  0, 12],
       [ 0,  1,  0],
       [ 0,  0,  4]])
>>> v2 = np.array( [[1 ,2 , 3 ]] )
>>> v2
array([[1, 2, 3]])
>>> np.transpose(v2)
array([[1],
       [2],
       [3]])
>>> A*np.transpose(v2)
array([[ 1,  0,  4],
       [ 0, -2,  0],
       [ 0,  0,  6]])
>>> A.dot(v2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: objects are not aligned
>>> A.dot(np.transpose(v2))
array([[13],
       [-2],
       [ 6]])
>>> print(A.shape, v.shape, v2.shape, v2.transpose().shape)
(3, 3) (3,) (1, 3) (3, 1)
```

Dans l'exemple précédent, `A` est une matrice 3×3, `v` est un vecteur à une dimension, donc un `array` dont la forme (`shape`) est (3), alors que `v2` est un vecteur ligne, mais un `array` à deux dimensions, dont la forme est (1,3). Lorsqu'un vecteur mono-dimensionnel est utilisé dans un contexte où il faudrait deux dimensions, il est en général considéré comme un vecteur ligne.

La transposée de `v` est `vt` qui est identique à `v`, alors que la transposée de `v2` est bien un vecteur colonne de forme (3,1).

L'opérateur de multiplication `*` réalise une multiplication terme à terme ; c'est ainsi que les termes de `A*A` sont les $a_{i,j}^2$. Les multiplications par `v`, `vt` ou `v2`, qui sont tous des vecteurs ligne, multiplient la i^e colonne de `A` par le i^e terme du vecteur. La multiplication par `np.transpose(v2)` multiplie la i^e ligne de la matrice par le terme correspondant du vecteur.

Le produit matriciel classique s'obtient par l'opérateur `@`[7] ou par la méthode `dot`, qui impose au paramètre d'être de la bonne forme : on ne peut pas multiplier `A` par `v2`, mais juste par sa transposée.

Voici quelques-uns des méthodes, champs et opérateurs les plus utiles de la classe *numpy.array* :

shape : donne la dimension du tableau, sous forme d'un tuple dont chaque composante est le nombre d'éléments selon la dimension correspondante. Une matrice aura une forme *shape* à deux éléments. Un vecteur aura une forme à un élément, ou une forme à deux éléments dont l'un vaut 1, selon l'usage qui en est fait.

Note: `len(A)` renvoie la case numéro 0 de `A.shape()`. C'est rarement ce qui est souhaité.

dtype : informe sur le type *numpy* commun aux éléments du tableau (cf. section 4.3.1).

transpose([axes]) : intervertit les deux axes pour un tableau à deux dimensions ; peut recevoir un tuple donnant le nouvel ordre des axes, dont les éléments sont une permutation de *(0, . . . , n-1)*.

7. À partir de Python 3.5.

diagonal([offset]) : renvoie une vue sur la *offset*^{e} diagonale du tableau ; *la* diagonale si *offset* est omis ou vaut 0. Il s'agit d'une *vue* : elle est utilisable en écriture comme en lecture.

dot(B) : effectue la multiplication matricielle avec le tableau B passé en paramètre. Le nombre de colonnes du tableau invoquant la méthode doit correspondre au nombre de lignes de B. Pour les dimensions supérieures, il vaut mieux utiliser spécifiquement `tensordot()`.

opérateurs + et - : fonctionnent ainsi que l'attend l'algèbre linéraire.

opérateurs *, / et ** : fonctionnement terme à terme.

opérateur @ : effectue la muptiplication matricielle.

inv(A) : calcule l'inverse d'une matrice carrée. Il ne s'agit pas d'une méthode, mais d'une fonction définie dans le module *numpy.linalg*.

constructeur `array` : construit un tableau à partir de l'objet passé en paramètre ; en général, il convient de passer une liste de listes de même longueur. Le constructeur infère le type de données à utiliser pour le *numpy.array* à partir du contenu de la liste.

atleast_1d, atleast_2d, atleast_3d(data) : créent un tableau *numpy.array* ayant un nombre de dimensions au moins `n` contenant pour chaque donnée passée en paramètre. Si la donnée a une dimension supérieure ou égale, elle est conservée. Par ces fonctions, une liste à m éléments devient un tabelau à une dimension à m éléments par `atleast_1d`, un tableau à une ligne et m colonnes par `atleast_2d`, et un tableau de dimension $1 \times m \times 1$ par `atleast_3d`.

arange(deb, fin, inc) : crée un tableau monodimensionnel dont la première case contient *deb*, et les suivantes suivent une suite arithmétique de raison *inc*. La dernière case contient la plus grande valeur de cette suite *strictement* inférieure à *fin*. Contrairement à la fonction de base *range*, *arange* peut recevoir des arguments non-entiers, et crée directement un tableau, sans passer par un itérateur.

linspace(deb, fin, nbelt) : crée un tableau mono-dimensionnel à *nbelt* cases, réparties de façon affine, dont la première vaut *deb*, et la dernière vaut *fin*.

4.4 Fonctions de haut niveau : scipy

Si les types de base (nombres, matrices, tableaux), leurs opérateurs, et leurs méthodes les plus basiques sont définies dans *numpy*, les fonctions de haut niveau, qui implémentent les notions d'analyse numérique, sont en général définies dans l'un des nombreux sous-modules de *scipy*

linalg : module d'algèbre linéaire. Dispose de fonctions d'inversion de matrice, de diagonalisation, de résolution de système linéaire, de calcul de norme matricielle, de décompositions, et de fonctions matricielles.

integrate : module d'intégration numérique. Dispose de fonctions de calcul approché d'intégrales, mais aussi de simulation d'équations aux dérivées partielles.

optimize : module permettant la recherche de racines d'une fonction, mais aussi détermination des minima, avec ou sans contraintes, et obtention de paramètres par la méthode des moindres carrés.

stats : module mettant en place nombre de lois de probabilité, des tirages aléatoires correspondant, et des tests statistiques [8].

fftpack, signal, ndimage : *signal* est un module traitement du signal 1D et 2D, avec de nombreux filtres [9]. La transformée de Fourier rapide est définie dans *fftpack*. Le module *ndimage* permet des liens avec des fonctions externes écrites en C plus simplement que *signal*.

8. Notions mathématiques vues en UV 2.1.
9. Notions vues en UV 1.5, 2.5 et 3.5

4.5 Tracé de courbes

Si plusieurs solutions sont utilisables pour dessiner dans une fenêtre graphique (Tkinter, turtle, PyQT, ...), la plus pratique pour représenter le graphe d'une fonction réelle ou la plupart des données mathématiques à une ou deux dimensions est la bibliothèque *matplotlib* [Hun07]. Elle permet également la représentation en 3D et la séparation d'une fenêtre en sous-schémas.

Le cœur du module est la classe *Axes* ; une instance de cette classe contient un schéma. La fenêtre graphique contenant un ou plusieurs objets *Axes* est une *Figure*. Il y a essentiellement deux façons de travailler avec le module *matplotlib*.

Utiliser pylab : L'importation du module *pylab* force l'importation des contenus de *numpy*, *scipy* et *matplotlib* dans le même espace de nommage. Il n'y a presque pas de collision de noms (celles qui se produisent concernent en général des fonctions identifiées comme obsolètes). Cela permet d'avoir rapidement une calculatrice de haut niveau.

Listing 4.1 – Tracé de courbe par *pylab*

```
1 from pylab import *
2
3 x = linspace(0, 3*np.pi, 42)
4 y = cos(x)
5 plot(x, y)
6 show()
```

Utiliser pyplot : Le module *pyplot* de *matplotlib*, lors de son chargement (le plus souvent par `import matplotlib.pyplot as plt`), se charge de créer une figure prête à l'usage. On y accède par l'objet `plt`

Listing 4.2 – Tracé de courbe par *pyplot*

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.linspace(0, 3*np.pi, 42)
5 y = np.cos(x)
6 plt.plot(x, y)
7 plt.show()
```

Créer manuellement une figure : Le module *pyplot* donne accès à la fonction *figure*, qui crée un objet *Figure* et le gère. Il est donc possible de l'instancier, ce qui permet éventuellement d'avoir plusieurs fenêtres dans un même programme. Attention, dans l'exemple suivant, le programme s'arrête après la ligne 10, et la figure se referme.

Listing 4.3 – Tracé de courbe par *figure*

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.linspace(0, 3*np.pi, 42)
5 y = np.cos(x)
6
7 fig = plt.figure()
8 ax = fig.add_subplot(1, 1, 1)
9 ax.plot(x, y)
10 fig.show()
```

Les deux premières solutions sont recommandées dans le cas d'un programme dont le seul objet est de réaliser un calcul et d'afficher son résultat. Si la fenêtre de tracé de courbe est une petite partie d'un projet plus vaste, et notamment si le programme doit continuer à travailler après l'affichage de la courbe, la troisième solution est à privilégier.

4.5.1 L'objet `Axes`

La classe Axes est la cheville ouvrière de la bibliothèque *matplotlib*. Toutefois, elle est rarement invoquée directement, et *matplotlib* fournit des contenants de plus haut niveau pour aider à sa gestion, grâce notamment au module *pyplot*. De nombreux didacticiels permettent de construire par mimétisme la création de très beaux graphiques [10] mais pour intégrer des éléments graphiques de *matplotlib* dans une fenêtre pilotée par un projet basé sur PyQT ou Tkinter, il convient de bien gérer les objets, et de renoncer au confort de *pylab* ou *pyplot*.

figure : crée une figure. Elle s'affichera comme une fenêtre indépendante si elle est gérée par *pyplot*. Sinon, il faut lui spécifier une façon de s'afficher, grâce à un descendant de *FigureCanvasBase*.

show : déclenche l'affichage de la figure.

close : demande au gestionnaire de figures de la supprimer. Il ne suffit pas de réaffecter la variable contenant la figure pour que son espace soit perçu comme disponible par le ramasse-miettes.

savefig : le sous-module *backends* contient des outils pour afficher les figures par différents outils graphiques (Qt, Wx, ...). Ces outils permettent en général l'exportation vers des fichiers, aux formats pdf, png, svg ou postscript.

subplot(L, C, n) : Rend actif le n^e sous-schéma dans un quadrillage à L lignes et C colonnes. La valeur de n est comprise entre 1 et $L \times C$. Ainsi `subplot(3, 4, 5)` désigne le premier schéma de la deuxième ligne. Quand $L \times C \leqslant 9$, il est possible de condenser l'appel en collant les trois chiffres : `subplot(325)`. Il est recommandé de ne pas changer les valeurs de L et C pour différents appels à `subplot`.

gca : *get current axes*. La méthode renvoie le système de coordonnées courant, de type *Axes* dans lequel s'appliquent les opérations d'affichage. Il le crée au besoin.

plot(x, y, args, ...) : ajoute une ou plusieurs courbes à une figure. x contient les abscisses des points à tracer, et y leurs ordonnées. Les arguments supplémentaires permettent de déterminer la façon de tracer la courbe, voire de transmettre des informations sur la courbe à d'autres composantes de la figure.

color : peut être un nom de couleur matplotlib (blue, green, red, cyan, magenta, yellow, black, white), une couleur HTML standard [11], un nombre compris entre 0 et 1 (donne un niveau de gris), un tuple de trois nombres entre 0 et 1, donnant les niveaux respectifs de rouge, vert, et bleu, ou une chaîne contenant ces niveaux en hexadécimal : `"#DC143C"`.

linestyle : peut valoir `"-"`, `"--"`, `"-."`, `":"`, `"None"`, `" "` ou `""`. Les trois derniers sont équivalents.

label : associe un nom à la courbe.

Il est possible d'associer un caractère de couleur à un code de style de ligne en une seule chaîne. Dans ce cas, on peut utiliser `plot` pour tracer plusieurs lignes simultanément : `plot(x1, y1, 'g-', x2, y2, 'r:')`

scatter(x, y, args) : x contient les abscisses des points à tracer, et y leurs ordonnées ; `scatter` trace les points correspondants comme des points isolés. L'argument *marker* permet de choisir la forme des points. `s` permet de gérer leur taille, et peut être un nombre commun

10. Précieux pour alimenter des rapports
11. Il y en a 140 : http://www.w3schools.com/html/html_colornames.asp

à tous les points, ou un tableau définissant une valeur par point. `c` permet de gérer leur couleur de la même manière.

xlim, ylim : fixe les bornes de la zone affichée. Par défaut, s'adapte aux données reçues.

legend : place une boîte associant les couleurs de courbe avec les arguments *label* fournis aux instructions `plot`. Les labels définis avec un caractère `r` avant le délimiteur de chaîne de caractères sont compilés par LaTeX avant d'être placés sur le schéma.

annotate : ajoute des informations textuelles et les relie à des éléments graphiques du schéma.

figtext(x, y, texte) : écrit *texte* aux coordonnées *(x, y)* sachant que le point (0,0) est en bas à gauche de la figure, et le point (1,1) en haut à droite.

Listing 4.4 – Tracé de courbe par *pylab*

```
import numpy as np
from pylab import *

""" Author : Nicolas P. Rougier (CC-by) """

figure(figsize=(12,8), dpi=120)
subplot(1,1,1)

X = np.linspace(-np.pi, np.pi, 256,endpoint=True)
C,S = np.cos(X), np.sin(X)

plot(X, C, color="blue", linewidth=2.5, linestyle="-", label="cosine")
plot(X, S, color="red", linewidth=2.5, linestyle="--", label="sine")

ax = gca()
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))

xlim(X.min()*1.1, X.max()*1.1)
xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi],
       [r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$'])

ylim(C.min()*1.1,C.max()*1.1)
yticks([-1, +1],
       [r'$-1$', r'$+1$'])

legend(loc='upper left')

t = 2*np.pi/3
plot([t,t],[0,np.cos(t)],
     color ='blue',  linewidth=.5, linestyle="--")
scatter([t,],[np.cos(t),], 50, color ='blue')
annotate(r'$\sin\left(\frac{2\pi}{3}\right)=\frac{\sqrt{3}}{2}$', xy=(t,
  np.sin(t)),  xycoords='data',
        xytext=(+10, +30), textcoords='offset points', fontsize=16,
        arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2"))

plot([t,t],[0,np.sin(t)],
          color ='red',  linewidth=.5, linestyle="--")
```

```
43 scatter([t,],[np.sin(t),], 50, color ='red')
44 annotate(r'$\cos\left(\frac{2\pi}{3}\right)=-\frac{1}{2}$', xy=(t, np.cos
     (t)),  xycoords='data',
45         xytext=(-90, -50), textcoords='offset points', fontsize=16,
46         arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2"))
47
48 for label in ax.get_xticklabels() + ax.get_yticklabels():
49     label.set_fontsize(16)
50     label.set_bbox(dict(facecolor='white', edgecolor='None', alpha=0.65 )
       )
51
52 savefig("../pdfs/courbes_et_latex.pdf")
53
54 show()
```

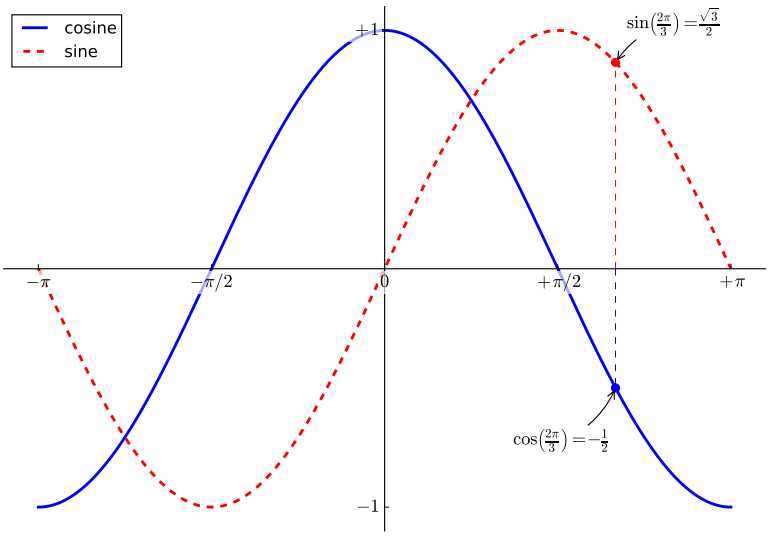


FIGURE 4.1 – Schéma produit – et enregistré – par le programme 4.4

Dans le programme 4.4, la figure créée en ligne 7 n'est pas affectée à une variable. Le module *pylab* dispose d'une figure par défaut, et la figure créée ici la remplace, en précisant la taille voulue. L'instruction `subplot` dit simplement qu'il s'agit du seul graphique dans la fenêtre, et en fait le lieu « actif » pour les instructions graphiques. Les instructions `plot` des lignes 13 et 14 s'appliquent donc à ce graphique.

Certaines fonctions sont des méthodes de la classe *Axes*, et il faut donc disposer explicitement de l'objet. Il est obtenu via la fonction `gca()`, et permet donc d'invoquer `spines`, qui efface l'encadrement des données, et de positionner les points marqués le long des axes.

La légende est fabriquée à partir des informations transmises via le label de `plot`. On pourra noter que les termes de la légende ne sont pas dans la même fonte que les autres : ils sont affichés directement par *matplotlib*, sans passer par LaTeX.

Les lignes 34–47 gèrent la mise en valeur du point $\frac{2\pi}{3}$: `plot` trace les tirets verticaux, `scatter` place les disques bleu ou rouge, et `annotate` ajoute un bloc de texte et le relie au point d'intérêt avec une flèche. Le caractère '`r`' avant la chaîne de caractères indique qu'il doit être évalué avec

LaTeX, ce qui permet d'inclure des expressions mathématiques.

`xticks` et `yticks` créent des listes de labels avec leurs coordonnées. La boucle en lignes 49–51 leur affecte des propriétés graphiques qui permettent leur bon affichage, y compris en gérant la tranparence avec *alpha*.

D'après une théorie, le jour où quelqu'un découvrira exactement à quoi sert l'Univers et pourquoi il est là, ledit Univers disparaîtra sur-le-champ pour se voir remplacé par quelque chose de considérablement plus inexplicable et bizarre.
Selon une autre théorie, la chose se serait en fait déjà produite.

Douglas Adams

5 Programmation orientée objet

Sommaire

5.1 Généralités

La programmation orientée objet consiste à organiser son programme en un ensemble d'objets qui interagissent les uns avec les autres, chacun ayant son propre rôle dans les tâches à accomplir. Pour concevoir un programme en langage orienté objet, il faut définir les types d'objets qui vont constituer le programme : chaque type possède des attributs (des données qui le caractérisent) et des méthodes [1] (des actions qu'il est capable d'effectuer). On initialise ensuite le programme en créant autant d'objets des différents types que nécessaire, et on l'exécute en appelant des méthodes de ces objets.

1. Une méthode est une fonction associée à un objet.

La programmation orientée objet existe depuis l'arrivée du langage *Simula'67* en 1967 et a été marquée par l'arrivée du langage *Smalltalk* en 1972. Ce langage a ensuite été formalisé en tant que *Smalltalk-80*. Cependant, elle n'est vraiment devenue un paradigme de programmation qu'au milieu des années 1980.

La principale différence entre la programmation structurée traditionnelle et la programmation orientée objet est que cette dernière met dans une même structure les données et les opérations qui leur sont associées. En programmation impérative, les données et les opérations associées sont séparées, et cette méthodologie nécessite l'envoi des structures de données aux procédures et fonctions qui les utilisent. La programmation orientée objet résout certains problèmes inhérents à cette conception en mettant dans une même entité les attributs et les opérations. Il est d'usage de présenter la programmation orientée objet comme plus proche du monde réel, dans lequel tous les objets disposent d'attributs auxquels sont associées des activités.

Le programmation orientée objet repose sur quelques concepts fondamentaux :

— l'encapsulation ;
— l'héritage ;
— le polymorphisme.

Ces différents concepts seront abordés partie 5.2.

Il existe des langages de modélisation permettant, entre autres, de représenter graphiquement les classes d'un programme objet et leurs relations. Dans la suite, nous utiliserons UML [2] pour illustrer nos propos.

5.2 Concepts fondamentaux de l'objet

En programmation comme dans de nombreux autres domaines, on cherchera à augmenter la réutilisabilité. Elle passe par la modularité du code, qui se caractérise par le fait que chacun des composants du système sera placé dans une unité fonctionnelle indépendante. C'est pourquoi on conçoit autant que possible chaque module de manière abstraite, en vue de résoudre des problèmes généraux : cela rend plus probable leur réutilisation dans un autre contexte.

Tous les langages de programmation permettent une programmation modulaire, mais elle est plus ou moins simple à mettre en œuvre, et plus ou moins formelle. Ainsi, il est possible de programmer de manière modulaire en C, mais uniquement au prix d'une grande discipline de la part du programmeur.

On notera ici que la modularité ne dépend donc pas du langage, mais de la capacité de l'architecte logiciel à dégager un découpage cohérent de l'application visée, en tirant parti des possibilités du langage.

Voyons maintenant plus en détail les mécanismes facilitant la modularité des programmes en langages orientés objet.

5.2.1 L'encapsulation

Définition 5.1 (Encapsulation)**.** *L'encapsulation est un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en cachant l'implantation de l'objet, c'est-à-dire en empêchant l'accès aux données par un autre moyen que les services proposés. L'encapsulation permet donc de garantir l'intégrité des données contenues dans l'objet (car les accès aux données sont contrôlés par les méthodes).*

Définition 5.2 (Classe, objet, instance)**.** *En programmation orientée objet, un type s'appelle une* classe*. Créer une variable d'une certaine classe s'appelle créer un* objet *ou une* instance*.*

2. Unified Modeling Language

Définition 5.3 (Variables d'instance)**.** *Une variable d'instance ou un attribut est une variable contenue dans un objet.*

Définition 5.4 (Méthode d'instance)**.** *Une méthode d'instance (souvent abrégée en méthode) ou opération est une fonction qui agit sur les attributs d'un objet.*

Il n'est pas possible de faire de l'encapsulation avec des langages impératifs comme le C : il est possible de modulariser, donc de déclarer une intention d'encapsulation, mais le langage ne fournit pas de moyen de faire respecter cette intention première, et le programmeur peut la violer à tout instant. C'est pourquoi nous parlions plus haut de l'auto-discipline du programmeur.

Certains langages impératifs (comme *Ada 83*) permettent une certaine encapsulation grâce à la notion de paquetage. Il est possible de mettre l'encapsulation en œuvre avec des langages fonctionnels (comme *Lisp*, *Scheme* ou *Caml* par exemple) mais aucun mécanisme simple n'est prévu par le langage.

Par contre, tous les langages orientés objet (*C++*, *Java*, *Python*, *Ruby*, ...) prévoient des moyens simples de mettre ce mécanisme en œuvre. Par exemple, ces langages prévoient une méthode particulière appelée *constructeur* dont le rôle est d'initialiser les informations définissant l'objet.

Définition 5.5 (Constructeur)**.** *Le constructeur d'une classe est une méthode servant à initialiser un objet lors de sa création. Son rôle principal est d'initialiser toutes les variables d'instance.*

Considérons par exemple une classe Cercle. Un cercle sera représenté par la position de son centre et par son rayon. Grâce à l'encapsulation, il est possible d'ajouter des contraintes sur ces attributs, comme par exemple imposer que le rayon soit toujours positif. Pour cela, il nous faudra nous intéresser aux entrées des méthodes de création de l'objet et de modification du rayon, et tester que le rayon passé en paramètre est bien positif.

5.2.2 L'héritage

Définition 5.6 (Héritage)**.** *L'héritage est un mécanisme spécifique aux langages orientés objet qui permet à une classe B (appelée sous-classe ou classe fille) d'hériter de toutes les propriétés d'une classe A (appelée super-classe ou classe mère). On dit que la classe B hérite de la classe A, ou encore qu'elle la spécialise. À l'inverse, la classe A est une généralisation de la classe B.*

La notion d'héritage est fondamentale en programmation orientée objet et elle offre de grands avantages en terme de modularité et de réutilisabilité. Elle permet également de « factoriser » du code commun à plusieurs classes. Ce mécanisme repose sur l'abstraction : il nécessite de possèder une vision globale du programme, donc des relations entre classes. Dans ce domaine, la modélisation peut être d'un grand secours, ne serait-ce qu'en proposant une vue graphique générale.

Exemple : nous voulons gérer des figures géométriques pouvant être des cercles ou des rectangles. Chaque figure possède un centre. Les cercles possèdent également un rayon, et les rectangles une longueur et une largeur. Nous constatons que la notion de centre est commune aux deux types de figures. Le problème sera alors découpé en trois classes : une super-classe *Figure* possédant un centre, et deux sous-classes *Cercle* et *Rectangle*. Cet exemple est représenté par le diagramme 5.1.

Remarque : certains langages objet (comme *Python* ou *C++*) permettent l'héritage multiple (une classe peut hériter de plusieurs classes). Par exemple, une classe D peut hériter de deux classes B et C, elles-mêmes héritant d'une classe A. Il n'est alors pas possible d'ordonner la liste d'héritage de la classe D. Python3 possède un algorithme de linéarisation de l'ordre de résolution ; si l'algorithme ne parvient pas à linéariser l'ordre, il lève une exception.

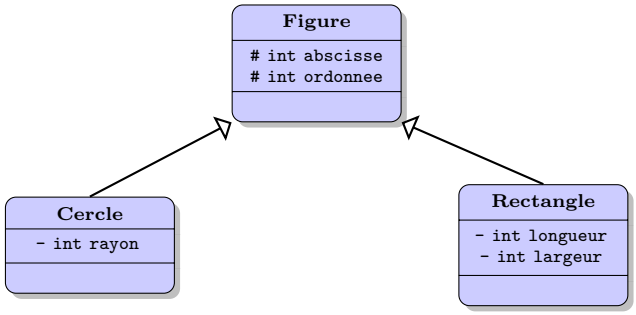


FIGURE 5.1 – Figures

La figure 5.2 et le listing 5.1 représentent un exemple d'héritage multiple. Dans ce cas, l'ambiguïté est levée dans le programme Python par l'ordre des super-classes de *D*. Ici, *D* hérite d'abord de *C*, puis de *B*, et enfin de *A* ; l'ordre d'évaluation des classes est donc *D, C, B, A*.

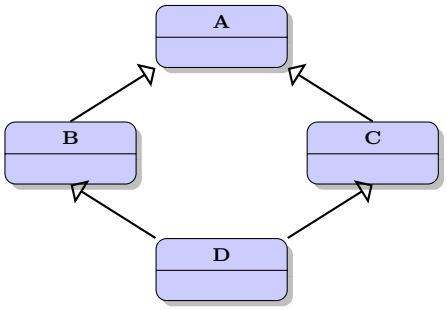


FIGURE 5.2 – Héritage multiple en diamant

Listing 5.1 – Héritage multiple en Python

```
1  class A:
2      def __init__(self):
3          print('Un')
4
5  class B(A):
6      def __init__(self):
7          super().__init__()
8          print('Deux')
9
10 class C(A):
11     def __init__(self):
12         super().__init__()
13         print('Trois')
14
15 class D(C, B):
16     def __init__(self):
17         super().__init__()
18         print('Soleil !')
19
20 x = D()
```

L'exécution du programme 5.1 affiche le résultat suivant :

```
Un
Deux
Trois
Soleil !
```

5.2.3 Polymorphisme

Définition 5.7 (Polymorphisme)**.** *Mécanisme selon lequel un même message peut être envoyé vers des objets de types différents, chaque objet réagissant de façon originale.*

Le polymorphisme est un mécanisme puissant qui permet de rendre la complexité de certaines constructions transparente pour le programmeur. On en distingue plusieurs types, mais nous n'étudierons que le polymorphisme implanté en Python, à savoir le polymorphisme d'héritage (également appelé *redéfinition*, *spécialisation* ou *overriding*).

Le polymorphisme d'héritage

Il consiste à tirer parti de l'héritage en allant chercher dynamiquement la méthode associée à un objet.

Imaginons une application au jeu d'échecs comportant les types *Roi*, *Reine*, *Fou*, *Cavalier*, *Tour* et *Pion*, héritant chacun du type *Piece*.

La méthode *deplacer()*, définie dans la classe *Piece*, pourra, grâce au polymorphisme d'héritage, effectuer le mouvement approprié en fonction du type de l'objet référencé au moment de l'appel. Cela permettra notamment au programmeur de dire *piece.deplacer()* sans avoir à se préoccuper de la classe de la pièce.

Exemple :

```
1 # Déclaration d'une liste de pièces
2 liste_pieces = [Pion(), Cavalier()]
3 # Appliquer la méthode déplacer à chaque pièce
4 for piece in liste_pieces:
5     # La méthode deplacer appelée dépend du type de pièce concerné
6     piece.deplacer()
```

5.2.4 Notion de classe et d'instance

Comme nous l'avons déjà dit, chaque classe va définir un format de données particulier, ainsi que les traitements et accès associés à ces données. Cependant, la définition d'un ensemble de classes ne suffit pas pour construire une application : il est nécessaire de spécifier comment on souhaite les utiliser.

En effet, un programme en exécution ne sera pas composé de classes, mais d'objets issus de ces classes, que l'on nomme aussi *instances* des classes. Une instance d'une classe implante le format défini dans celle-ci, mais avec des valeurs particulières pour les données.

Par exemple, reprenons notre classe *Cercle* et considèrons qu'elle contient deux entiers x et y définissant son centre : nous pouvons en créer deux instances *cercle1* et *cercle2*, dont les (x,y) respectifs sont (0,0) et (3,2).

En résumé, une classe définit un type, et si on souhaite créer des objets de ce type, on instancie la classe (on crée des instances de la classe). En définissant les classes, on définit les outils ; en créant les instances, on les utilise.

5.3 Programmation orientée objet en Python

5.3.1 Classes et objets

En Python, une classe est définie par le mot-clé *class* suivi du nom de la classe. L'héritage est défini entre les parenthèses qui suivent le nom de la classe ; par défaut, la classe héritera de *object*. Il est d'usage de faire commencer le nom de la classe par une majuscule (voir les conventions de nommage annexe A p. 217). Exemple : *Cercle*.

En Python, le constructeur est une méthode nommée `__init__` dont le premier paramètre est *self*. Il est automatiquement appelé lors de la création d'un objet.

Exemple 5.1 (Classe Cercle)**.** *Considérons par exemple la classe* Cercle *suivante (représentée par le diagramme de la figure 5.3), dont le contenu sera détaillé tout au long de ce chapitre :*

Listing 5.2 – Classe Cercle

```
1  class Cercle:
2      """Définition de la classe cercle.
3      """
4      def __init__(self, x=0, y=0, r=1):
5          """Constructeur de la classe Cercle.
6          Initialise les variables d'instance.
7          """
8          self.__abscisse = x
9          self.__ordonnee = y
10         self.__rayon = r
11
12     def double_rayon(self):
13         """Méthode d'instance de la classe Cercle.
14         Accède aux variables d'instance.
15         """
16         self.__rayon *= 2
17
18 if __name__ == '__main__':
19     # Création d'un objet de type Cercle
20     c1 = Cercle(0, 0, 42)  # appelle automatiquement le constructeur
21     # Appel à une méthode de la classe Cercle
22     c1.double_rayon()
23     # Utilisation des paramètres par défaut
24     c2 = Cercle(r=42) # x->0 et y->0
```

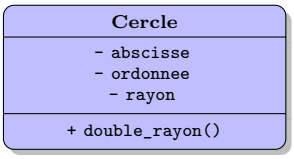


FIGURE 5.3 – Classe Cercle

Il est possible de créer des variables dont le type est une classe (comme par exemple la classe *Cercle*). La variable ainsi créée est alors un objet. Par exemple, l'instruction suivante permet de déclarer une variable *un_objet_cercle* de type *Cercle* :

```
1      un_objet_cercle = Cercle(0, 0, 10)
```

Cette opération crée un espace mémoire contenant toutes les informations sur l'objet, et en particulier les valeurs de ses différentes variables d'instance. La variable *un_objet_cercle* est alors un point d'accès vers cet objet (voir figure 5.4).

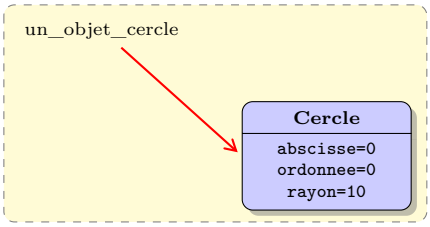


FIGURE 5.4 – Création d'un objet

Il est ensuite possible de déclarer d'autres variables du même type représentant le même objet. Cette opération est réalisée par les instructions suivantes :

```
1    un_autre_objet_cercle = un_objet_cercle
```

On dispose alors de deux points d'accès vers le même objet, c'est-à-dire que les variables *un_objet_cercle* et *un_autre_objet_cercle* référencent le même endroit dans la mémoire. La situation obtenue est représentée figure 5.5.

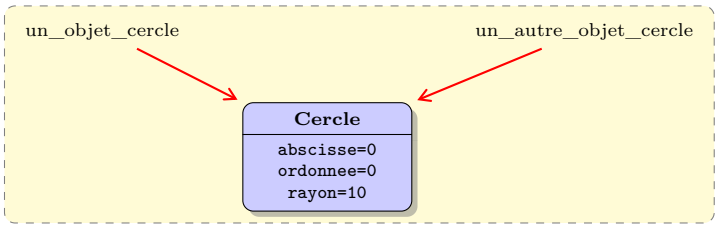


FIGURE 5.5 – Deux références vers un objet

Dans le cas de figure suivant, un deuxième objet est créé, et un deuxième espace mémoire est donc réservé. Afin de réaliser cette opération, il faut appeler le *constructeur* `__init__` une deuxième fois, par l'intermédiaire, toujours, du nom de sa classe :

```
1    un_autre_objet_cercle = Cercle(1, 0, 5)
```

Nous disposons alors de deux objets, chacun possédant ses propres variables d'instance. Les deux objets évoluent alors de manière indépendante. Un tel cas de figure est représenté figure 5.6.

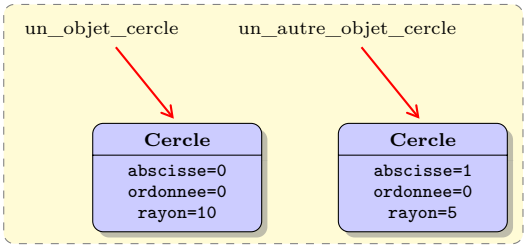


FIGURE 5.6 – Création d'un second objet

Lors de ces opérations, des objets de type *Cercle* sont créés et affectés à des variables qui permettent ensuite de les manipuler. Par exemple :

```
1    un_objet_cercle.double_rayon();
```

L'objet *self*

Le mot-clé *self*, utilisé dans une méthode d'instance, représente un objet particulier : l'objet courant. Considérons l'exemple 5.1, et plus particulièrement la méthode *double_rayon*. Le `self.__rayon` signifie « la variable `__rayon` de l'objet auquel la méthode est appliquée ». Le mot-clé *self* permet donc d'expliciter le contexte dans lequel chercher une certaine variable.

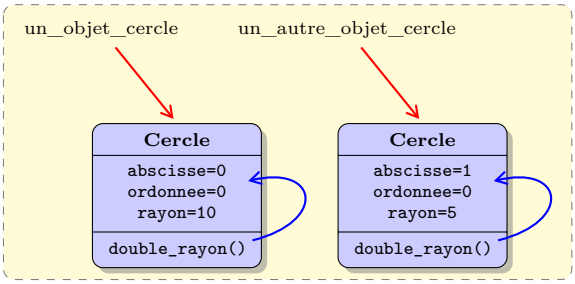


FIGURE 5.7 – Utilisation de *self*

Considérons l'exemple de la figure 5.7 dans lequel nous disposons de deux variables *un_objet_cercle* et *un_autre_objet_cercle* référençant deux objets différents. Lors de l'application de la méthode *double_rayon()* à la variable *un_objet_cercle*, l'instruction `self.__rayon *= 2` est exécutée dans le contexte de l'objet courant. La valeur du rayon de l'objet *un_objet_cercle* est alors doublée.

L'objet *None*

Le mot-clé *None* permet de signaler explicitement qu'une variable ne représente aucun objet. Exemple :

```
1    un_objet_cercle = None
```

Il est alors impossible d'utiliser la variable *un_objet_cercle* tant qu'une nouvelle valeur ne lui a pas été affectée. En cas d'application d'une méthode (ici, *double_rayon()*) sur cette variable, Python lève l'exception suivante :

```
AttributeError: 'NoneType' object has no attribute 'double_rayon'
```

5.3.2 Variables et méthodes d'instance

Les attributs d'une classe sont appelés variables d'instance. Par exemple, les variables d'instance de la classe *Cercle* sont *abscisse, ordonnee* et *rayon*. Ces variables sont utilisables dans toutes les méthodes d'instance de la classe.

Remarque : les variables et méthodes d'instance sont définies en opposition aux variables et méthodes de classe (définies section 5.3.6).

Les méthodes d'instance sont des méthodes qui appartiennent aux instances, en ce sens qu'elles vont les utiliser pour modifier la valeur de leurs variables d'instance, ou pour effectuer

des traitements à partir de celles-ci. Dans tous les cas, il s'agira d'un traitement sur des données propres à l'instance, qui fournira un résultat local.

Visibilité des attributs

En Python, tous les attributs d'un objet ou d'une classe sont publics. Ils sont donc accessibles depuis n'importe quel contexte. Il n'est pas possible de limiter la portée des variables d'instance à l'objet courant, mais par convention, une variable dont le nom commence par un tiret bas (caractère `_`) doit être considérée comme privée. Attention, il s'agit uniquement d'une convention et la variable est toujours accessible.

Si le nom de la variable commence par deux tirets bas et se termine par au plus un tiret bas, elle est renommée dans la représentation interne de l'objet par *_nomDeLaClasse_nomDeLaVariable* ; elle n'est donc pas accessible directement. Ce phénomène est illustré par le listing 5.3 : la variable d'instance `__z` de la classe `A` est renommée en `_A__z`.

Listing 5.3 – Visibilité des variables d'instance

```
1  class A:
2      def __init__(self):
3          self.x = 1   # Variable publique
4          self._y = 4  # Variable privée par convention
5          self.__z = 9 # Variable privée : renommée en interne en _A__z
6
7  if __name__ == '__main__':
8      # Création de l'instance
9      obj = A()
10     # Affichage des variables d'instance de obj
11     print(vars(obj))
12     # Accès à x autorisé
13     print(obj.x)
14     # Accès à _y possible mais non recommandé
15     print(obj._y)
16     # Accès à __z impossible : il faut passer par _A__z
17     # print(obj.__z)  -> Erreur
18     print(obj._A__z)
```

Son exécution produit :

```
{'_A__z': 9, '_y': 4, 'x': 1}
1
4
9
```

Accesseurs

Le principe d'encapsulation incite à protéger les variables d'instances et à n'y accéder qu'au travers de méthodes. Il est alors possible de contrôler l'accès aux variables (par exemple à l'aide de tests), de gérer l'accès concurrent aux données, ...

Il existe deux moyens classiques pour réaliser ce contrôle : la création de méthodes *get* et *set* et l'utilisation de propriétés.

Le listing 5.4 reprend l'exemple de la classe *Cercle* et lui ajoute un accès à la variable *rayon* par des méthodes *get* et *set*. L'accès au rayon du cercle est alors contrôlé, mais la syntaxe pour lire ou modifier cette variable est beaucoup plus lourde.

Listing 5.4 – Méthodes *get* et *set*

```
1  class Cercle:
```

```
    def __init__(self, x=0, y=0, r=1):
        self.__abscisse = x
        self.__ordonnee = y
        self.set_rayon(r)  # Appelle explicitement le setter

    def get_rayon(self):
        return self.__rayon

    def set_rayon(self, r):
        if r >= 0:
            self.__rayon = r
        else:
            self.__rayon = 0

if __name__ == '__main__':
    # Initialisation avec les paramètres par défaut :
    # x->0, y->0, r->1
    c1 = Cercle()
    c1.set_rayon(-12)  # Appel au setter : le rayon vaut 0
    print(c1.get_rayon())  # Acces au rayon via le getter
    # Grâce au setter, c2 a un rayon de 0
    c2 = Cercle(r=-42)
```

Décorateurs

Afin de permettre un contrôle d'accès aux données tout en gardant une syntaxe plus légère, le langage Python prévoit un mécanisme de décorateurs ; le listing 5.5 illustre ce mécanisme.

Le principe est de remplacer la méthode *get_rayon* par une méthode *rayon* devant laquelle figure le décorateur *@property*. On indique ainsi à Python que l'attribut *rayon* est en lecture seule.

Il reste alors à remplacer la méthode *set_rayon* par une autre méthode *rayon* devant laquelle figure le commentaire *@rayon.setter*. Cette méthode sera appelée pour toute modification de l'attribut *rayon* d'un objet de type *Cercle*.

Grâce à ce mécanisme, le contrôle d'accès reste présent, mais l'appel aux méthodes se fait de manière transparente.

Listing 5.5 – Décorateur

```
class Cercle:
    def __init__(self, x=0, y=0, r=1):
        self.__abscisse = x
        self.__ordonnee = y
        self.rayon = r  # Appelle implicitement le setter

    """Le @property transforme la variable rayon
    en attribut en lecture seule.
    """
    @property
    def rayon(self):
        return self.__rayon

    """Le @rayon.setter indique que la méthode sert
    à modifier l'attribut rayon.
    """
    @rayon.setter
    def rayon(self, r):
        if r >= 0:
```

```
            self.__rayon = r
        else:
            self.__rayon = 0

if __name__ == '__main__':
    # Initialisation avec les paramètres par défaut :
    # x->0, y->0, r->1
    c1 = Cercle()
    c1.rayon = -12  # Appel au setter : le rayon vaut 0
    print(c1.rayon)  # Acces au rayon via la propriété
    # Grâce au setter, c2 a un rayon de 0
    c2 = Cercle(r=-42)
```

5.3.3 Héritage

Une des principales propriétés de l'approche objet, l'héritage, permet d'étendre une classe, *i.e.* créer une classe-fille qui possédera toutes ses propriétés, plus d'autres qu'il conviendra de définir.

L'héritage est spécifié en Python entre parenthèses lors de la déclaration de la classe fille. Par exemple, considérons la classe Cercle héritant de la classe Figure. On définit tout d'abord la classe Figure, comme conteneur pour l'abscisse et l'ordonnée du centre des figures. Ainsi, toute Figure a un centre.

```
class Figure:
    def __init__(self, x, y):
        self._abscisse = x
        self._ordonnee = y
```

L'écriture de la classe Cercle est alors simplifiée en :

Listing 5.6 – Héritage

```
class Cercle(Figure):
    def __init__(self, x=0, y=0, r=1):
        super().__init__(x, y)
        self._rayon = r
```

De même, on définit la classe Rectangle :

```
class Rectangle(Figure):
    def __init__(self, x=0, y=0, l=1, h=1):
        super().__init__(x, y)
        self._largeur = l
        self._hauteur = h
```

On a ainsi factorisé les activités liées à la gestion du centre dans la classe Figure, et simplifié d'autant les classes Cercle et Rectangle. On a aussi facilité l'ajout éventuel d'autres formes de figures.

Pour faire référence à la super-classe, on utilise le mot-clé *super*. Ainsi, pour accéder à une variable de la super-classe :

```
super().nom_de_variable
```

Remarque : si aucun héritage n'est précisé explicitement, alors la classe hérite de *object*. Ainsi, toutes les classes en héritent directement ou indirectement, et elles ont donc accès à toutes les méthodes de cette classe.

Grâce à l'héritage, il est possible de manipuler des données de types similaires de manière uniforme. Ainsi dans le listing 5.7, on manipule des cercles et des rectangles (sous-classes de *Figure*) de manière uniforme grâce à l'héritage et au polymorphisme.

Listing 5.7 – Utilisation de l'héritage

```
# creation d'une liste de figures
mesFigures = []
# la première figure est un cercle
mesFigures.append(Cercle(12, 23, 42))
# la deuxième figure est un rectangle
mesFigures.append(Rectangle(421, 666, 33, 3))
# affichage de l'aire des figures
for f in mesFigures:
    print(f.aire())
```

Constructeur

Lors de la définition d'un constructeur dans une classe-fille, il est souvent utile d'utiliser un constructeur de la super-classe, notamment pour factoriser l'initialisation. Pour ce faire, nous utilisons le constructeur `super().__init()__`, qui se confond avec le constructeur équivalent dans la classe mère. Il ne faut surtout pas faire de copier–coller du constructeur de la super-classe.

Exemple : le constructeur de la classe *Figure* est représenté listing 5.8, et le constructeur de la classe *Cercle* est représenté listing 5.9. Remarquons que le second constructeur est simplifié grâce à l'utilisation de `super().__init()__`.

Listing 5.8 – Constructeur de la super-classe

```
class Figure:
    def __init__(self, x, y):
        self._abscisse = x
        self._ordonnee = y
```

Listing 5.9 – Appel au constructeur de la super-classe

```
class Cercle(Figure):
    def __init__(self, x, y, rayon):
        # Appel au constructeur de la super-classe
        super().__init__(x, y)
        # Initialisation spécifique à la sous-classe
        self._rayon = rayon
```

5.3.4 Polymorphisme

Dans le cas des *Cercle* et *Rectangle*, qui héritent de *Figure*, cela consiste à toujours manipuler des objets *Figure*, alors même que ce sont les propriétés caractéristiques propres au cercle et au rectangle qui nous intéressent.

Le code correspondant au modèle de la figure 5.8 est représenté par le listing 5.10. Ce code contient uniquement des fragments des classes *Figure*, *Cercle* et *Rectangle*.

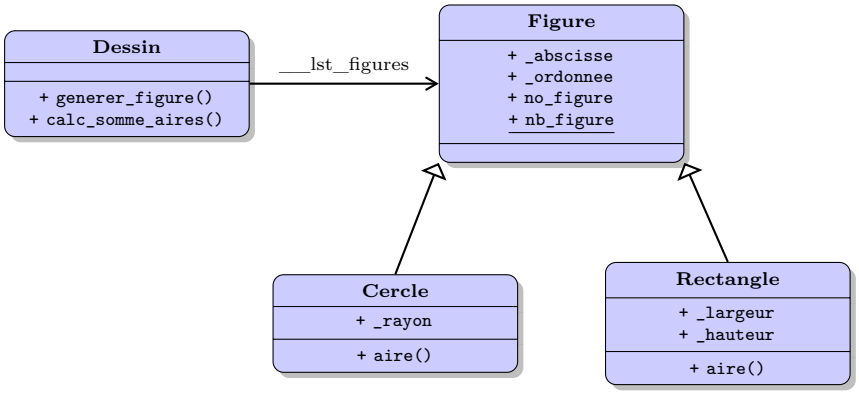


FIGURE 5.8 – Modèle de l'application

Remarque : dans le listing 5.10, les variables d'instance sont `_abscisse`, `_ordonnee`, `_rayon`, `_largeur` et `_hauteur`. Le simple tiret bas commençant leurs noms indique que les variables sont privées, et qu'il est déconseillé d'accéder directement à leur contenu. Pour des raisons de longueur de code en version papier, l'accès se fait malgré tout de façon directe, en lecture comme en écriture. Si le lecteur avait pour ambition d'écrire un code semblable, il lui faudrait soit ajouter des accesseurs (listing 5.4), soit décorer les variables d'instance (listing 5.5).

Listing 5.10 – Exemple d'utilisation du polymorphisme

```
1  import numpy as np
2
3
4  class Figure:
5      """Super-classe Figure contenant toutes les caractéristiques
6      communes aux différents types de figures.
7      """
8
9      def __init__(self, x, y):
10         self._abscisse = x
11         self._ordonnee = y
12
13     def __str__(self):
14         """Représentation de la figure sous forme de
15         chaîne de caractères.
16         """
17         return 'pos = ({0:.2f}, {1:.2f})'.\
18           format(self._abscisse, self._ordonnee)
19
20
21 class Cercle(Figure):
22     """Sous-classe de figure. Attribut supplémentaire : rayon du cercle.
23     """
24     def __init__(self, x, y, rayon):
25         # Appel au constructeur de la super-classe
26         super().__init__(x, y)
27         # Initialisation spécifique à la sous-classe
28         self._rayon = rayon
29
30     def __str__(self):
31         """Représentation du cercle sous forme de chaîne de caractère.
```

```
32          Utilise la représentation de la super-classe.
33          """
34          return 'Cercle : {0}, rayon = {1:.2f}'.\
35            format(super().__str__(), self._rayon)
36
37      def aire(self):
38          """Calcul de l'aire du cercle.
39          """
40          return np.pi * self._rayon**2
41
42
43  class Rectangle(Figure):
44      """Sous-classe de figure. Attributs supplémentaires :
45          largeur et hauteur du rectangle.
46      """
47      def __init__(self, x, y, l, h):
48          super().__init__(x, y)
49          self._largeur = l
50          self._hauteur = h
51
52      def __str__(self):
53          return 'Rectangle : {0}, dim = ({1:.2f}, {2:.2f})'.\
54            format(super().__str__(), self._largeur, self._hauteur)
55
56      def aire(self):
57          return self._largeur * self._hauteur
58
59
60  class Dessin:
61      """Un dessin est composé d'un ensemble de figures.
62      """
63      def __init__(self, nb):
64          """Initialisation du dessin avec nb figures.
65          """
66          self.__lst_figures = list()
67          for i in range(nb):
68              f = self.generer_figure()
69              self.__lst_figures.append(f)
70
71      def generer_figure(self):
72          """Génération d'une figure aléatoire.
73          """
74          # Coordonnées de la figure
75          x, y = np.random.random(2) * 100
76          # Choix du type de figure
77          choix = np.random.random()
78          # 60% de chances de choisir un cercle
79          if choix < 0.6:
80              r = np.random.random() * 30
81              f = Cercle(x, y, r)
82          else:
83              l1, l2 = np.random.random(2) * 40
84              f = Rectangle(x, y, l1, l2)
85          return f
86
87      def __str__(self):
```

```
88          """Représentation textuelle du dessin.
89          Il faut parcourir la liste de figures composant ce dessin et
90          s'appuyer sur le polymorphisme.
91          """
92          s = '{'
93          for i, f in enumerate(self.__lst_figures):
94              if i > 0:
95                  s += '\n'
96              s += '[{}]'.format(f)
97          s += '}'
98          return s
99
100     def calc_somme_aires(self):
101         """Calcul de la somme des aires des figures composant le dessin.
102         Utilise le polymorphisme.
103         """
104         s = 0
105         for f in self.__lst_figures:
106             s += f.aire()
107         return s
108
109 if __name__ == '__main__':
110     # Création d'un dessin composé de 5 figures
111     d = Dessin(10)
112     print(d)
113     # Affichage de la somme des aires des figures du dessin
114     print('Aire totale : ',d.calc_somme_aires())
```

Lors de son exécution, ce programme suit un scénario qui peut être représenté par le diagramme de la figure 5.9.

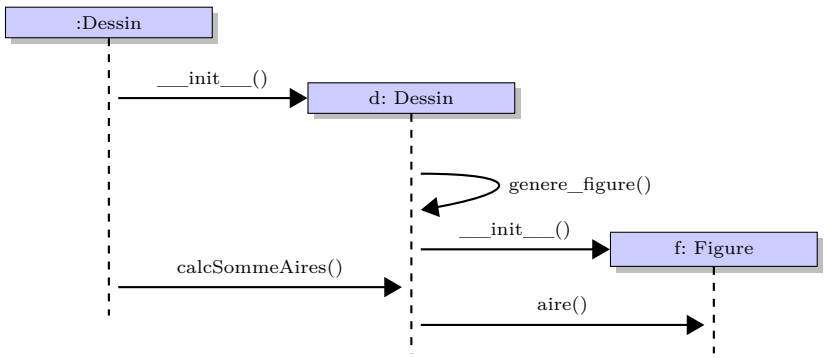


FIGURE 5.9 – Scénario d'utilisation des figures

Une collection d'objets du type général *Figure* est remplie avec des objets des types spécialisés *Cercle* et *Rectangle*. Le choix du type se fait au hasard (utilisation de *random()* dans la méthode *generer_figure()*) : on ne peut donc pas savoir avant l'exécution de quoi sera faite la collection. Voici un exemple de trace d'exécution :

```
{[Cercle : pos = (33.97, 71.57), rayon = 8.38]
[Cercle : pos = (58.02, 27.62), rayon = 10.53]
[Rectangle : pos = (8.36, 25.27), dim = (5.21, 9.46)]
[Cercle : pos = (71.84, 86.23), rayon = 29.81]
[Rectangle : pos = (22.88, 32.13), dim = (38.34, 23.88)]}
```

```
Aire totale :  4324.90521393
```

La méthode *aire()* est appelée sur chaque objet *Figure*, mais on voit qu'à l'exécution ce sont bien les méthodes spécialisées qui ont été appelées. Le lien vers la méthode spécialisée correcte s'est fait à la volée, durant l'exécution : c'est du *late-binding*.

Exemple d'utilisation de polymorphisme : ___*str*___

La méthode `__str__` est le mécanisme de Python permettant d'afficher des objets : pour afficher un objet quelconque, Python le convertit tout d'abord en chaîne de caractères grâce à la méthode *str* qui appelle `__str__()`. Cette méthode est définie dans la classe *object* et peut être redéfinie dans toutes les classes. Ainsi, si f est une figure, l'appel à `str(f)` ou à `f.__str__()` convertira f en chaîne de caractères, soit par le mécanisme par défaut de Python, soit par une méthode redéfinie par l'utilisateur.

En général, le mécanisme par défaut offre peu d'intérêt et il est conseillé de redéfinir la méthode `__str__()`. Cette méthode doit retourner une chaîne de caractères représentant l'objet.

Considérons par exemple le listing 5.11.

Listing 5.11 – Affichage de figures

```
class Figure:
    def __init__(self, x=0, y=0):
        self._abscisse = x
        self._ordonnee = y

if __name__ == '__main__':
    lst = []
    for _ in range(5):
        lst.append(Figure(np.random.rand()*100, np.random.rand()*100))

    for f in lst:
        print(f)
```

Comme aucune méthode `__str__()` n'est déclarée, Python utilisera le mécanisme par défaut de la classe *object*, et l'affichage sera le suivant :

```
<__main__.Figure object at 0x7f7e860e34e0>
<__main__.Figure object at 0x7f7e860e3518>
<__main__.Figure object at 0x7f7e860e3550>
<__main__.Figure object at 0x7f7e860e3588>
<__main__.Figure object at 0x7f7e860e35c0>
```

Le listing 5.12 représente un exemple de méthode `__str__()` de *Figure* :

Listing 5.12 – Méthode `__str__()`

```
    def __str__(self):
        return 'Pos = ({0}, {1})'.format(self._abscisse, self._ordonnee)
```

Alors, la boucle précédente du *main* donnera le résultat suivant :

```
Pos = (42.5199, 30.8643)
Pos = (63.0657, 47.0733)
Pos = (59.9565, 81.2497)
Pos = (15.9717, 62.9864)
Pos = (26.2393, 6.58151)
```

Remarques :

— la méthode *str* n'affiche rien ; elle se contente de convertir un objet en chaîne de caractères ;
— le listing 5.13 décrit différentes utilisations de *str*.

Listing 5.13 – Utilisations de `__str__`

```
# Convertir un_objet en chaîne de caractères
s1 = str(un_objet)
# Méthode équivalente mais plus lourde à écrire
s2 = un_objet.__str__()

"""Méthodes équivalentes pour afficher un_objet
"""
# Méthode préférée
print(un_objet)
# Méthodes correctes mais maladroites
print(str(un_objet))
print(un_objet.__str__())
# Utilisation d'une variable intermédiaire
s = str(un_objet)
print(s)

# Appels complètement inutiles
str(un_objet)
un_objet.__str__()
```

5.3.5 Abstractions

Classes et méthodes abstraites

Il est très courant de vouloir regrouper des types semblables en un ensemble. L'héritage est un bon moyen de le faire, notamment en vue d'utiliser le polymorphisme. Cependant il n'est pas toujours souhaitable que ce type soit instanciable. Le plus souvent, il ne présente aucun intérêt en dehors de l'unification des types spécialisés. De plus, il peut être souhaitable de forcer les sous-classes à redéfinir une méthode particulière. On dit alors que la méthode de la super-classe est abstraite.

Python propose un mécanisme de gestion des méthodes abstraites grâce au module *abc*[3]. La section 8.5 décrit ce module.

Dans notre exemple, il n'est pas nécessaire que *Figure* soit instanciée : seules ses sous-classes redéfinissant la méthode *aire* sont utilisables. On va donc la déclarer abstraite :

Listing 5.14 – Exemple de méthode abstraite

```
import numpy as np
from abc import ABCMeta, abstractmethod


class Figure(metaclass=ABCMeta):
    def __init__(self, x, y):
        self._abscisse = x
        self._ordonnee = y

    @abstractmethod
    def aire(self):
        """La méthode aire est déclarée abstraite.
```

3. Abstract Base Classes

```
        Il n'est pas possible d'instancier la classe Figure.
        Les sous-classes de Figure instanciables doivent
        redéfinir toutes les méthodes abstraites.
        """
        pass


class Cercle(Figure):
    """Sous-classe de figure. Attribut supplémentaire : rayon du cercle.
    Redéfinit la méthode abstraite aire.
    """
    def __init__(self, x, y, rayon):
        super().__init__(x, y)
        self._rayon = rayon

    def aire(self):
        """Calcul de l'aire du cercle.
        """
        return np.pi * self._rayon**2


class FigureOrientee(Figure):
    """Sous-classe de figure, ajoutant un paramètre d'orientation.
    Cette classe ne redéfinit pas aire ; elle est donc abstraite.
    """
    def __init__(self, x, y, angle):
        super().__init__(x, y)
        self._angle = angle

    def rotation(self, angle):
        self._angle += angle
```

Dans ce cas, il est interdit d'instancier les classes *Figure* et *FigureOrientee* ; par contre, la classe *Cercle* redéfinit la méthode *aire*, et elle est donc instanciable. Exemple :

```
>>> f = Figure(0, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Figure with abstract methods
  aire
>>> f = FigureOrientee(0, 0, 1.57)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class FigureOrientee with abstract
  methods aire
>>> f = Cercle(0, 0, 4)
```

5.3.6 Variables et méthodes de classe

Tout ce dont nous avons discuté jusqu'à maintenant, en terme de variables et de méthodes, visait à spécifier le contenu et le comportement des instances de la classe.

Il est cependant possible de créer des variables et des méthodes qui ne sont pas des propriétés de l'instance mais de la classe. Ainsi, les variables de classe sont des propriétés partagées par toutes les instances.

En Python, il faut utiliser le décorateur `@classmethod` pour créer une méthode de classe. Le premier argument de la méthode est alors la classe (*cls*) et pas l'instance (*self*).

Le listing 5.15 représente un exemple d'utilisation de variable et de méthode de classe.

Listing 5.15 – Utilisation de variable de classe

```
class A:
    # Variable de classe : partagée par toutes les instances
    variable_de_classe = 0

    def __str__(self):
        return str(self.variable_de_classe)

    @classmethod
    def set_variable(cls, val):
        """Grâce au décorateur @classmethod il s'agit ici
        d'une méthode de classe. Le premier paramètre est
        donc la classe (cls) et pas l'objet (self).
        """
        cls.variable_de_classe = val

if __name__ == '__main__':
    x = A()
    y = A()
    # x et y partagent la même variable de classe
    print(x, y)
    x.set_variable(42)
    # Autre possibilité pour modifier la variable :
    # A.variable_de_classe = 42
    print(x, y)
```

Exemple d'utilisation de variables et méthodes de classe : numérotation unique

Un cas dans lequel l'utilisation de variables et méthodes de classe peut être justifiée est la mise en œuvre d'un compteur d'objets. Il est impossible de compter le nombre d'objets créés de type *Figure* en utilisant des variables *d'instance* de la classe *Figure*. Par contre, c'est réalisable avec des variables de classe. Ce compteur peut servir à attribuer un numéro unique à chaque objet. Le listing 5.16 représente une telle utilisation.

Listing 5.16 – Variable de classe : compteur de figures

```
import numpy as np


class Figure:
    # Variable de classe : nombre de figures existantes
    nb_figure = 0

    def __init__(self, x, y):
        """Création d'une figure. Initialise automatiquement le numéro
        de figure et incrémente le nombre de figures existantes.
        """
        self._abscisse = x
        self._ordonnee = y
        self.no_figure = Figure.nb_figure
        Figure.nb_figure += 1

    def __str__(self):
        return '#%d pos = (%g, %g)' % (self.no_figure,
```

```
                                        self._abscisse,
                                        self._ordonnee)


if __name__ == '__main__':
    l = list()  # liste de figures remplie aléatoirement
    for _ in range(5):
        l.append(Figure(np.random.rand()*100, np.random.rand()*100))

    # Le nombre de figures est identique pour toutes les instances
    print(l[0].nb_figure)
    for f in l:
        print(f)
```

Le résultat de l'exécution du listing 5.16 est représenté ci-dessous :

```
5
#0 pos = (73.9536, 85.3991)
#1 pos = (7.19658, 75.713)
#2 pos = (87.4179, 51.1968)
#3 pos = (88.2175, 22.843)
#4 pos = (75.342, 1.41665)
```

Noter que la variable de classe est initialisée lors de sa déclaration et qu'il n'y a pas d'autre possibilité : il n'est pas possible de l'initialiser dans le constructeur.

5.4 Conception de programme orienté objet

La principale difficulté lors de la conception d'un programme orienté objet réside dans le choix de la décomposition en classes. Il existe en effet un très grand nombre de possibilités, plus ou moins faciles à mettre en œuvre, et surtout plus ou moins faciles à faire évoluer.

Certaines situations se retrouvant très souvent, des solutions types permettant d'y répondre sont apparues. Ces solutions sont nommées motifs de conception ou *design patterns*. Leur étude sera abordée lors du cours de modélisation de deuxième année. Toutefois, l'ouvrage [Fre+05] est une très bonne introduction aux *design patterns*[4]. Il en recense un grand nombre et décrit dans quels cas les appliquer. Pour plus d'informations, consulter [Gam+95], également appelé « le GOF[5] » qui reste la référence en terme de *design patterns*.

La suite de cette section décrit quelques *patterns* de manière très sommaire.

5.4.1 Pattern observateur

Le motif de conception observateur est utile dans le cas où des données sont accessibles (en lecture et en écriture) par plusieurs moyens.

Le cas classique est un ensemble de données accessible par plusieurs interfaces graphiques (une représentation sous forme de tableau, une sous forme de camembert, une sous forme d'histogramme par exemple). Les données peuvent être modifiées depuis toutes les interfaces, et une modification doit être répercutée dans toutes les interfaces.

Ce motif est particulièrement utile lorsque :

— une modification dans un objet doit entraîner des modifications dans d'autres objets dont le nombre n'est pas connu a priori ;

— un objet doit être capable de signaler une modification à d'autres objets sans faire d'hypothèse sur lesdits objets.

4. Bien que le langage d'application soit Java et pas Python.
5. Gang Of Four

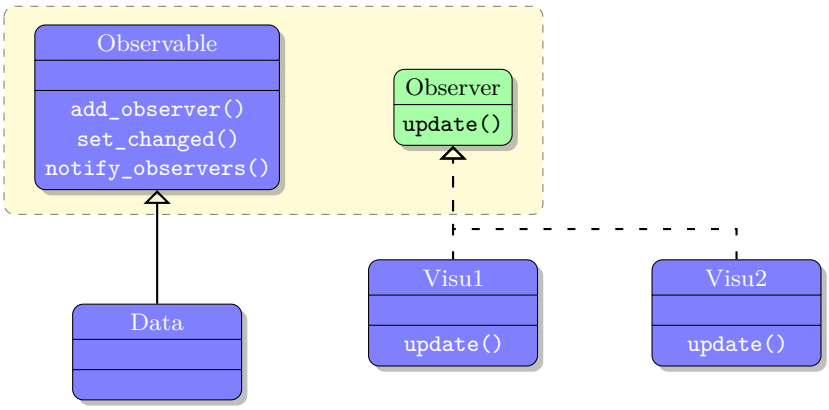


FIGURE 5.10 – Diagramme de classe du pattern observateur

Le motif n'existe pas par défaut en Python, mais il est très simple à mette en œuvre. Le listing 5.17 est un exemple d'implantation de ce motif.

Listing 5.17 – Motif observateur

```
from abc import ABCMeta, abstractmethod


class Observer(metaclass=ABCMeta):
    """Classe dont tous les visualisateurs doivent hériter.
    Les sous-classes doivent obligatoirement reféfinir la méthode update.
    """
    @abstractmethod
    def update(self, observable, arg):
        """Méthode appelée lorsqu'un objet observé est modifié.
        """
        pass


class Observable:
    """Classe dont tous les objets observés doivent hériter.

    Attributes
    ----------
    obs : liste
        Variables de type Observer liées à l'instance.
    changed : bool
        Status : est-ce que les données observées ont été modifiées ?
    """
    def __init__(self):
        self.obs = []
        self.changed = False

    def add_observer(self, observer):
        if observer not in self.obs:
            self.obs.append(observer)

    def delete_observer(self, observer):
        self.obs.remove(observer)

```

```
36      def notify_observers(self, arg=None):
37          if self.changed:
38              for observer in self.obs:
39                  observer.update(self, arg)
40              self.changed = False
41
42      def delete_observers(self):
43          self.obs = []
44
45      def set_changed(self):
46          self.changed = True
47
48      def clear_changed(self):
49          self.changed = False
50
51      def has_changed(self):
52          return self.changed
53
54      def count_observers(self):
55          return len(self.obs)
```

L'utilisation du motif observateur est très simple : la classe contenant les données (*Data*) doit hériter de *Observable*, et les classes représentant les données (*Visu1* et *Visu2*) doivent hériter de la classe *Observer*. Le listing 5.18 représente un exemple d'utilisation de ce motif.

Listing 5.18 – Utilisation du motif observateur

```
1  from ch4_pattern_observer import *
2
3
4  class Data(Observable):
5      """Les données sont observables. Lors de l'initialisation,
6      on leur attache les observateurs.
7      """
8      def __init__(self, val=0):
9          super().__init__()
10         self.value = val
11         self.add_observer(Visu1())
12         self.add_observer(Visu2())
13
14     @property
15     def value(self):
16         return self.__value
17
18     @value.setter
19     def value(self, val):
20         """Lors d'une modification des données, on met à jour l'affichage
21         via les obsevateurs.
22         """
23         self.__value = val
24         super().set_changed()
25         super().notify_observers()
26
27
28 class Visu1(Observer):
29     def update(self, observable, arg):
30         print(observable.value)
31
```

```
32
33 class Visu2(Observer):
34     def update(self, observable, arg):
35         print('Scoop ! value passe à {}'.format(observable.value))
36
37
38 if __name__ == "__main__":
39     x = Data()
40     # La modification de l'attribut value déclenche l'appel à update.
41     x.value = 54
42     x.value = 42
```

La trace de l'exécution du listing 5.18 donne :

```
54
Scoop ! value passe à 54
42
Scoop ! value passe à 42
```

Bilan du motif observateur :

— ce motif permet de découpler très fortement les données de leur affichage (les données ne savent rien des observateurs en dehors du fait qu'ils implantent l'interface *Observer*) ;
— les données mettent à jour les observateurs via une interface commune ;
— ce motif permet de réaliser très simplement une relation *1-n* entre les données et leur visualisation.

5.4.2 Pattern stratégie

Le motif de conception stratégie est utile pour des situations où il est nécessaire de permuter dynamiquement les algorithmes utilisés dans une application. Le motif stratégie est prévu pour fournir des moyens de définir une famille d'algorithmes, encapsuler chacun comme objet, et les rendre interchangeables. Le motif stratégie laisse les algorithmes changer indépendamment des clients qui les emploient.

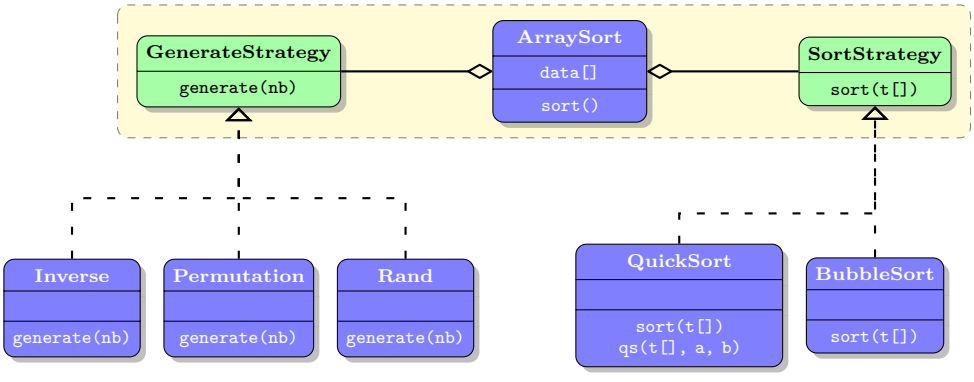


FIGURE 5.11 – Diagramme de classe du pattern stratégie

Ce motif est utile quand :

— plusieurs classes ne diffèrent que par leur comportement ;
— plusieurs variantes d'un algorithme sont nécessaires (ex. figure 5.11) ;
— une classe possède plusieurs méthodes, et chaque méthode a plusieurs variantes ;

— une classe définit plusieurs comportements qui apparaissent comme des conditions multiples dans les méthodes ; on peut alors remplacer les différentes branches conditionnelles par différentes classes de stratégie.

La figure 5.11 décrit l'application de ce motif à des algorithmes de tris. Ici, la classe *ArraySort* permet de trier un tableau. Pour ce faire, elle utilise une stratégie parmi l'ensemble des tris disponibles (une classe implantant l'interface *SortStrategy* : *BubbleSort*, *ShellSort*, ou *QuickSort*). La génération du tableau à trier se fait par la classe *GenerateStrategy* et ses sous–classes : *Inverse*, *Permutation* et *Rand*.

Le listing 5.19 contient le code Python correspondant à cet exemple.

Listing 5.19 – Motif stratégie

```
import numpy as np
from abc import ABCMeta, abstractmethod

class ArraySort:
    """Génération de tableaux aléatoires et tri
    """
    def __init__(self, sort, gener, nb_elem=10):
        # Génère un tableau aléatoire grâce à la stratégie de génération
        self.__data = gener.generate(nb_elem)
        self.__sort = sort

    def sort(self):
        # Trie le tableau en utilisant la stratégie de tri
        self.__sort.sort(self.__data)

    def __str__(self):
        return str(self.__data)

class GenerateStrategy(metaclass=ABCMeta):
    """Classe abstraite pour les générations de tableau.
    """
    @abstractmethod
    def generate(self, nb):
        """Méthode abstraite : doit être redéfinie dans les sous-classes.
        """
        pass

class Permutation(GenerateStrategy):
    """Génération d'une permutation de [0,n[.
    """
    def generate(self, nb):
        data = np.array(list(range(nb)))
        np.random.shuffle(data)
        return data

class Inverse(GenerateStrategy):
    """Génération d'une liste inversée.
    """
    def generate(self, nb):
        lst = list(range(nb))
        data = np.array(lst[::-1])
        return data

class Rand(GenerateStrategy):
    """Génération d'un tableau aléatoire.
```

```
46      """
47      def generate(self, nb):
48          return np.random.randint(0, nb, nb)
49
50  class SortStrategy(metaclass=ABCMeta):
51      """Classe abstraite pour les tris définis.
52      """
53      @abstractmethod
54      def sort(self, tab):
55          pass
56
57  class BubbleSort(SortStrategy):
58      """Stratégie de tri bulle.
59      """
60      def sort(self, tab):
61          """Tri du tableau tab par la méthode du tri bulles.
62          """
63          for i in range(len(tab), 0, -1):
64              modif = False
65              for j in range(1, i):
66                  if tab[j - 1] > tab[j]:
67                      tab[j], tab[j - 1] = tab[j - 1], tab[j]
68                      modif = True
69              if not modif:
70                  return
71
72  class QuickSort(SortStrategy):
73      """Stratégie de tri par segmentation.
74      """
75      def sort(self, tab):
76          return self.tri_seg_rec(tab, 0, len(tab)-1)
77
78      def tri_seg_rec(self, tab, a, b):
79          """Fonction récursive de tri par segmentation.
80          """
81          x, y = a, b
82          if x < y:
83              # Clé de tri : tout ce qui est inférieur est placé à gauche
84              # tout ce qui est supérieur à droite.
85              cle = tab[(x + y) // 2]
86              while x <= y:
87                  # Se décaler tant que tab[x] est du bon côté de la clé
88                  while x < b and tab[x] < cle:
89                      x += 1
90                  # Se décaler tant que tab[y] est du bon côté de la clé
91                  while y > a and tab[y] > cle:
92                      y -= 1
93                  if x <= y:
94                      # Ici tab[x] et tab[y] sont du mauvais côté de la clé
95                      # Il faut donc les permuter
96                      tab[x], tab[y] = tab[y], tab[x]
97                      x += 1
98                      y -= 1
99              # Appels récursifs sur les deux sous-tableaux
100             if x < b:
101                 self.tri_seg_rec(tab, x, b)
```

```
            if a < y:
                self.tri_seg_rec(tab, a, y)


if __name__ == "__main__":
    # Liste des algorithmes de tri disponibles
    sorting_alg = [BubbleSort(), QuickSort()]
    # Liste des générateurs de tableaux disponibles
    gener_alg = [Permutation(), Rand(), Inverse()]
    for alg in sorting_alg:
        for gen in gener_alg:
            # Pour chaque tri et chaque générateur, créer la stratégie
            strategy = ArraySort(sort=alg, gener=gen)
            print(alg, gen)
            print(strategy)
            strategy.sort()
            print(strategy)
```

Variante du pattern stratégie : le langage Python offre la possibilité d'affecter des fonctions à des variables. Ceci permet de réécrire le motif stratégie comme indiqué dans l'exemple 5.20.

Listing 5.20 – Motif stratégie, variante

```
import numpy as np


class ArraySort:
    """Génération de tableaux aléatoires et tri
    """
    def __init__(self, sort, gener, nb_elem=10):
        # Génère un tableau aléatoire grâce à la stratégie de génération
        self.__data = gener(nb_elem)
        self.__sort = sort

    def sort(self):
        # Trie le tableau en utilisant la stratégie de tri
        self.__sort(self.__data)

    def __str__(self):
        return str(self.__data)


def generate_permutation(nb):
    """Génération d'une permutation de [0,n[.
    """
    data = np.array(list(range(nb)))
    np.random.shuffle(data)
    return data


def generate_inverse(nb):
    """Génération d'une liste inversée.
    """
    lst = list(range(nb))
    data = np.array(lst[::-1])
    return data
```

```
34
35
36  def generate_rand(nb):
37      """Génération d'un tableau aléatoire.
38      """
39      return np.random.randint(0, nb, nb)
40
41
42  def sort_bubble(tab):
43      """Tri du tableau tab par la méthode du tri bulles.
44      """
45      for i in range(len(tab), 0, -1):
46          modif = False
47          for j in range(1, i):
48              if tab[j - 1] > tab[j]:
49                  tab[j], tab[j - 1] = tab[j - 1], tab[j]
50                  modif = True
51          if not modif:
52              return
53
54
55  def sort_segmentation(tab):
56      """Stratégie de tri par segmentation.
57      """
58      return tri_seg_rec(tab, 0, len(tab)-1)
59
60
61  def tri_seg_rec(tab, a, b):
62      """Fonction récursive de tri par segmentation.
63      """
64      x, y = a, b
65      if x < y:
66          # Clé de tri : tout ce qui est inférieur est placé à gauche
67          # tout ce qui est supérieur à droite.
68          cle = tab[(x + y) // 2]
69          while x <= y:
70              # Se décaler tant que tab[x] est du bon côté de la clé
71              while x < b and tab[x] < cle:
72                  x += 1
73              # Se décaler tant que tab[y] est du bon côté de la clé
74              while y > a and tab[y] > cle:
75                  y -= 1
76              if x <= y:
77                  # Ici tab[x] et tab[y] sont du mauvais côté de la clé
78                  # Il faut donc les permuter
79                  tab[x], tab[y] = tab[y], tab[x]
80                  x += 1
81                  y -= 1
82          # Appels récursifs sur les deux sous-tableaux
83          if x < b:
84              tri_seg_rec(tab, x, b)
85          if a < y:
86              tri_seg_rec(tab, a, y)
87
88
89  if __name__ == "__main__":
```

```
90      # Liste des algorithmes de tri disponibles
91      sorting_alg = [sort_bubble, sort_segmentation]
92      # Liste des générateurs de tableaux disponibles
93      gener_alg = [generate_permutation, generate_inverse, generate_rand]
94      for alg in sorting_alg:
95          for gen in gener_alg:
96              # Pour chaque tri et chaque générateur, créer la stratégie
97              # Remarque : on affecte ici des fonctions
98              strategy = ArraySort(sort=alg, gener=gen)
99              print(alg, gen)
100             print(strategy)
101             strategy.sort()
102             print(strategy)
```

Quelques avantages du pattern stratégie :

— des familles d'algorithmes apparaissent, ce qui peut favoriser la factorisation de code ;
— ce pattern offre une alternative à l'héritage. Il serait en effet possible de sous-classer directement la classe *ArraySort*, mais il serait alors impossible de changer d'algorithme dynamiquement ;
— il permet de définir des applications utilisant plusieurs stratégies (par exemple une stratégie de tri et une stratégie de génération) ;
— il offre également une alternative élégante à des comportements codés dans différentes méthodes avec des tests (*if*) ;
— il permet de choisir simplement différentes implantations pour le même comportement .

... et quelques inconvénients :

— il est nécessaire de connaître les différentes stratégies pour en choisir une (ceci est particulièrement gênant si quelqu'un doit reprendre votre code) ;
— les prototypes des méthodes des différentes stratégies doivent être identiques, ce qui oblige à passer des paramètres inutiles à certaines stratégies ;
— l'utilisation du pattern stratégie augmente le nombre d'objets créés par l'application.

Listing 5.21 – Motif stratégie, utilisation de l'affectation dynamique

```
1   import numpy as np
2   # strategies.py contient les 5 stratégies de l'exemple précédent
3   from strategies import *
4
5
6   class ArraySort:
7       """Génération de tableaux aléatoires et tri
8       """
9       def __init__(self, sort, gener, nb_elem=10):
10          # Génère un tableau aléatoire grâce à la stratégie de génération
11          self._nb_elem  = nb_elem
12          if gener:
13              self.__data = gener(self._nb_elem)
14          self.__sort = sort
15
16      def set_gener(self, gener):
17          # Génère un nouveau tableau avec la fonction passée en argument
18          self.__data = gener(self._nb_elem)
19
20      def set_sort(self, algo_tri):
21          # met à jour la stratégie de tri
22          self.__sort = algo_tri
23
```

```
24      def sort(self):
25          # Trie le tableau en utilisant la stratégie de tri
26          self.__sort(self.__data)
27
28      def __str__(self):
29          return str(self.__data)
30
31
32  if __name__ == "__main__":
33      # Liste des algorithmes de tri disponibles
34      sorting_alg = [sort_bubble, sort_segmentation]
35      # Liste des générateurs de tableaux disponibles
36      gener_alg = [generate_permutation, generate_inverse, generate_rand]
37      # On crée un ArraySort non fonctionnel : il faudra lui affecter
38      # deux stratégies avant de faire travailler ses méthodes.
39      strategy = ArraySort(sort=None, gener=None)
40      for alg in sorting_alg:
41          # Affecter la méthode de tri
42          strategy.set_sort(alg)
43          for gen in gener_alg:
44              # Pour chaque tri et chaque générateur, créer un tableau et
                  le trier
45              # Affecter la méthode de génération de tableau et l'appliquer
46              strategy.set_gener(gen)
47              print(alg, gen)
48              print(strategy)
49              strategy.sort()
50              print(strategy)
```

5.4.3 Pattern état

Le motif de conception état est utile lorsqu'il faut représenter un objet possédant un comportement (défini par un automate par exemple). L'idée générale consiste à dire que l'objet possède un état (une classe abstraite qui se décline en plusieurs états concrets) et que chaque état réagit de manière spécifique à un évènement.

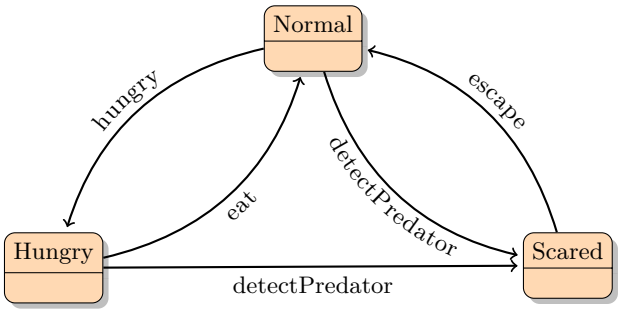


FIGURE 5.12 – Diagramme d'état du pattern état

Dans notre cas, nous allons simuler le comportement d'un animal. Ce comportement est représenté par l'automate figure 5.12 : par défaut, l'animal est dans un état *Normal* ; s'il aperçoit un prédateur, il passe dans l'état *Scared*, et s'il a faim, il passe dans un état *Hungry*. Depuis l'état *Hungry*, il faut apercevoir un prédateur et passer dans l'état *Scared*, ou manger et revenir dans l'état *Normal*. Il ne sort de l'état *Scared* que s'il a échappé au prédateur, et il revient alors à l'état

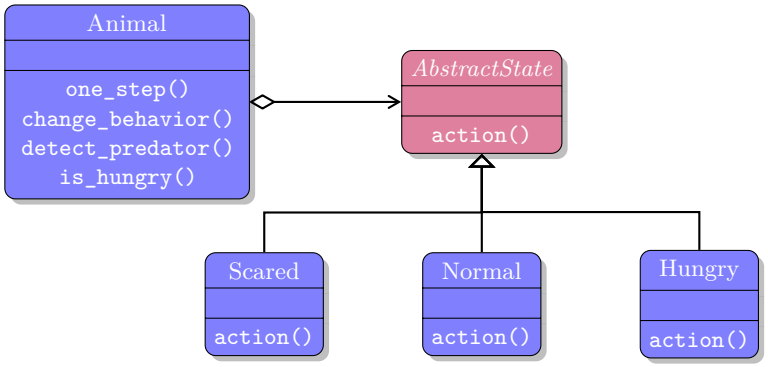


FIGURE 5.13 – Diagramme de classe du pattern état

Normal. Le diagramme de classe correspondant est représenté par la figure 5.13. On remarque que l'animal possède un état abstrait (*AbstractState*) et 3 états concrets (*Normal*, *Hungry* et *Scared*). Le code correspondant est représenté dans le listing 5.22.

Listing 5.22 – Motif état

```
import numpy as np
from abc import ABCMeta, abstractmethod

class Animal:
    """Animal possédant un comportement décrit par un automate.
    """
    def __init__(self):
        # Génère un tableau aléatoire grâce à la stratégie de génération
        self.__behavior = Normal()

    def detect_predator(self):
        """Simulation statistique : 20% de chances de
        détecter un prédateur.
        """
        return np.random.rand() < 0.2

    def is_hungry(self):
        """Simulation statistique : 30% de chances de
        d'avoir faim.
        """
        return np.random.rand() < 0.3

    def eat(self):
        pass

    def change_behavior(self, new_b):
        self.__behavior = new_b

    def one_step(self):
        """Exécute un tour de simulation.
        """
        self.__behavior.action(self)
        print(self.__behavior)

class AbstractState(metaclass=ABCMeta):
```

```
36      """Classe abstraite pour les comportements.
37      """
38      @abstractmethod
39      def action(self, animal):
40          """Méthode abstraite : doit être redéfinie dans les sous-classes.
41          """
42          pass
43
44  class Normal(AbstractState):
45      """Définition du comportement standard d'un animal.
46      """
47      def action(self, animal):
48          self.move(animal)
49          if animal.detect_predator():
50              # Prédateur détecté ; nouveau comportement : fuite
51              animal.change_behavior(Scared())
52          elif animal.is_hungry():
53              # Animal affamé : chercher de la nourriture
54              animal.change_behavior(Hungry())
55
56      def move(self, animal):
57          pass  # Déplacement de l'animal...
58
59  class Hungry(AbstractState):
60      """Définition du comportement d'un animal affamé.
61      """
62      def action(self, animal):
63          # Chercher de la nourriture
64          trouve = self.move(animal)
65          if animal.detect_predator():
66              # Prédateur détecté ; nouveau comportement : fuite
67              animal.change_behavior(Scared())
68          elif trouve:
69              # Nourriture trouvée : manger
70              animal.eat()
71              # Puis revenir dans l'état standard
72              animal.change_behavior(Normal())
73
74      def move(self, animal):
75          # 40% de chances de trouver de la nourriture
76          return np.random.rand() < 0.4
77
78  class Scared(AbstractState):
79      """Définition du comportement d'un animal effrayé.
80      """
81      def action(self, animal):
82          # Chercher à s'enfuir
83          escape = self.move(animal)
84          if escape:
85              # Fuite réussie : revenir à l'état standard
86              animal.change_behavior(Normal())
87
88      def move(self, animal):
89          # 50% de chances de s'enfuir
90          return np.random.rand() < 0.5
91
```

```
92 if __name__ == "__main__":
93     a = Animal()
94     for _ in range(10):
95         a.one_step()
```

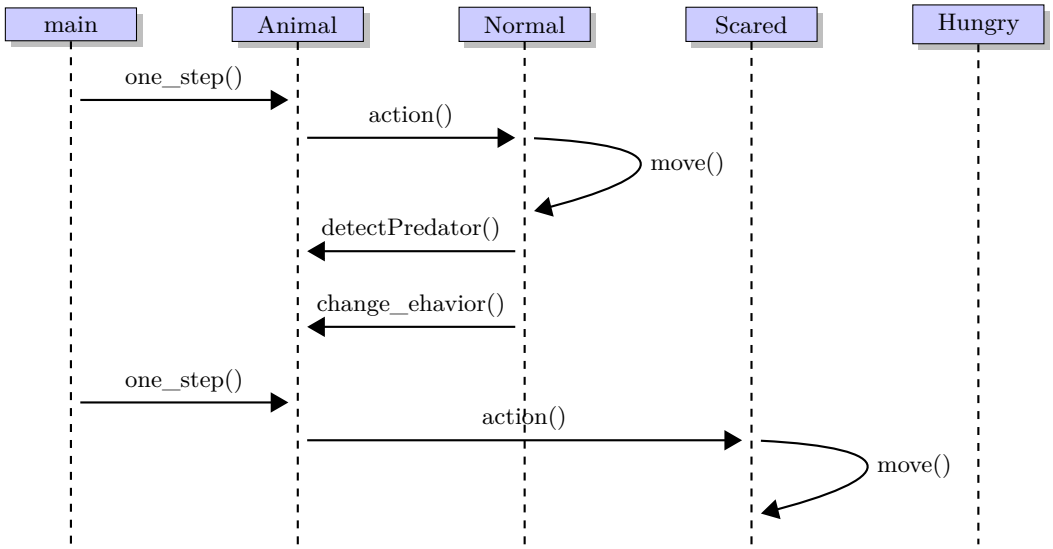


FIGURE 5.14 – Diagramme de séquence du pattern état

Un scénario d'application du pattern état est représenté figure 5.14. Ce diagramme permet de voir l'enchaînement des méthodes au travers des différents objets (par exemple la méthode *one_step()* de la class *Animal* appelle *action()* de *Normal*).

Remarque : ce diagramme ne représente qu'un exemple de scénario possible.

Avantages du pattern état :

- il permet de rassembler tout le code ayant trait à un état ; ajouter un nouvel état se fait alors très simplement ;
- les transitions entre les états sont décrites explicitement, ce qui simplifie la relecture du code.

Inconvénients :

- ce pattern nécessite la création de nombreuses classes supplémentaires (une par état).

Remarque sur l'implantation : dans l'exemple fourni, un nouvel objet état est créé lors de chaque transition. Une solution élégante pour pallier ce problème consiste à coupler le pattern état avec le pattern singleton.

. . . et sur sa couverture on peut lire en larges lettres amicales la mention : PAS DE PANIQUE *!*

Douglas Adams

6 Algorithmique 2

Sommaire

AYANT connaissance des principes fondamentaux de l'objet et des bases de l'algorithmique, il est maintenant possible d'étudier de nouvelles structures algorithmiques fondamentales : les structures dynamiques. Ces structures sont utilisées pour stocker des ensembles d'éléments dont la taille n'est pas connue *a priori*.

6.1 Listes

Définition 6.1 (Liste chaînée)**.** *Une liste chaînée est une structure algorithmique* dynamique *(dont la taille peut évoluer au cours du temps) qui permet un accès aux éléments de manière* séquentielle *(il faut passer par le premier élément pour accéder au second).*

Une liste peut prendre plusieurs formes : elle peut être simplement chaînée, doublement chaînée, triée ou non, circulaire ou non, comporter des cycles ... Dans une liste simplement chaînée, il est possible d'accéder à un élément à partir de son prédécesseur. Dans une liste doublement chaînée, il est possible d'accéder à un élément depuis son prédécesseur ou son successeur.

Des listes simplement chaînées, doublement chaînées et circulaires sont représentées figure 6.1.

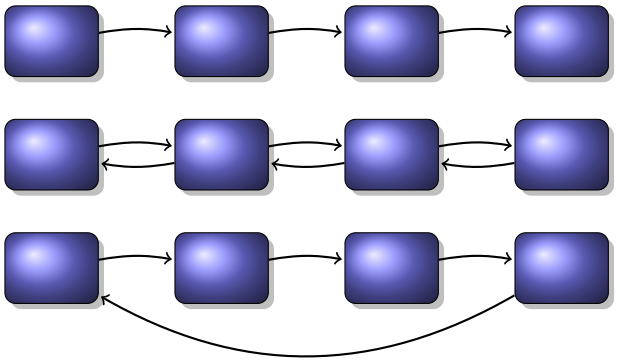


FIGURE 6.1 – Différents types de listes

Une liste est constituée d'éléments ou nœuds chaînés entre eux. Chaque nœud doit connaître son successeur (et son prédécesseur si la liste est doublement chaînée). Une liste contient un nœud de tête (également un nœud de fin si elle est doublement chaînée) et des méthodes permettant d'insérer, de rechercher, de supprimer un élément.

6.1.1 Création des nœuds

La classe définissant les nœuds de la liste doit contenir une ou plusieurs valeurs et surtout un nœud permettant le chaînage. La classe *Node* correspondante est définie ainsi :

Listing 6.1 – Classe Node (liste)

```
1  class Node:
2      """Élement de base de la liste chaînée.
3
4      Attributes
5      ----------
6      val
7          Valeur du noeud
8      next : Node
9          Noeud suivant
10     """
11
12     def __init__(self, val=0):
13         """Creation d'un noeud ; le noeud suivant est fixé à None.
14         """
15         self.val = val
16         self.next = None
```

Un exemple de classes permettant de définir une liste chaînée est représenté figure 6.2.

6.1.2 Parcours d'une liste

Il s'agit de l'opération la plus simple à effectuer sur une liste. Le principe est de partir de la tête de liste et d'avancer tant que l'élément courant n'est pas la fin de liste (*None*). Ce principe

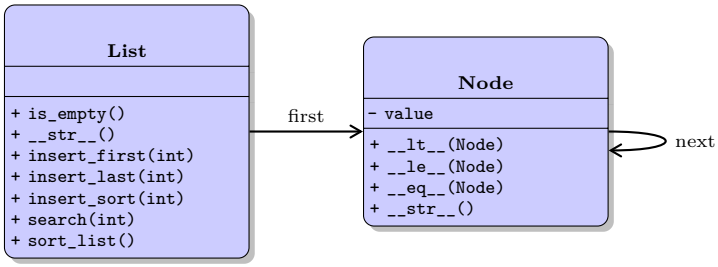


FIGURE 6.2 – Classe liste

est représenté par l'algorithme 6.1.

Algorithme 6.1 : Algorithme de parcours de liste

Entrées : premier : noeud `# tête de liste`
Données : n : noeud `# noeud courant`
n ← premier;
tant que $n \neq None$ **faire**
  n ← n.suivant;
fin

Il existe plusieurs manières de traduire cet algorithme en Python : avec une boucle *while* ou une méthode récursive.

On suppose dans le programme Python que le premier élément de la liste est une variable de type *Node* nommée `__first`. Le parcours avec un *while* peut s'écrire :

Listing 6.2 – Parcours d'une liste chaînée

```
def parcours(self):
    # On part obligatoirement du premier noeud de la liste
    n = self.__first
    # Tant qu'on n'est pas arrivé à None on peut continuer
    while n is not None:
        # Action à exécuter ici

        # Passer au noeud suivant
        n = n.next
```

6.1.3 Recherche d'un élément dans une liste

L'algorithme de recherche d'un élément dans une liste s'appuie sur un parcours simple. Il s'agit de partir de l'élément de tête et d'avancer jusqu'à arriver sur *None* ou sur l'élément recherché. Si le parcours finit sur *None*, alors l'élément n'est pas dans la liste et l'algorithme retourne *None*, sinon l'algorithme retourne le premier élément ayant la valeur recherchée. Cette méthode est décrite par l'algorithme 6.2.

Étude de l'algorithme

Étudions la complexité de cet algorithme. Pour cela, remarquons que le pire des cas est celui où l'élément n'est pas dans la liste. Il est alors nécessaire de parcourir toute la liste. La complexité de la recherche d'un élément dans une liste est donc de $\Theta(n)$ dans le pire des cas. On admettra qu'il en est de même en moyenne.

Algorithme 6.2 : Algorithme de recherche d'élément dans une liste

Entrées : int val
Données : debut : noeud `# tête de liste`
Données : n : noeud `# noeud courant`
n ← debut;
tant que $n \neq None$ *et* $n.valeur \neq val$ **faire**
  n ← n.suivant;
fin
retourner n;

Ceci fait de la représentation par liste une manière peu efficace de stocker (et surtout de retrouver) des données.

6.1.4 Ajout d'un élément dans une liste

Il existe plusieurs possibilités pour ajouter un élément dans une liste. Les trois principales sont :
— insertion en début de liste ;
— insertion en fin de liste ;
— insertion triée [1].
Chaque méthode possède des avantages et des inconvénients.

Ajout en début de liste

Cette méthode possède l'avantage d'être très simple à mettre en œuvre, très rapide, mais elle ne respecte pas l'ordre des données. Le principe est le suivant :
— soit un nouveau nœud *nouv* (à ajouter dans la liste) ;
— rattacher le premier nœud de la liste à *nouv* ;
— définir *nouv* en tant que nouvelle tête de liste.
Ces étapes sont représentées figure 6.3 et par l'algorithme 6.3.

Algorithme 6.3 : Algorithme d'insertion en début de liste

Entrées : nouv : noeud à ajouter
Données : debut : noeud `# tête de liste`
nouv.suivant ← debut;
debut ← nouv;

Ceci se traduit en python de la manière suivante :

Listing 6.3 – Insertion en début de liste

```
1 def insert_first(self, val):
2     """Insère une valeur en début de liste.
3     Commence par créer un noeud.
4     """
5     n = Node(val)
6     n.next = self.__first
7     self.__first = n
```

1. Suppose que les éléments sont munis d'une relation d'ordre.

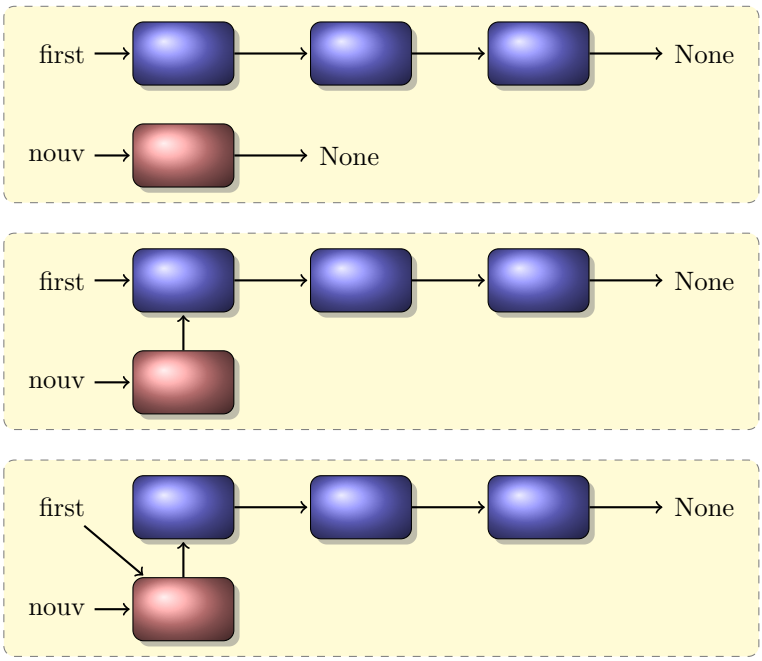


FIGURE 6.3 – Insertion en début de liste

Étude de l'algorithme

Cet algorithme est peu coûteux : quelque soit le cas, l'insertion se fait en un nombre constant [2] d'opérations. Sa complexité est donc de $\Theta(1)$.

Ajout en fin de liste

Le principal avantage de cette méthode d'insertion est de conserver l'ordre des éléments : si a est entré dans la liste avant b, alors il sera positionné avant dans la liste. L'inconvénient de cette méthode est sa complexité algorithmique.

Pour insérer un élément (*nouv*) à la fin d'une liste simplement chaînée, il est nécessaire de se placer sur le dernier élément, puis de chaîner *nouv* avec cet élément. L'algorithme d'insertion d'un élément en fin de liste est donc divisé en deux parties :

1. se déplacer jusqu'au dernier élément ;
2. chaîner le nouvel élément avec la fin de la liste.

Cette procédure est représentée par l'algorithme 6.4 et par la figure 6.4.

Remarques :
- avancer jusqu'au dernier élément se traduit par avancer jusqu'à l'élément dont le suivant est *None* ;
- si la liste est vide, un traitement particulier est nécessaire.

Étude de l'algorithme

Dans tous les cas, le nombre d'opérations nécessaire pour se placer sur le dernier élément de la liste est de n, et le nombre d'opérations pour réaliser le chaînage est de 1. La complexité de

2. Par rapport à la taille de la liste

Algorithme 6.4 : Algorithme d'insertion en fin de liste

```
Entrées : nouv : noeud à ajouter
Données : debut : noeud                                  # tête de liste
Données :  n : noeud                                     # noeud courant
si liste = ∅                                  # cas particulier liste vide
 alors
    debut ← nouv ;
sinon
    # cas général : se placer sur le dernier élément
    n ← debut;
    tant que n.suivant ≠ None faire
        n ← n.suivant;
    fin
    # on est maintenant sur le dernier élément
    n.suivant ← nouv;
fin
```

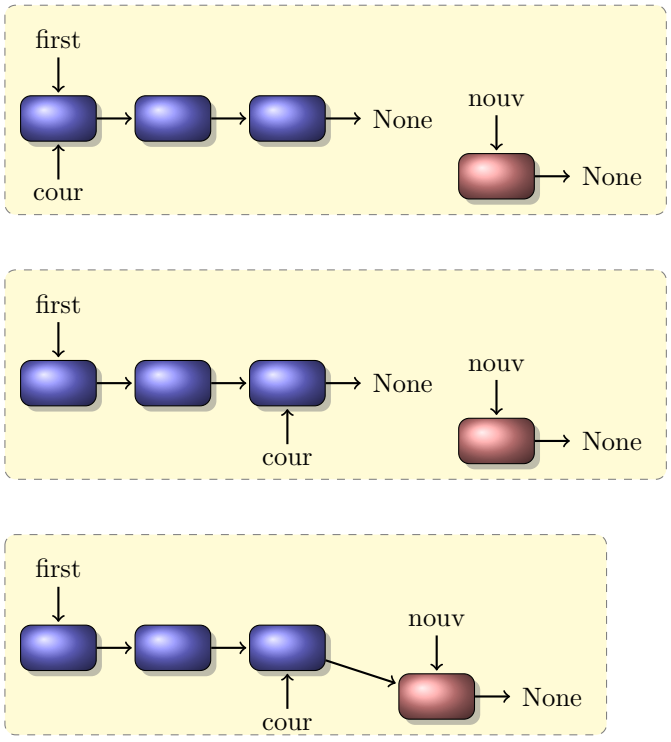


FIGURE 6.4 – Insertion en fin de liste

cet algorithme est donc de $\Theta(n)$. On constate que l'insertion d'un élément en fin de liste est peu efficace.

Ajout dans une liste triée

L'insertion dans une liste triée est semblable à l'insertion en fin de liste. Considérons le cas de l'insertion dans une liste triée par ordre croissant. Le principe est de se positionner au bon endroit dans la liste puis d'insérer l'élément.

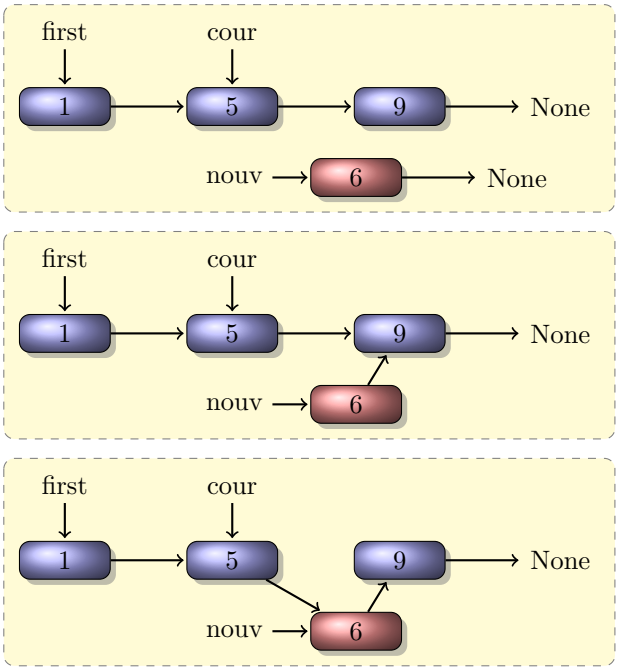


FIGURE 6.5 – Insertion dans une liste triée

Pour se positionner au bon endroit, il faut partir du début de la liste, et avancer tant que le suivant de l'élément courant est différent de None et tant qu'il est inférieur à l'élément à insérer. Ce principe est représenté figure 6.5.

L'insertion dans une liste triée est décrit par l'algorithme 6.5.

Il y a deux cas particuliers à tester :

— le cas où la liste est vide ;

— le cas où l'élément à insérer est le premier de la liste (le plus petit).

Il est bien entendu possible de tester tous ces cas dans l'algorithme. Une autre possibilité est de créer un élément temporaire (*tmp*), de raccrocher le début de la liste à *tmp* et de commencer le parcours de la liste à partir de *tmp*. La liste finale sera la liste privée de *tmp*. Il n'y a alors plus de cas particulier à tester !

En effet, si la liste est vide, le suivant de *tmp* vaut *None* et l'algorithme va accrocher *nouv* à la suite de *tmp*. La liste finale contient alors uniquement *nouv*, ce qui est le résultat attendu. Si *nouv* est le plus petit élément de la liste, le suivant de *tmp* est supérieur à *nouv*, et *nouv* est alors accroché après *tmp*, ce qui correspond au début de la liste.

Étude de l'algorithme

Le pire des cas pour cet algorithme est le cas où l'élément est à ajouter en fin de liste. L'insertion triée correspond alors à une insertion en fin de liste. Sa complexité est donc de $\Theta(n)$.

Algorithme 6.5 : Algorithme d'insertion dans une liste triée

```
Entrées : nouv : noeud à ajouter
Données : debut : noeud                                  # tête de liste
Données : n : noeud                                      # noeud courant
Données : tmp : noeud                                    # noeud temporaire
créer noeud tmp;
tmp.next ← debut;
n ← tmp;
tant que n.suivant ≠ None et n.suivant < nouv faire
    n ← n.suivant;
fin
# on est maintenant au bon endroit dans la liste
nouv.suivant ← n.suivant;
n.suivant ← nouv;
debut ← tmp.suivant                                      # supprimer l'élément temporaire
```

Le meilleur des cas correspond à une insertion en début de liste (complexité $\Theta(1)$).

Étudions la complexité moyenne de cet algorithme. Soit une liste composée de n éléments à laquelle nous voulons ajouter un élément e. Il existe $n+1$ emplacements possibles (tous équiprobables) pour e : en tête de liste, en deuxième position, ..., en dernière $(n+1)$ position. En moyenne, le coût de cet algorithme sera de :

$$\frac{1}{n+1}\sum_{i=1}^{n+1} i = \frac{1}{n+1}\frac{(n+1)(n+2)}{2} = \frac{n+2}{2} = \Theta(n)$$

6.1.5 Suppression d'un élément d'une liste

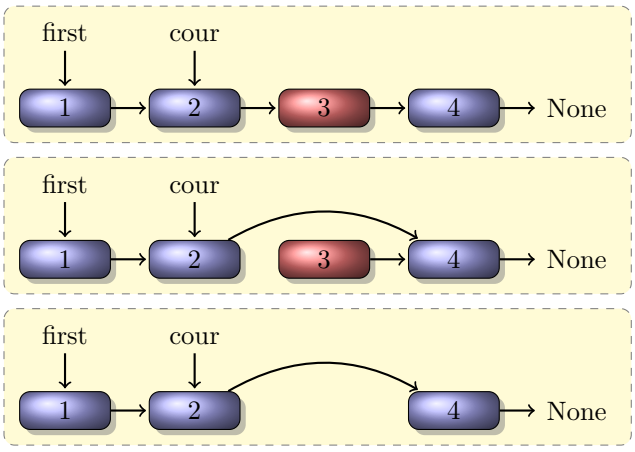


FIGURE 6.6 – Suppression d'un élément dans une liste

Le principe de la suppression d'un élément dans une liste est le suivant : il faut se placer avant l'élément à supprimer (*suppr*), puis modifier le chaînage de manière à supprimer le lien vers l'élément *suppr*. Cette opération est représentée figure 6.6.

Cet algorithme nécessite également la prise en compte de deux cas particuliers : le cas où la liste est vide et le cas où l'élément à supprimer est le premier de la liste. Une fois encore, la création d'un élément temporaire ajouté en début de liste permet d'éviter ces cas particuliers.

Algorithme 6.6 : Algorithme de suppression d'un élément dans une liste

Entrées : val : entier `# val : valeur du noeud à supprimer`
Données : debut : noeud `# tête de liste`
Données : n : noeud `# noeud courant`
Données : tmp : noeud `# noeud temporaire`
créer noeud tmp;
tmp.next ← debut;
n ← tmp;
tant que *n.suivant* ≠ *None et n.suivant.val* ≠ *val* **faire**
  n ← n.suivant;
fin
`# on est maintenant au bon endroit dans la liste`
n.suivant ← n.suivant.suivant;
debut ← tmp.suivant;

L'algorithme 6.6 représente ce traitement.

Remarque : l'élément n'est pas réellement supprimé, il est juste rendu inaccessible dans la liste. La suppression de la mémoire utilisée par l'élément sera automatiquement réalisée par le ramasse-miettes de Python.

Étude de l'algorithme

La complexité de cet algorithme est semblable à celle de l'algorithme d'insertion dans une liste triée. Les complexités moyenne et dans le pire des cas sont de $\Theta(n)$.

6.1.6 Liste doublement chaînée

Dans une liste doublement chaînée, chaque nœud connaît son successeur, mais également son prédécesseur. Il est alors nécessaire d'ajouter une variable d'instance à la classe *Node*, qui devient :

Listing 6.4 – Classe Node (liste doublement chaînée)

```
1  class Node:
2      def __init__(self, val=0):
3          """Creation d'un noeud ; les noeuds suivant et précédent
4          sont fixés à None.
5          """
6          self.val = val
7          self.next = None # Noeud suivant
8          self.prev = None # Noeud précédent
9
10     # ...
```

La méthode de recherche est identique à celle des listes simplement chaînées. Les méthodes d'insertion et de suppression doivent être légèrement modifiées : il faut mettre à jour les champs *next* et *prev* des différents nœuds. De plus, il est maintenant possible de se placer directement sur le dernier élément.

Remarque : la classe *list* de Python permet d'utiliser des listes. Elle contient toutes les fonctionnalités présentées dans ce chapitre.

6.2 Arbres

6.2.1 Différentes classes d'arbres

Définition 6.2 (Arbre)**.** *Un arbre est une collection non vide de nœuds et d'arêtes possédant les propriétés suivantes :*

- *un nœud est un objet simple ;*
- *une arête est un lien entre deux nœuds ;*
- *une branche de l'arbre est une suite de nœuds distincts dans laquelle deux nœuds successifs sont reliés par une arête ;*
- *il existe un nœud spécial appelé racine ;*
- *la propriété qui définit un arbre est qu'il existe exactement une branche entre la racine et chacun des autres nœuds de l'arbre.*

Il est habituel de représenter les arbres avec la racine située au dessus de tous les autres nœuds. Chaque nœud, excepté la racine, possède un et un seul nœud « au dessus de lui », appelé père. Les nœuds directement situés sous un nœud sont appelés fils. On peut aussi parler de frère, grand-père, etc. Les nœuds n'ayant pas de descendance sont appelés feuilles ou nœuds terminaux.

Définition 6.3 (Hauteur d'un arbre)**.** *La hauteur d'un nœud est la longueur du plus long chemin de ce nœud aux feuilles qui en dépendent plus 1 : c'est le nombre de nœuds du chemin. La hauteur d'un arbre est la hauteur de sa racine. L'arbre vide a une hauteur 0, et l'arbre réduit à une racine a une hauteur 1.*

Un exemple d'arbre est représenté figure 6.7. La hauteur de cet arbre est de 3.

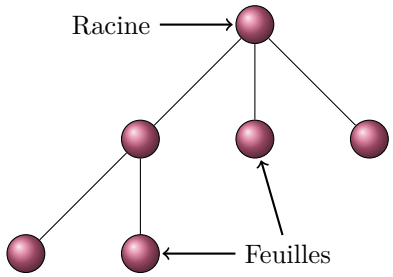


FIGURE 6.7 – Arbre

Définition 6.4 (Arbre binaire)**.** *Un arbre binaire est un arbre dont chaque nœud possède au plus deux fils.*

Définition 6.5 (Arbre binaire complet)**.** *Un arbre est complet si chaque nœud qui n'est pas une feuille possède exactement deux fils et où toutes les feuilles sont au même niveau.*

La figure 6.8 représente l'arbre complet de hauteur 3. L'arbre représenté par la figure 6.9 est un arbre dégénéré appelé peigne droit, c'est-à-dire un arbre pour lequel tous les fils gauches sont nuls. Dans ce cas, l'arbre est équivalent à une liste.

Arbre binaire en Python

Une structure d'arbre binaire est représentée figure 6.10. Comme dans le cas des listes, cette structure repose sur des nœuds qui sont chaînés entre eux. Toutefois, chaque nœud possède non pas un mais deux successeurs qui sont souvent nommés fils gauche et fils droit. Il est également possible d'ajouter le chaînage du fils vers le père, et on obtient alors l'équivalent des listes doublement chaînées.

La classe Python *Node* correspondant à un arbre binaire est représentée dans le listing 6.5:

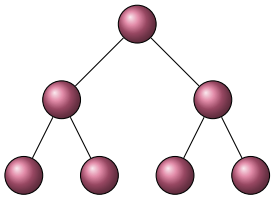


FIGURE 6.8 – Arbre complet

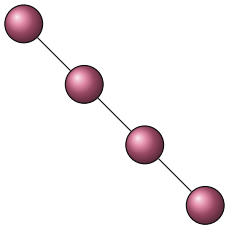


FIGURE 6.9 – Arbre peigne

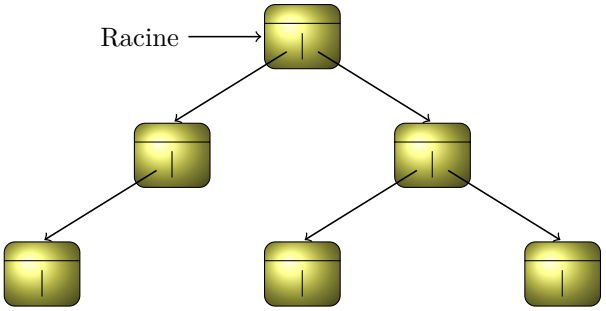


FIGURE 6.10 – Représentation d'un arbre binaire

Listing 6.5 – Classe Node (arbre binaire)

```
1 class Node:
2     def __init__(self, val=0):
3         """Creation d'un noeud ; les fils sont fixés à None.
4         """
5         self.val = val
6         self.left = None # Fils gauche
7         self.right = None # Fils droit
8 
9     # ...
```

Remarque : il est également possible de définir des arbres ternaires, quaternaires, et plus généralement des arbres n-aires. Dans ce cas, chaque nœud possédera une liste de n fils.

Un exemple de classes permettant de définir un arbre binaire est représenté figure 6.11.

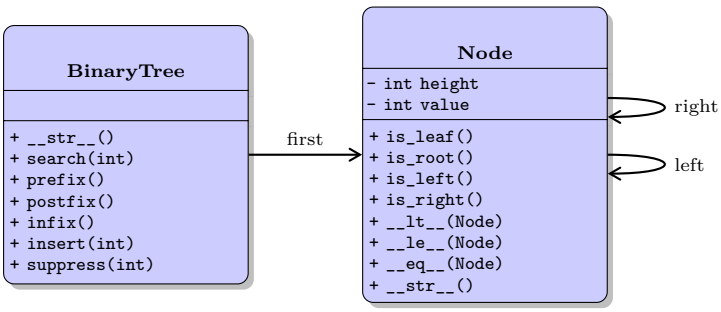


FIGURE 6.11 – Classe arbre binaire

Arbre binaire de recherche

Définition 6.6 (Arbre binaire de recherche)**.** *Un arbre binaire de recherche est un arbre binaire vérifiant la propriété suivante : pour tout nœud n de l'arbre, tous les nœuds de son sous-arbre gauche ont une valeur inférieure ou égale à n, et tous les nœuds de son sous-arbre droit ont une valeur supérieure ou égale à n.*

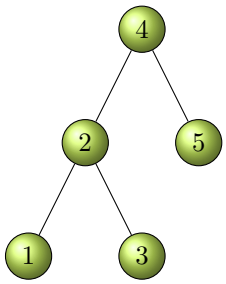


FIGURE 6.12 – Arbre binaire de recherche

La figure 6.12 représente un arbre binaire de recherche. Ce type d'arbre est une structure dynamique pour laquelle les méthodes d'ajout, de recherche et de suppression de données sont beaucoup plus efficaces que dans le cas d'une liste.

Arbre général

Définition 6.7 (Arbre général)**.** *Un arbre général est un arbre pour lequel chaque nœud possède un nombre quelconque de fils (non borné* a priori*).*

Les arbres généraux peuvent servir à représenter par exemple une arborescence de répertoires. Ils peuvent également servir à explorer les différentes possibilités pour un jeu (le premier joueur a le choix entre n_1 coups, pour chacun de ces coups le second joueur a le choix entre $m_1, \ldots, m_{n_1}$ coups et ainsi de suite). Les arbres généraux sont plus lourds à gérer que les arbres n-aires, c'est pourquoi on limitera leur utilisation autant que possible.

Un arbre général sera représenté ainsi : chaque nœud possède un premier fils, et chaque fils possède une liste de frères. Le code Python correspondant à un tel nœud est :

```
1  class Node:
2      def __init__(self, val=0):
3          """Creation d'un noeud ; le fils est fixé à None,
4          les frères à la liste vide.
5          """
6          self.val = val
7          self.child = None # Fils
8          self.brothers = [] # Frères
9
10     # ...
```

6.2.2 Création des nœuds

Nous considérerons par la suite uniquement le cas d'un arbre binaire de recherche contenant des valeurs entières. La classe *Node* correspondante est définie dans le listing 6.5.

La classe *BinaryTree* représentant l'arbre possède une variable d'instance de type *Node*. Sa déclaration est :

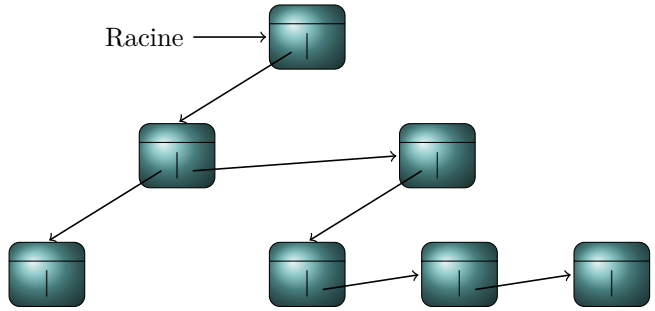


FIGURE 6.13 – Arbre général

Listing 6.6 – Classe arbre binaire

```
class BinaryTree:
    def __init__(self):
        """Creation d'un arbre vide. La racine est initialisée à None.
        """
        self.__root = None
```

6.2.3 Recherche d'un élément dans un arbre binaire de recherche

La recherche d'un élément dans un arbre est une des opérations les plus simples à réaliser. Il est possible d'écrire cette méthode de manière récursive ou itérative. Le principe est de comparer la valeur recherchée avec celle du nœud courant :

— si elle est égale, le nœud est trouvé ;
— si elle est inférieure, il faut poursuivre la recherche dans le sous-arbre gauche ;
— sinon, il faut poursuivre la recherche dans le sous-arbre droit.

Les algorithmes de recherche itérative et récursive sont détaillés algorithme 6.7 et algorithme 6.8.

Algorithme 6.7 : Recherche itérative dans un arbre binaire de recherche

Entrées : entier val
Données : racine : noeud `# racine de l'arbre`
Données : n : noeud `# noeud courant`
n ← racine;
tant que $n \neq None$ *et* $n.val \neq val$ **faire**
  si $val < n.val$ **alors**
    n ← n.filsGauche;
  sinon
    n ← n.filsDroit;
  fin
fin
retourner n `# n a la valeur recherchée ou n vaut None`

Étude de l'algorithme

Étudions la complexité de la recherche d'un élément dans un arbre binaire de recherche dans le pire des cas. Remarquons que le pire des cas est toujours le cas où l'élément recherché n'est pas dans l'arbre. Dans ce cas, rechercher un élément coûte au maximum la hauteur h de l'arbre : $\Theta(h)$.

Fonction rechRec(entier val, Noeud cour)

```
si cour = None ou cour.val = val alors
    retourner cour;
sinon si val < cour.val alors
    retourner rechRec (val, n.filsGauche);
sinon
    retourner rechRec (val, n.filsDroit);
fin
```

Algorithme 6.8 : Recherche récursive dans un arbre binaire de recherche

```
Entrées : entier val                          # Valeur à rechercher
retourner rechRec(val, racine)
```

Cherchons tout d'abord la configuration d'arbre la plus défavorable : il s'agit d'un *peigne*. Dans ce cas, la hauteur de l'arbre vaut $h = n$, et la complexité de la recherche est de $\Theta(n)$, comme dans le cas d'une liste.

Considérons maintenant le cas où l'arbre est complet. Dans ce cas, n vaut $\sum_{i=1}^{h-1} 2^i = 2^h - 1$. Donc, $h = \log_2(n+1) \simeq \log_2 n$. La complexité est donc de $\Theta(\log n)$. La recherche d'un élément dans un arbre binaire de recherche complet est donc une méthode très efficace.

6.2.4 Parcours d'un arbre binaire de recherche

Parcourir un arbre veut dire explorer l'ensemble des nœuds le composant. Il existe plusieurs types de parcours :

- parcours en largeur, c'est-à-dire visitant l'arbre « par étage ». Tout d'abord la racine, puis les fils de première génération, et ainsi de suite jusqu'aux feuilles.
- parcours en profondeur qui consiste à suivre une branche jusqu'à la feuille, puis remonter pour traiter les autres branches. En fonction du moment où sera traité le nœud courant, on parlera de parcours préfixe (nœud courant avant le sous-arbre gauche), infixe (nœud courant entre les deux sous-arbres), ou postfixe (après le sous-arbre droit). Cette méthode de parcours est classique et elle s'écrit très simplement de manière récursive. Son écriture itérative est beaucoup plus délicate et ne sera pas abordée.

Toutes les méthodes de parcours récursives sont appelées à partir de la racine, comme par exemple *prefixe(root)*.

Parcours préfixe

Lors d'un parcours préfixe, le nœud courant est traité en premier. Puis le sous-arbre gauche est traité, et enfin le sous-arbre droit. Un parcours préfixe appliqué à l'arbre de la figure 6.12 traitera les nœuds : 4 2 1 3 5.

Procédure prefixe(Noeud cour)

```
si cour ≠ None alors
    traiter cour;
    prefixe (cour.filsGauche);
    prefixe (cour.filsDroit);
fin
```

Parcours infixe

Lors d'un parcours infixe, le sous-arbre gauche est traité en premier. Puis le nœud courant est traité, et enfin le sous-arbre droit. Un parcours infixe appliqué à l'arbre de la figure 6.12 traitera les nœuds : 1 2 3 4 5. On remarque qu'un parcours infixe appliqué à un arbre binaire de recherche affiche les nœuds dans l'ordre.

Procédure infixe(Noeud cour)

```
si cour ≠ None alors
    infixe (cour.filsGauche);
    traiter cour;
    infixe (cour.filsDroit);
fin
```

Parcours postfixe

Lors d'un parcours postfixe, le sous-arbre gauche est traité en premier. Puis le sous-arbre droit est traité, et enfin le nœud courant. Un parcours postfixe appliqué à l'arbre de la figure 6.12 traitera les nœuds : 1 3 2 5 4.

Procédure postfixe(Noeud cour)

```
si cour ≠ None alors
    postfixe (cour.filsGauche);
    postfixe (cour.filsDroit);
    traiter cour;
fin
```

6.2.5 Ajout d'un élément dans un arbre binaire de recherche

Dans un arbre, les nœuds sont toujours ajoutés en tant que feuilles. De plus, nous sommes dans le cas d'un arbre binaire de recherche, et il faut donc respecter la notion d'ordre. L'algorithme d'insertion d'un élément n est donc le suivant :

— se déplacer jusqu'au nœud en dessous duquel il faut ajouter l'élément n (si $n.val > cour.val$ aller à droite, sinon aller à gauche).
— insérer le nœud n en tant que feuille.

Algorithme 6.9 : Insertion dans un arbre binaire de recherche

```
Entrées : noeud val
si arbre = ∅ alors
    racine ← val;
sinon
    insert(val, racine)                    # insère val sous la racine
fin
```

Cet algorithme se traduit plus formellement par la figure 6.9. Il peut s'écrire de manière récursive ou itérative. La solution retenue ici est la méthode itérative.

Algorithme 6.10 : insert(noeud val,noeud pere)

```
n ← pere;
tant que n ≠ None faire
    pere ← n;
    si val < n alors
        n ← n.filsGauche;
    sinon
        n ← n.filsDroit;
    fin
fin
# maintenant le fils de pere est None
si val < pere alors
    pere.filsGauche ← val;
sinon
    pere.filsDroit ← val;
fin
```

Étude de l'algorithme

L'ajout d'un nouveau nœud se fait toujours sur les feuilles de l'arbre. Le coût de l'insertion est donc le coût nécessaire pour se déplacer jusqu'à la bonne feuille. Comme dans le cas de la recherche d'un élément, tout dépend de la forme de l'arbre. La complexité de cette méthode varie de $\Theta(n)$ dans le cas d'un arbre dégénéré (peigne) à $\Theta(\log n)$ dans le cas d'un arbre complet.

6.2.6 Suppression d'un élément dans un arbre binaire de recherche

La suppression d'un nœud n d'un arbre est semblable à la suppression d'un nœud dans une liste : il faut placer le nœud courant (*cour*) dans l'arbre avant (i.e. au dessus de) n puis supprimer n du chaînage. Contrairement au cas d'une liste, il faut maintenant raccrocher deux sous-arbres à *cour* alors qu'un seul emplacement est disponible. Une solution à ce problème consiste à placer le sous-arbre gauche en tant que fils de *cour* puis d'insérer le sous-arbre restant grâce à la méthode insérer vue précédemment.

Remarque : l'algorithme de suppression nécessite soit le test de plusieurs cas particuliers (arbre vide, suppression de la racine), soit l'utilisation d'un nœud temporaire. Cette dernière méthode a été choisie dans l'algorithme décrit en figure 6.11. La racine de l'arbre sera rattachée en tant que fils droit du nœud temporaire (par exemple).

Exemple : l'arbre initial est représenté figure 6.14. Nous voulons supprimer le nœud 8. Première étape (voir figure 6.15) : supprimer le nœud 8 et faire remonter le sous-arbre gauche (5-6-7) à sa place.

Deuxième étape (voir figure 6.16) : insérer le nœud orphelin 9 (avec tous ses fils éventuels) dans le sous-arbre que nous venons de faire remonter (à partir de 6). On obtient alors l'arbre représenté figure 6.17, qui est toujours un arbre binaire de recherche.

Étude de l'algorithme

Le coût de la suppression d'un élément dans un arbre binaire de recherche est équivalent au coût d'ajout d'un élément dans l'arbre.

Algorithme 6.11 : Suppression dans un arbre binaire de recherche

Entrées : entier val
tmp ← Noeud() `# noeud temporaire pour éviter les cas particuliers`
tmp.right ← racine;
n ← tmp;
tant que *n ≠ None et n.val ≠ val* **faire**
  pere ← n;
  si *val < n* **alors**
    n ← n.filsGauche;
  sinon
    n ← n.filsDroit;
  fin
fin
`# maintenant le fils de pere est soit l'élément à supprimer soit None`
si *n ≠ None et val < pere* **alors**
  `# supprimer fils gauche`
  Noeud orphelin ← n.filsDroit;
  pere.filsGauche ← n.filsGauche;
  inserer(orphelin, pere.filsGauche);
sinon si *n ≠ None et val > pere* **alors**
  `# supprimer fils droit`
  Noeud orphelin ← n.filsDroit;
  pere.filsDroit ← n.filsGauche;
  inserer(orphelin, pere.filsDroit);
sinon
  racine ← tmp.right;
fin

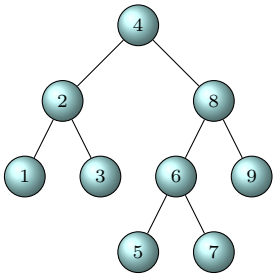


FIGURE 6.14 – Suppression : étape 1

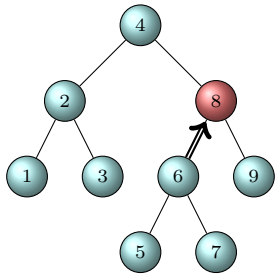


FIGURE 6.15 – Suppression : étape 2

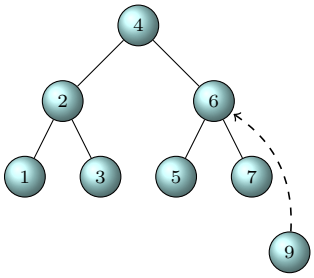


FIGURE 6.16 – Suppression : étape 3

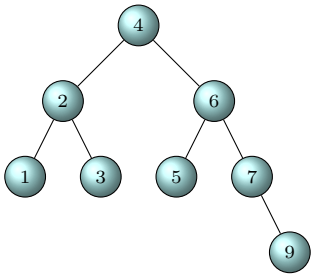


FIGURE 6.17 – Suppression : étape 4

6.2.7 Équilibrage d'un arbre binaire de recherche

Nous avons vu que les méthodes de recherche, d'insertion et de suppression dans un arbre binaire de recherche sont très efficaces lorsque l'arbre est complet. Il n'est malheureusement en général pas possible d'obtenir un arbre complet, et il faut alors se contenter d'approximations.

Une possibilité est de construire un arbre *équilibré*, ce qui est toujours possible et peut se faire simplement.

Définition 6.8 (Arbre équilibré)**.** *Un arbre binaire est dit équilibré lorsque la différence entre les hauteurs des fils gauche et droit de tout nœud ne peut excéder 1.*

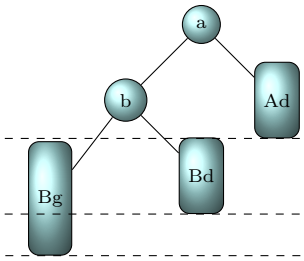


FIGURE 6.18 – Arbre déséquilibré : cas 1

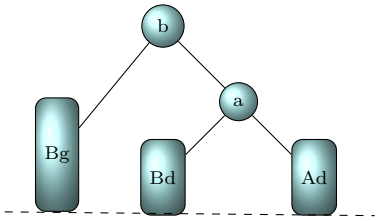


FIGURE 6.19 – Arbre rééquilibré : cas 1

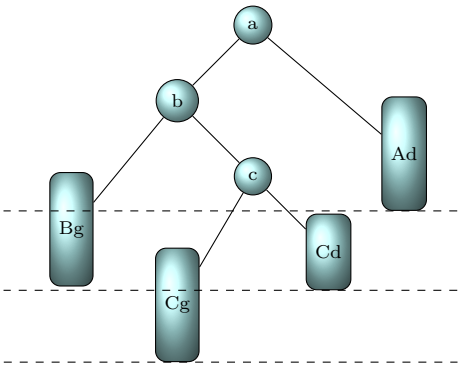


FIGURE 6.20 – Arbre déséquilibré : cas 2

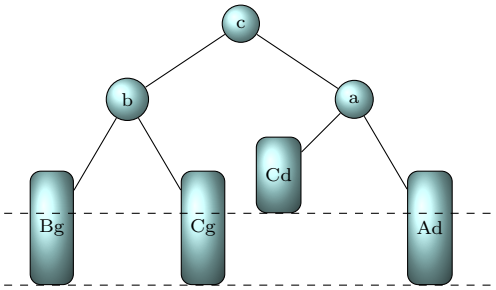


FIGURE 6.21 – Arbre rééquilibré : cas 2

Dans un arbre binaire de recherche équilibré, la complexité des méthodes de recherche, d'insertion et de parcours est de $\Theta(\log n)$.

Pour obtenir un arbre équilibré, il faut vérifier après chaque insertion et chaque suppression si l'arbre comporte un déséquilibre. Si oui, il faut alors lancer la procédure de rééquilibrage.

Après une insertion, deux types de déséquilibres sont possibles. Le premier cas est représenté figure 6.18, et l'équilibrage correspondant figure 6.19. Le second cas est représenté figure 6.20, et l'équilibrage correspondant figure 6.21.

6.3 Tables de hachage

Définition 6.9 (Table de hachage)**.** *Une table de hachage (*hash table*) est une structure de données permettant d'associer une clé à chaque élément, de stocker des paires clé/élément, et de trouver très rapidement un élément (grâce à sa clé). On dit que les éléments sont stockés dans des* alvéoles*.*

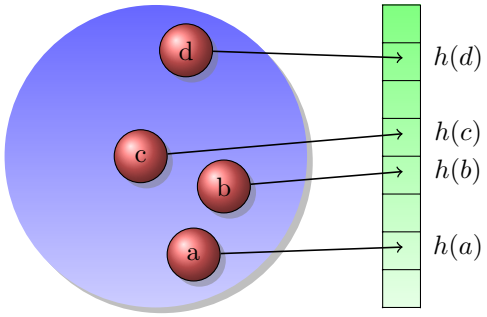


FIGURE 6.22 – Table de hachage

Afin de réaliser une table de hachage, il est nécessaire de pouvoir associer une valeur entière à chaque élément. Une fonction permettant cette association est alors définie : la fonction de hachage, ou *hash function*. Cette fonction associe une valeur de hachage ou *hash code*, ou clé à un élément. Soit $\mathcal{K}$ l'ensemble des éléments à insérer dans la table.

Définition 6.10 (Fonction de hachage)**.** *Une fonction de hachage est une fonction qui associe une clé à chaque élément ; chaque clé permet d'identifier l'alvéole contenant l'élément. La fonction de hachage h est définie par :*

$$\begin{array}{rccl} h: & \mathcal{K} & \rightarrow & [0, n-1] \subset \mathbb{N} \\ & e & \mapsto & h(e) \end{array}$$

Une table de hachage correspond à un tableau ou une liste T de taille n pour lequel chaque élément e de clé $k = h(e)$ est stocké dans la case $T[k]$. Une telle table de hachage est représentée figure 6.22.

Si la fonction de hachage h est injective, alors à chaque valeur possible correspond une unique clé. Dans une case du tableau ou de la liste peut alors se trouver un seul élément. L'insertion (algorithme 6.12) et la recherche (algorithme 6.13) d'un élément dans la table de hachage s'effectue en temps constant $\Theta(1)$ (*i.e.* le temps est indépendant du nombre d'éléments dans la table et vaut le temps nécessaire pour calculer la fonction de hachage).

Algorithme 6.12 : Insertion d'un élément dans une table de hachage

Entrées : Table T, Element e
T[h(e)] ← e ;

Algorithme 6.13 : Recherche d'un élément dans une table de hachage

Entrées : Table T, Element e
retourner T[h(e)];

6.3.1 Collisions dans une table de hachage

Il est assez simple de trouver une fonction de hachage injective : si x est un mot composé de p caractères de code ASCII $a_0, ..., a_{p-1}$, il suffit par exemple de prendre la fonction

$$h : x \mapsto \sum_{i=0}^{p-1} (a_i).256^i$$

Cependant, une telle fonction nous force à réserver inutilement une grande quantité de mémoire (la plupart des cases du tableau composant la table de hachage seront vides). On ajoute donc souvent une contrainte sur la valeur maximale que peut prendre $h(x)$. Il est alors beaucoup plus difficile de respecter la condition d'injectivité.

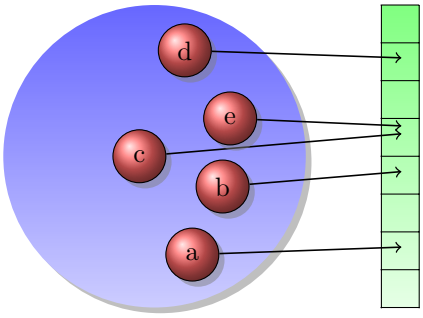


FIGURE 6.23 – Table de hachage avec collisions

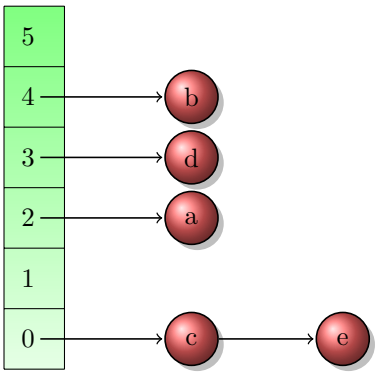


FIGURE 6.24 – Résolution des collisions

C'est pourquoi cette condition n'est en général pas vérifiée. Il se peut alors que deux éléments différents aient le même code de hachage. On dit alors qu'il y a collision entre plusieurs éléments : ils sont à ranger dans la même case du tableau. Une bonne fonction de hachage doit limiter au maximum les collisions (car elles nuisent à son efficacité), mais elle ne peut en général pas les éviter totalement. Le problème consiste alors de faire de la fonction de hachage une variable aléatoire le plus uniforme possible.

Une table de hachage contenant des collisions est représentée figure 6.23. Dans cet exemple, les éléments c et e ont la même valeur de hachage.

Une table de hachage doit donc avoir la possibilité de contenir plusieurs éléments dans chaque case du tableau, ce qui est simplement réalisé en les plaçant dans une liste chaînée. Chaque case du tableau contient alors la liste de tous les éléments ayant une certaine clé. La structure obtenue est représentée figure 6.24.

Chaque case du tableau $T[i]$ est alors une liste chaînée. Cette liste doit supporter les opérations classiques d'insertion en tête, de recherche et de suppression d'un élément. Les fonctions d'insertion (6.14) et de recherche (6.15) dans la table de hachage s'appuient sur ces opérations.

Algorithme 6.14 : Insertion d'un élément dans une table de hachage avec collisions

Entrées : Table T, Element e

```
T[h(e)].insereEnTeteDeListe(e);
```

Algorithme 6.15 : Recherche d'un élément dans une table de hachage avec collisions

Entrées : Table T, Element e

```
retourner T[h(e)].rechercheDansLaListe(e);
```

6.3.2 Choix de la fonction de hachage

Définition 6.11 (Facteur de remplissage)**.** *Étant donnée une table de hachage* T *à* m *alvéoles contenant* n *éléments, on définit le facteur de remplissage* $\alpha = \frac{n}{m}$.

La figure 6.25 représente la pire des fonctions de hachage possible : tous les éléments ont la même clé. Dans ce cas, les opérations d'insertion, de suppression et de recherche dans la table ont le même coût que dans une liste chaînée : $\Theta(n)$.

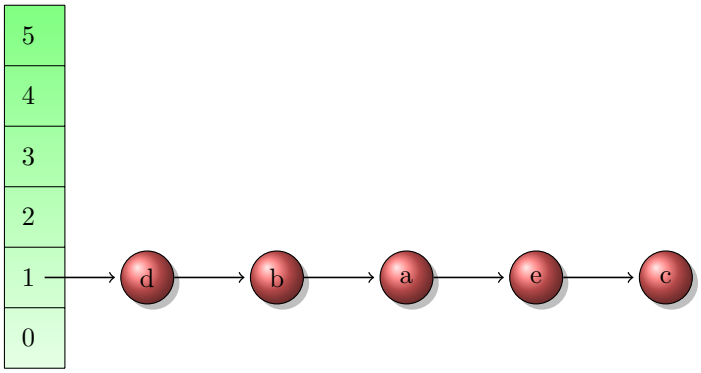


FIGURE 6.25 – Mauvaise fonction de hachage

Dans le cas d'une « bonne » fonction de hachage, les éléments sont uniformément répartis, et si le nombre d'éléments et la taille du tableau sont du même ordre de grandeur, alors les opérations de base ont une complexité $\Theta(1)$. Plus généralement, la complexité moyenne de la recherche d'un élément dans la table vaut $\Theta(1+\alpha)$.

Il existe plusieurs méthodes pour construire une fonction de hachage efficace, parmi lesquelles on pourra citer la méthode de la division et la méthode de la multiplication.

Méthode de la division

Dans la méthode de la division, un entier λ est associé à chaque valeur, et la clé de hachage est obtenue par le reste de la division de λ par la taille m de la table.

Par exemple, si $\lambda = 100$ et $m = 12$, alors $h(\lambda) = 4$. Cette méthode est simple à mettre en oeuvre et assez efficace, à condition toutefois de bien choisir m. Il faut par exemple éviter que m soit trop proche d'une puissance de 2. Les nombres premiers sont généralement de bons choix pour m.

Exemple : on souhaite par exemple stocker 2000 éléments, et un α voisin de 3 est acceptable. On peut alors choisir $m = 701$.

Méthode de la multiplication

Dans la méthode de la multiplication, on crée la fonction de hachage en multipliant la valeur λ par une constante $0 < A < 1$, on extrait la partie décimale de λA, on multiplie ce résultat par m, et on prend la partie entière du résultat.

En résumé, $h(\lambda) = \lfloor m(\lambda A \mod 1) \rfloor$.

Avec cette méthode, le choix de m n'est absolument pas critique. On choisit généralement $m = 2^p$ pour simplifier les calculs numériques, mais ce n'est pas une obligation. Concernant A, si on manipule des entiers sur w bits, on cherche une valeur sous la forme $s/2^w$. Dans [Knu73], Knuth conseille de prendre une valeur proche de $(\sqrt{5}-1)/2$.

Exemple : La valeur λ vaut 123456 pour une taille da table de hachage vérifiant $p = 14$, donc $m = 2^{14} = 16384$. Sur un système dont les entiers sont codés sur 32 bits, un bon choix pour A pourra être : $A = 2654435769/2^{32} \approx (\sqrt{5}-1)/2$. Ainsi $\lambda A \approx 76300.00410081446$, soit une partie décimale de 0.00410081446. On a donc $m(kA \mod 1) = 67.187744140625 \rightsquigarrow h(k) = 67$.

Exemples

Considérons le cas d'un dictionnaire que nous voulons stocker dans une table de hachage. On suppose que les mots sont uniquement composés de lettres entre a et z (uniquement des minuscules, pas d'accents). On considère qu'un mot est de la forme $m = a_0a_1...a_{m-1}$ où chaque a_i est une lettre. L'exemple 6.1 représente une mauvaise fonction de hachage et 6.2 une bonne fonction.

Exemple 6.1 (Mauvaise fonction de hachage)**.** *Considérons la fonction qui à un mot associe le code ASCII de sa première lettre moins le code ASCII de* a*. Pour chaque mot, cette fonction retourne une valeur entre 0 et 25 inclus :*

$$\begin{aligned} h : \quad & \mathcal{K} \to [0, 25] \\ & x \mapsto a_0 - \text{'a'} \end{aligned}$$

Il s'agit d'une mauvaise fonction de hachage, car la répartition est loin d'être uniforme.

Exemple 6.2 (Bonne fonction de hachage)**.** *Une bonne fonction de hachage consiste, par exemple, à considérer la somme des valeurs des caractères pondérées par 26 puissance la position de la lettre dans le mot. Afin de respecter la contrainte* $\forall e, h(e) \in [0, n-1]$*, cette somme est prise modulo la taille du tableau.*

$$\begin{aligned} h : \quad & \mathcal{K} \to [0, n-1] \\ & x \mapsto \left(\sum_{i=0}^{p-1} (a_i - \text{'a'}).26^i \right) \mod n \end{aligned}$$

Cette fonction conduit à une bonne répartition si $n \wedge 26 = 1$*.*

Remarque : la classe *dict* de Python permet d'utiliser des tables de hachage.

Tu sais, remarqua Arthur, c'est en de tels moments, quand je me retrouve coincé dans un sas vogon en compagnie d'un natif de Bételgeuse, au seuil d'une mort imminente par asphyxie dans les profondeurs de l'espace, que je regrette de ne pas avoir écouté ce que me disait ma mère quand j'étais petit.
— Eh bien, que te disait-elle ?
— Je ne sais pas. Je n'ai pas écouté.

Douglas Adams

7

Mécanismes spécifiques de l'API Python

Sommaire

7.1 Gestion de la mémoire en Python

La gestion de la mémoire est relativement simple en Python. En effet, elle est en grande partie prise en charge par la machine virtuelle. Toutefois, une bonne compréhension des mécanismes sous-jacents permet d'éviter de nombreuses erreurs de programmation. Il est en particulier important de comprendre le mécanisme de référence.

7.1.1 Création d'objets et références

Chaque variable Python occupe un emplacement mémoire. Cet emplacement est défini à la construction de l'objet. La mémoire est réservée par la machine virtuelle et elle le reste tant qu'il existe une référence vers cet objet. Une référence vers un objet est une variable permettant de le manipuler. Exemple :

Listing 7.1 – Opérations sur les références

```
1  # Pour l'instant, x ne reference aucun objet
2  x = None
3
4  # Un nouvel objet est créé et x référence cet objet
5  x = list(range(4))
6
7  # y est une nouvelle référence ; x et y référencent le même objet
8  y = x
9
10 # z référence un nouvel objet qui est une copie de x
11 z = list(x)
12
13 # Modification de x
14 x[0] = -1
15
16 # x et y sont modifiés, pas z
17 print(x, y, z)
```

Cet exemple est représenté figure 7.1. Dans cet exemple, il existe trois références vers des

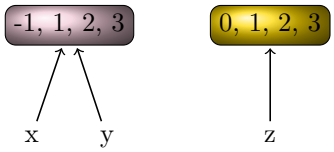


FIGURE 7.1 – Références

objets : *x, y* et *z*, et il existe deux objets. *x* et *y* référencent le même objet, c'est-à-dire le même emplacement mémoire. Ainsi, toute modification de l'objet *x* modifie également l'objet *y*. Par contre, *z* est une *copie* de *x*, c'est-à-dire un autre objet ayant le même contenu. Ainsi, une modification de *x* ne modifie pas le contenu de *z*.

L'exécution de ce programme affichera le résultat :

```
[-1, 1, 2, 3] [-1, 1, 2, 3] [0, 1, 2, 3]
```

Remarque : dans le listing 7.1, dire en ligne 1 que *x* ne référence aucun objet est un abus de langage. Dans les faits, *x* référence un objet qui est *None*.

7.1.2 Destruction d'objets : le ramasse-miettes

En Python, lorsqu'un objet n'est plus utile, il n'est pas nécessaire de le détruire explicitement. Chaque objet dispose d'un compteur de références, qui détermine le nombre de fois que la zone mémoire occupée par l'objet est référencée par le programme. Ce peut être directement par une variable du programme, par une variable locale d'une fonction en cours d'exécution, ou par une variable d'insance d'un autre objet. Si ce compteur de références tombe à 0, la zone mémoire utilisée par l'objet ne pourra plus jamais être référencée.

Un ramasse-miettes (un objet de type *garbage collector* attaché à chaque programme) est chargé de désallouer la mémoire des objets dont le compteur de références tombe à 0.

Un cycle de références survient si `A` a une variable d'instance dont la valeur est `B`, et `B` a une variable d'instance dont la valeur est `A`. Lorsque ni `A` ni `B` ne sont plus référencés par le programme, chaque objet a un compteur de références de 1, mais aucun n'est accessible, et leur espace mémoire doit être désalloué. Le listing 7.2 illustre ce problème.

Si un objet `C` se référence lui-même, l'effet est le même. Si le ramasse-miette utilise uniquement le mécanisme de comptage de références, ces zones mémoires sont à la fois inutiles et allouées : c'est une fuite de mémoire, et donc une perte de mémoire disponible.

Listing 7.2 – Exemple de références croisées

```
class Ref:
    def __init__(self, val=None):
        self.val = val

if __name__ == "__main__":
    A = Ref()   # Pour l'instant A ne référence personne
    # Ici le compteur de références de A vaut 1
    B = Ref(A)  # B référence A
    # Ici le compteur de A vaut 2, le compteur de B vaut 1
    A.val = B   # A référence B
    # Ici le compteur de A vaut 2, le compteur de B vaut 2
    A = None    # Le compteur de références de A est toujours de 1
    B = None    # Le compteur de références de B est toujours de 1
```

Depuis la version 2.0 de Python, le ramasse-miettes est également capable de détecter les cycles de références. Cette fonctionnalité est cependant assez lourde – algorithmiquement – et n'est lancée que de temps en temps, ou lorsque certains seuils de quantité d'objets actifs dans le programme sont atteints.

7.1.3 Copie d'objets

Copier un objet est une tâche délicate, en particulier si l'objet à copier contient des références vers d'autres objets. Il existe alors deux types de copies possibles :

— copie superficielle ;
— copie profonde.

Le listing 7.3 illustre ces deux types de copie.

Listing 7.3 – Copie superficielle et copie profonde

```
import copy


class X:
    def __init__(self, v1, v2):
        self.v1 = v1
        self.v2 = v2

    def __copy__(self):
        """Copie superficielle :
             Copie simplement les variables d'instance.
        """
        return X(self.v1, self.v2)

    def __deepcopy__(self, memo):
        """Copie profonde :
```

```
17              Applique récursivement la copie profonde à toutes les
18              variables d'instance.
19          """
20          res = X(copy.deepcopy(self.v1), copy.deepcopy(self.v2))
21          memo[id(self)] = res
22          return res
23
24      def __str__(self):
25          return str(self.v1) + ' ' + str(self.v2)
26
27  if __name__ == "__main__":
28      # Premier paramètre type simple, second paramètre objet
29      a = X(1, [2, 3])
30      b = copy.copy(a)       # copie superficielle de a
31      c = copy.deepcopy(a)   # copie profonde de a
32      a.v1 = 42              # Modification de la variable simple
33      a.v2.append(4)         # Modification de la variable objet
```

Réaliser une copie superficielle d'un objet de type X revient à copier uniquement les références de ses variables d'instance :

Les champs *obj* de l'objet initial et de la copie référencent le même objet. Modifier la variable d'instance *obj* pour un des deux le modifie également pour l'autre. Ce n'est en général pas le comportement attendu pour une copie. Cette copie est illustrée figure 7.2.

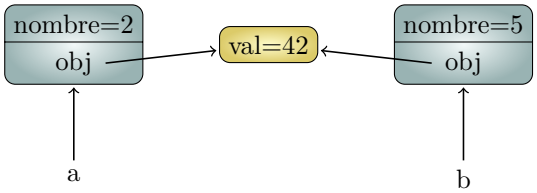


FIGURE 7.2 – Copie superficielle

Réaliser une copie profonde d'un objet consiste à réaliser une copie de toutes les variables d'instance de l'objet.

Cette fois, toutes les variables sont copiées. Remarquer que la copie profonde d'une classe nécessite des méthodes de copies profondes dans toutes les classes des variables d'instances la composant. Cette copie est illustrée figure 7.3.

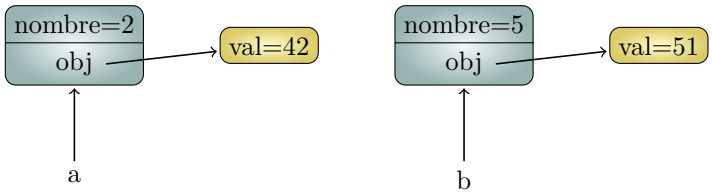


FIGURE 7.3 – Copie profonde

Attention : dans le cas d'une liste doublement chaînée, la copie profonde peut être délicate à implanter.

7.2 Exceptions

Les exceptions Python sont un mécanisme permettant de traiter les erreurs survenant en cours d'exécution. Une exception détourne l'exécution normale du programme vers un bloc de traitement de l'erreur.

Par défaut, les exceptions ne sont généralement pas gérées par le programme. Elles provoquent l'affichage d'un message d'erreur et l'interruption du programme. Le listing 7.4 est un exemple dans lequel l'exception *ZeroDivisionError* est provoquée.

Listing 7.4 – Exemple d'exception

```
  >>> a = 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Bloc try-except-finally

Le traitement local d'une exception se fait par l'utilisation du bloc *try-except*. Le mot-clé *try* signifie que certaines instructions du bloc peuvent lever des exceptions. Le bloc *except* se charge de traiter ces exceptions.

Listing 7.5 – Utilisation de try/except

```
1 try:
2     # Cette fonction peut lever une exception
3     methode_levant_une_exception()
4 # Ici on récupère toutes les exceptions héritant de Exception
5 except Exception as err:
6     # Traitement de l'exception
7     print(err)
8 # Suite du programme
```

Remarque : un bloc *try* peut lever plusieurs exceptions. Il sera alors suivi de plusieurs blocs *except*. Exemple :

```
1 try:
2     # ...
3 except Exception1:
4     # ...
5 except Exception2:
6     # ...
7 finally:
8     # ...
```

L'instruction finally, qui est optionnelle, sert à définir les traitements à exécuter après les blocs *try* et *except* dans tous les cas, qu'une exception soit levée ou non.

Exemple

Dans le listing 7.6, on convertit des chaînes de caractères en nombres réels. Dans le cas où la conversion n'est pas possible, l'exception est gérée.

Listing 7.6 – Gestion d'exception

```
for nombre in ['1', 'un', '1.0', '1j']:
```

```
    # Des exceptions peuvent se produire dans le bloc "try"
    try:
        # L'instruction float() peut lever une exception
        val = float(nombre)
        print(val)
    # Si une exception de type "ValueError" se produit,
    # exécuter le bloc suivant
    except ValueError as err:
        # Gestion minimale : on se contente d'afficher l'exception
        print(err)
```

Attention : les exceptions ne sont en aucun cas prévues pour remplacer les tests. Par exemple, il ne faut jamais utiliser les exceptions pour gérer un dépassement dans un tableau ou une liste (comme dans l'exemple 7.1), il faut utiliser des tests (comme dans l'exemple 7.2). Il s'agit en effet d'un comportement nominal du programme.

Exemple 7.1 (Mauvaise utilisation d'exceptions)**.** *Dans cet exemple, les exceptions sont utilisées en remplacement de tests. Il s'agit d'une très mauvaise stratégie.*

```
try :
    # Calcul de l'indice desire
    indice = calcul_indice_tableau()
    # Accès systématique à la liste : l'exception gère les dépassements
    x = tableau[indice]
except IndexError as err:
    pass
```

Exemple 7.2 (Remplacement de l'exception par un test)**.** *Cet exemple représente la bonne manière de traiter les problèmes de dépassement de tableau ou de liste.*

```
# Calcul de l'indice désiré
indice = calcul_indice_tableau()
# Test de validité de l'indice
if indice>=0 and indice<len(tableau):
    # Accès au tableau si l'indice est valide
    x = tableau[indice]
}
```

7.2.1 Déclenchement d'une exception

Il est possible de lever une exception dans une méthode afin de signaler une erreur d'exécution. Exemple :

```
>>> raise Exception('Attention !')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: Attention !
```

7.2.2 Définition de classes d'exception

Il est possible à l'utilisateur de définir ses propres classes d'exception. Il s'agit de définir une nouvelle classe répondant aux contraintes suivantes :

— la classe doit hériter de la classe *Exception* ;

— il est possible de redéfinir la méthode `__str__` pour personnaliser l'affichage de l'exception.

Le listing 7.7 est un exemple de classe d'exception définie par l'utilisateur.

Listing 7.7 – Classe d'exception

```
class MonException(Exception):
    """Exception définie par l'utilisateur.

    Attributes
    ----------
        valeur -- valeur de l'exception
    """
    def __init__(self, valeur):
        self.valeur = valeur

    def __str__(self):
        return repr(self.valeur)

try:
    raise(MonException(6*7))
except MonException as err:
    print('Exception : ', err)
```

7.3 Utilisation de fichiers

Un programme peut avoir besoin d'accéder à des données persistantes, c'est-à-dire des données qui ne sont pas stockées en mémoire vive et qui sont encore accessibles après un redémarrage de la machine. Ces données peuvent par exemple être stockées dans des fichiers (texte, image, ...) ou dans des bases de données. Cette partie décrit l'accès aux fichiers.

Afin de travailler sur les données contenues dans le fichier, le programme Python doit tout d'abord les transférer en mémoire vive. Après traitement, il peut transférer les données de la mémoire vers un fichier. Il existe quelques classes Python permettant d'effectuer ce transfert, en lecture ou en écriture. Consulter la documentation Python pour la liste des classes permettant des entrées-sorties.

Il existe deux types de fichiers : les fichiers texte et les fichiers binaires. Les fichiers texte comprennent les sources de programmes (fichiers *.py* par exemple), des textes (fichiers *.txt* sous windows), des fichiers de configuration (le *.bashrc* sous linux par exemple), les fichiers HTML, les sources LaTeX... Tous ces fichiers peuvent s'ouvrir avec un éditeur de texte (*emacs*, *vi*, ...).

Les fichiers binaires comprennent les programmes compilés (les *.pyc* Python par exemple), les images (*.jpg*, *.png*), les vidéos (*.mpg*, *.avi*), les documents libreoffice, excel, word (*.ods*, *.odt*, *.xls*, *.doc*), les fichiers compressés (*.gz*, *.bz2*, *.zip*, *.rar*), ... Attention, ces fichiers ne peuvent pas s'ouvrir avec un éditeur de texte : ils nécessitent un logiciel particulier.

7.3.1 Lecture de fichiers texte

La lecture de fichiers texte en Python est assez simple : après avoir ouvert le fichier grâce à l'instruction *open*, il est possible de lire un certain nombre de caractères avec l'instruction *read*, de lire toutes les lignes du fichier avec l'instruction *readlines*, ou de parcourir le fichier avec une boucle *for*.

Le nombre de fichiers pouvant être ouverts simultanément est limité, et il faut donc penser à fermer le fichier lorsqu'il ne sert plus (grâce à l'instruction *close*).

Le listing 7.8 illustre ces opérations.

Listing 7.8 – Lecture de fichier

```
import sys

filename = 'fichier.txt'

try:
    # Ouverture du fichier texte en lecture seule
    f = open(filename)
except IOError as e:
    # En cas d'exception, afficher un message d'erreur
    print('Ouverture du fichier impossible\n', e)
    # et quitter le programme
    sys.exit(1)

# Lecture du fichier, et mise du contenu dans la variable caracteres
caracteres = f.read()

print('Nombre de caractères du fichier :', len(caracteres))

# Retour au début du fichier
f.seek(0)

# Lecture de la première ligne
ligne = f.readline()

f.seek(0)

# Lire le fichier sous forme de liste de lignes
les_lignes = f.readlines()

print('Nombre de lignes :', len(les_lignes))

f.seek(0)

# Lecture du fichier ligne à ligne
for ligne in f:
    # Affiche chaque ligne du fichier à l'envers
    print(ligne[::-1], end='')

# Fermeture du fichier après lecture
f.close()
```

Depuis Python 2.5, il est possible de définir des contextes d'exécution. Toute variable créée dans un contexte est automatiquement supprimée à la fin du bloc. Dans le cas d'une ouverture de fichier, il est alors automatiquement fermé. L'ouverture d'un contexte se fait par le mot-clé *with*, comme montré dans le lising 7.9.

Listing 7.9 – Lecture de fichier dans un contexte

```
# Ouverture du fichier texte en lecture seule
with open('fichier.txt', encoding='latin1') as f:
    # Lecture de 128 caractères
    caracteres = f.read(128)

# Le f.close() est implicite à la fin du bloc
```

7.3.2 Écriture de fichiers texte

Pour écrire dans un fichier, il faut qu'il soit ouvert en écriture, c'est en dire en mode « w » [1] ou en mode « a » [2]. Dans le premier cas, le contenu du fichier est supprimé à l'ouverture, et dans le deuxième cas l'écriture se fait à la fin du fichier. Le listing 7.10 illustre ce fonctionnement.

Listing 7.10 – Lecture de fichier

```
import sys

filename = sys.argv[1]

# Ouverture du fichier texte en écriture
with open(filename, mode='w') as f: # w -> write
    for mot in [1, 'abc', (1, 2)]:
        f.write(str(mot))
        f.write('\n')

# Fermeture du fichier implicite

# Ouverture du fichier texte en écriture
with open(filename, mode='a') as f: # a -> append
    f.write('Suite du fichier...\n')
    # Écriture avec print
    print('Une nouvelle ligne du fichier', file=f)
```

Remarque : il ne faut pas oublier de fermer le fichier [3] ; sinon il peut se retrouver dans un état incohérent.

7.3.3 Lecture et écriture de fichiers binaires

Un fichier binaire ne contient pas de texte, mais un ensemble d'octets qui pourra être interprété comme des entiers, des réels, des caractères, ... Ces fichiers sont généralement plus compacts que des fichiers texte contenant les mêmes informations, mais il est nécessaire de connaître leur structure pour y accéder.

Le listing 7.11 représente un exemple d'utilisation de fichier binaire. L'instruction *write* permet d'écrire des octets dans un fichier, et l'instruction *read* permet de lire des octets depuis un fichier. On notera l'utilisation des fonctions *to_bytes* et *from_bytes* qui convertissent une variable entière en octets et inversement.

Listing 7.11 – Utilisation de fichier binaire

```
import sys

filename = sys.argv[1]

# Ouverture du fichier binaire en écriture
with open(filename, mode='wb') as f:  # w -> write, b->binary
    size = 10  # Nombre de données
    # Conversion de l'entier en liste d'octets en format "big endian"
    octets = size.to_bytes(4, 'big')
    # Écriture du nombre de données dans le fichier
    f.write(octets)
```

1. write
2. append
3. Soit explicitement avec un *f.close()*, soit implicitement en utilisant un contexte *with*.

```
12
13      debut = 10
14      for nb in range(debut, debut+size):
15          # Conversion du nombre en liste d'octets
16          # Conversion sur 4 octets
17          octets = nb.to_bytes(4, 'big')
18          f.write(octets)
19
20  # Ouverture du fichier binaire en lecture
21  with open(filename, mode='rb') as f:  # r -> read, b->binary
22      # Lecture du nombre de données
23      octets = f.read(4)
24      # Conversion du nombre de données en entier
25      nb = int.from_bytes(octets, 'big')
26      for _ in range(nb):
27          octets = f.read(4)
28          val = int.from_bytes(octets, 'big')
29          print(val)
```

7.3.4 Lecture et écriture de paquets binaires : utilisation de struct

Python permet de décoder simplement des données binaires organisées sous forme de paquets grâce au module *struct*. Lorsque l'on cherche à décoder un fichier binaire, le document de départ est la description du format de fichier (exemple : listing 7.12). Dans notre cas, la documentation indique également que le fichier est organisé au format « *little endian*[4] ».

Listing 7.12 – Exemple de format de fichier

```
1  struct datagram {
2    long size;
3    short channel;
4    short mode;
5    float transducer_depth; // [m]
6    float frequency; // [Hz]
7    float pulse_length; // [s]
8  }
```

Afin de décoder des données correspondant à ce format, se reporter à la documentation de *struct* : https://docs.python.org/3/library/struct.html. Comme les données sont au format *little endian*, le décodage (*unpack*) de données doit commencer par « < ».

Le listing 7.13 est un programme Python permettant de décoder le format défini précédemment.

Listing 7.13 – Décodage de fichier binaire (format décrit par le listing 7.12)

```
1  import struct
2
3  f = open(filename, 'rb')
4  data = f.read(4) # Lecture de size : 1 long -> 4 octets
5  nb, = struct.unpack('<l', data)
6  data = f.read(nb) # Lecture de nb éléments
7  # Décodage de data : 2 short et 3 float
8  c, m, d, f, pl = struct.unpack('<2s3f', data)
```

De manière similaire, l'écriture de fichier binaire peut se faire en utilisant la commande *struct.pack*.

4. c'est-à-dire que les octets sont rangés du poids faible vers le poids fort. Dans le format « *big endian* », les octets sont rangés du poids fort vers le poids faible.

7.3.5 Sérialisation

Le module *pickle* permet de sérialiser des données, c'est-à-dire qu'il permet de lire ou d'écrire des données complexes dans des fichiers binaires.

Le listing 7.14 est un exemple d'utilisation de *pickle* pour la sérialisation. La fonction *dump* sauvegarde la variable dans un fichier, et la fonction *load* restaure la variable depuis le fichier.

Listing 7.14 – Utilisation de pickle

```
import sys
import pickle

filename = sys.argv[1]

# Création de données quelconques
data = {
    'a': [1, 2.0, 3, 4+6j],
    'b': ("chaîne de caractères", b"liste d'octets"),
    'c': set([None, True, False])
}

# Ouverture du fichier en écriture binaire
with open(filename, 'wb') as f:
    # Utilisation de pickle pour écrire les données
    pickle.dump(data, f)

# Ouverture du fichier en lecture binaire
with open(filename, 'rb') as f:
    # Utilisation de pickle pour lire les données
    new_data = pickle.load(f)
```

7.4 Itérateurs et générateurs

7.4.1 Utilisation d'itérateurs

Python offre la possibilité de parcourir facilement les collections d'éléments grâce à des itérateurs. Le parcours s'effectue alors simplement avec une boucle *for*. Dans le listing 7.15, on peut voir que tous les parcours se font de manière unifiée.

Listing 7.15 – Exemple d'itérations

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for cle in {'one':1, 'two':2}:
    print(cle)
for char in "123":
    print(char)
for ligne in open('monFichier.txt'):
    print(ligne, end='')
```

7.4.2 Définition d'itérateurs

Définir une classe itérable est assez simple. Il suffit que la classe contienne deux méthodes :

— `__iter__` dont le rôle est d'initialiser l'itérateur ;
— `__next__` qui définit comment passer à l'itération suivante ; cette méthode lève l'exception *StopIteration* lorsque les itérations sont terminées.

Le listing 7.16 contient une classe itérateur permettant de décrire les valeurs prises par la suite de Fibonacci. La fonction `__iter__` initialise les valeurs initiales de la suite [5] et la fonction `__next__` définit le passage au terme suivant [6]. Ensuite, une simple boucle *for* permet de parcourir la suite de Fibonacci.

Listing 7.16 – Définition d'une classe itérateur

```
class Fibonacci:
    """Classe permettant de calculer les valeurs de la suite
    de Fibonacci. S'appuie sur une structure d'itérateur.

    Attributes
    ----------
    u_n2, u_n1 : int, int
        Termes :math:`u_{n-2}` et :math:`u_{n-1}` de la suite
        de Fibonacci
    nb : int
        Nombre d'itérations effectuées
    max : int
        Nombre d'itérations maximal
    """
    def __init__(self, max):
        """Initialisation de l'itérateur.

        Parameters
        ----------
        max : int
            Nombre d'itérations à effectuer
        """
        self.max = max

    def __iter__(self):
        """Pour être un itérateur, la classe doit posséder une méthode
        __iter__ dont la dernière instruction est "return self".
        Cette méthode initialise les itérations.
        """
        self.u_n2 = 0
        self.u_n1 = 1
        self.nb = 0
        return self

    def __next__(self):
        """Méthode définissant le passage à l'itération suivante.
        Si plus aucune itération n'est possible, la méthode lève
        l'exception "StopIteration".
        """
        self.nb += 1
        fib = self.u_n2
        # Test de fin d'itérations
        if self.nb > self.max:
            raise StopIteration
        # Passage au terme suivant de la suite de Fibonacci
```

5. $u_0 = 0, u_1 = 1$
6. $u_{n+2} = u_{n+1} + u_n$

```
46          self.u_n2, self.u_n1 = self.u_n1, self.u_n2 + self.u_n1
47          return fib
48
49  if __name__ == "__main__":
50      # Utilisation de l'itérateur grâce à la boucle for.
51      for f in Fibonacci(10):
52          print(f)
```

7.4.3 Définition de générateur

Python propose un mécanisme simplifié de création d'itérateurs : les générateurs. Un générateur est une méthode classique qui utilise l'opérateur *yield* chaque fois qu'elle doit retourner une valeur.

Le listing 7.17 reprend l'exemple de la suite de Fibonacci sous forme de générateur. Cette fois, tout le code définissant le comportement de la suite de Fibonacci (de l'initialisation aux itérations) est regroupé dans la même fonction. L'instruction *yield* indique la valeur à retourner à chaque itération.

Listing 7.17 – Définition d'un générateur

```
1  def fibonacci(n):
2      """Générateur de valeurs de la suite de Fibonacci.
3      """
4      # Variables u_{n-2} et u_{n-1}, par défaut u_0 et u_1
5      u_n2, u_n1 = 0, 1
6      i = 0
7      while i < n:
8          # La valeur à sortir pour l'itération courante est u_n2
9          yield u_n2
10         # Passage à l'itération suivante
11         u_n2, u_n1 = u_n1, u_n2 + u_n1
12         i += 1
13
14 for f in fibonacci(10):
15     print(f)
```

Exemple 7.3 (Générateur et liste chaînée)**.** *Il est assez simple de créer un générateur à partir d'une fonction de parcours de liste chaînée. À partir du listing 6.2 p. 135 on obtient le générateur suivant :*

Listing 7.18 – Générateur et liste chaînée

```
1  def parcours(self):
2      n = self.__first
3      while n is not None:
4          yield n
5          n = n.next
```

7.5 Optimisation des performances de Python

Python est un langage interprété et typé dynamiquement ce qui permet un développement très rapide. Ceci se paye par des performances très en deçà de celles des langages compilés tels que C, C++, Java, Fortran, ... Cette baisse de performances est particulièrement sensible lorsque

le programme comporte un grand nombre d'itérations. Une solution peut être d'apporter plus d'attention à l'algorithmique, et en particulier à la complexité du programme (voir chapitre D).

Une autre possibilité consiste à identifier les parties les plus lentes à l'aide d'un *profiler* et de les optimiser en les compilant, éventuellement avec de la compilation à la volée [7].

7.5.1 numba

Numba est un compilateur à la volée pour Python. L'opération, quasi-transparente pour le programmeur, permet d'accélérer le programme jusqu'à un facteur 200.

Considérons par exemple de programme du listing 7.19. La structure du programme Python n'est pas efficace avec une structure de boucles imbriquées. L'utilisation d'un langage interprété est très pénalisant.

Listing 7.19 – Fonction à accélérer

```
def boucle1(n):
    s = 0
    for i in range(n):
        for j in range(n):
            if i == j:
                s += i
    return s
```

Le listing 7.20 utilise *numba* pour accélérer le programme. Cette utilisation se résume à une importation du module *numba* et la décoration (avec le `@jit`) de la fonction à compiler à la volée.

Listing 7.20 – Utilisation de numba

```
from numba import jit

@jit
def boucle2(n):
    s = 0
    for i in range(n):
        for j in range(n):
            if i == j:
                s += i
    return s
```

Le listing 7.21 utilise les deux fonctions définies précédemment, et le listing 7.22 contient les résultats de l'exécution. Dans cet exemple, le gain en temps de calcul est d'un facteur 66.

Listing 7.21 – Test de numba

```
if __name__ == '__main__':
    n = 20000
    t1 = datetime.now()
    boucle1(n)
    t2 = datetime.now()
    boucle2(n)
    t3 = datetime.now()

    print('Sans numba : ', t2 - t1)
    print('Avec numba : ', t3 - t2)
```

7. Compilation *just in time* ou *jit* en anglais.

Listing 7.22 – Résultat du test

```
Sans numba :  0:00:16.829839
Avec numba :  0:00:00.253887
```

L'utilisation de *numba* est très simple, et le gain en performances est très appréciable. Notons toutefois que toutes les fonctionnalités de Python ne sont pas supportées par *numba*. Pour plus de détails, consulter la documentation : https://numba.pydata.org.

7.5.2 pypy

Pypy est une réimplantation de Python en Python. Cette version contient un compilateur à la volée, et elle est donc beaucoup plus rapide que la version installée par défaut. Elle se veut le plus exhaustive possible, mais ne possède pas toutes les fonctionnalités de Python. Consulter la documentation en ligne pour plus d'informations : https://pypy.org/.

La fonction du listing 7.19 s'exécute directement avec le code du listing 7.23. Le temps d'exécution, indiqué sur le listing 7.24 est proche de celui de *numba*. Le gain est ici d'un facteur 33 par rapport à la version non compilée à la volée.

Listing 7.23 – Test de pypy

```
if __name__ == '__main__':
    n = 20000
    t1 = datetime.now()
    boucle1(n)
    t2 = datetime.now()

    print('Avec pypy : ', t2 - t1)
```

Listing 7.24 – Résultat du test

```
Avec pypy :  0:00:00.508036
```

7.5.3 Cython

Cython permet de compiler une partie du programme Python afin d'accélérer son exécution. Il ne s'agit pas ici d'une compilation à la volée, et le programmeur doit donc exécuter une compilation explicite de son programme.

Pour que la compilation soit plus efficace, il est recommandé de typer statiquement les variables. Le listing 7.25 contient deux traductions (une sans typage et une avec) de la fonction du listing 7.19 en Cython. Le mot-clef `cdef` définit une variable comme ayant un type C statique, ou garantit le type de la valeur de retour d'une fonction. Le mot-clef `cpdef` définit la variable ou la fonction à la fois comme C et Python, permettant son utilisation dans l'un ou l'autre contexte, au prix d'un temps de calcul supplémentaire pour passer de l'un à l'autre. Notons que le fichier contenant ces fonctions doit avoir l'extension *.pyx*.

Listing 7.25 – Fonctions Cython : fichier *fonctions_boucles.pyx*

```
def boucle1(n):
    s = 0
    for i in range(n):
        for j in range(n):
            if i == j:
                s += i
    return s

```

```
cpdef int boucle2(int n):
    cdef int i, j, s
    s = 0
    for i in range(n):
        for j in range(n):
            if i == j:
                s += i
    return s
```

Afin de compiler le programme du listing 7.25, il faut créer le programme du listing 7.27 et utiliser l'instruction indiquée dans le listing 7.26.

Listing 7.26 – Compilation du programme Cython

```
python3 setup.py build_ext --inplace
```

Listing 7.27 – Fichier *setup.py*

```
from distutils.core import setup
from Cython.Build import cythonize

setup(
    ext_modules = cythonize("fonctions_boucles.pyx")
)
```

Le test du programme est similaire aux cas précédents. Le programme de test est le listing 7.28, et le résultat de l'exécution est le listing 7.29.

Listing 7.28 – Test du programme Cython compilé

```
from fonctions_boucles import boucle1, boucle2
from datetime import datetime

if __name__ == '__main__':
    n = 20000
    t1 = datetime.now()
    boucle1(n)
    t2 = datetime.now()
    boucle2(n)
    t3 = datetime.now()

    print('Sans typage : ', t2 - t1)
    print('Avec typage : ', t3 - t2)
```

Listing 7.29 – Résultat du test Cython

```
Sans typage :  0:00:11.772652
Avec typage :  0:00:00.239080
```

On constate que le gain de performance est particulièrement sensible lors du typage des variables. Cython permet donc un gain appréciable, mais le prix à payer est un alourdissement de la syntaxe. La documentation de Cython est disponible en ligne : https://cython.readthedocs.io/en/latest/index.html.

“Que faites-vous avec les sorcières ?
— Nous les brûlons.
— Et pourquoi les sorcières brûlent-elles ?
— Parce qu’elles sont faites de bois.
— Et comment sait-on qu’elles sont faites de bois ?
— Parce qu’elles flottent sur l’eau.
— Et quoi d’autre flotte sur l’eau ?
— Un canard.
— Et donc, si elle pèse autant qu’un canard, c’est une sorcière.”

The Monty Python, *The Quest for The Holy Grail*

8

Quelques modules python

Sommaire

Savoir développer un programme de a à z est important, mais il est tout aussi important de savoir réutiliser des programmes existants. En particulier, Python fournit une API très conséquente qu'il serait dommage d'ignorer.

Ce chapitre décrit brièvement quelques modules importants de l'API python. La documentation complète du langage, y compris les API *standard* se trouve sur https://docs.python.org/3/. La documentation des autres API se trouve en général sur le site dédié, avec son code source.

8.1 String

Le module *string* contient un ensemble de chaînes de caractères constantes qui peuvent simplifier l'exploitation de données textuelles. Il contient entre autres les constantes suivantes :

whitespace : tous les caractères blancs de la table ASCII ;

ascii_lowercase : toutes les lettres minuscules de la table ASCII ;

ascii_uppercase : toutes les lettres majuscules de la table ASCII ;

ascii_letters : combinaison de ascii_lowercase et ascii_uppercase ;

digits : tous les chiffres décimaux ;

hexdigits : tous les chiffres hexadécimaux ;

octdigits : tous les chiffres octaux ;

punctuation : tous les caractères de ponctuation ;

printable : tous les caractères affichables (regroupe toutes les catégories ci-dessus).

Le listing 8.1 reprend quelques exemples d'utilisation du module *string*.

Listing 8.1 – Exemple d'utilisation du module *string*

```
>>> import string
>>> print(string.ascii_lowercase + string.digits)
abcdefghijklmnopqrstuvwxyz0123456789
>>> [c for c in string.hexdigits]
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e
  ', 'f', 'A', 'B', 'C', 'D', 'E', 'F']
```

8.1.1 classe Formatter

Le module *string* contient également la classe *Formatter* qui possède la même syntaxe que la méthode *str.format*. Nous donnons dans cette section un aperçu du formatage de chaînes de caractères. Pour aller plus loin et utiliser les options de formatage avancées, consulter la documentation en ligne de Python [1].

La classe *Formatter* permet de construire des chaînes de caractères dans lesquelles seront insérées des valeurs quelconques (nombres, texte, ...).

1. https://docs.python.org/3/library/string.html

Le listing 8.2 représente cette situation : on cherche à afficher le message « Factorielle i vaut f. » avec i allant de 1 à 9 et f valant i!. Dans cette exemple, les endroits où il faut insérer une valeur sont représentés par des accolades ({}). Le premier paramètre de la fonction *format* remplace la première paire d'accolades, le deuxième paramètre la deuxième paire, ...

Listing 8.2 – Exemple de formatage de chaîne de caractères

```
fact = 1
for i in range(1, 10):
    fact *= i
    print('Factorielle {} vaut {}'.format(i, fact))
```

L'instruction *format* s'adapte à tous les types de paramètres. Par exemple, le listing 8.3 utilise un paramètre de type *str* et un paramètre de type *int*.

Listing 8.3 – Formatage de valeurs de différents types

```
mot = 'réponse'
valeur = 42
'La {} est {}'.format(mot, valeur)
```

Comme le montre le listing 8.4, il est possible de préciser le numéro du paramètre à utiliser entre les accolades. Le même paramètre peut ainsi être utilisé plusieurs fois. Par exemple, le paramètre 1 et le paramètre 2 sont utilisés deux fois chacun.

Listing 8.4 – Formatage et utilisation de l'ordre des paramètres

```
'{1}{3}{2}{1}{0}{5}{2}{4}'.format('N', 'PA', 'E ','S D', '!', 'IQU')
```

Il est possible de définir une taille de message à afficher et un alignement. Le listing 8.5 illustre cette possibilité.

Listing 8.5 – Alignement des chaînes de caractères

```
>>> '{:<20}'.format('gauche')
'gauche              '
>>> '{:>20}'.format('droite')
'               droite'
>>> '{:^20}'.format('centre')
'       centre       '
```

8.2 Interactions avec l'interpéteur Python : sys

Ce module donne accès à un ensemble de variables et de fonctions qui permettent d'interagir avec l'interprète Python.

8.2.1 argv

La variable *argv* est un tableau contenant le nom du programme exécuté et la liste des paramètres passés au programme. Dans le listing 8.6, le programme affiche la valeur de *argv*. Lors du test, on passe différents arguments à ce programme. Chaque mot est un paramètre, et les guillemets permettent de regrouper plusieurs mots dans le même paramètre.

Listing 8.6 – Utilisation de argv

```
$ cat test_argv.py
import sys

```

```
print(sys.argv)
$ python3 test_argv.py test "avec des" arguments 1 2 "3 4"
['test_argv.py', 'test', 'avec des', 'arguments', '1', '2', '3 4']
```

8.2.2 exit

La fonction *exit* permet de sortir d'un programme Python. Le paramètre passé à la fonction peut être récupéré dans le *shell* ayant lancé la commande, comme l'illustre le listing 8.7.

Listing 8.7 – Utilisation de exit

```
$ python3
Python 3.4.2 (default, Oct  8 2014, 10:45:20)
[GCC 4.9.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.exit(42)
$ echo $?
42
```

8.2.3 float_info, int_info

Les variables *float_info* et *int_info* contiennent les informations relatives au codage des nombres réels et entiers.

Listing 8.8 – float_info et int_info

```
>>> sys.float_info
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308,
min=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15,
mant_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)
>>> sys.float_info.epsilon
2.220446049250313e-16
>>> sys.int_info
sys.int_info(bits_per_digit=30, sizeof_digit=4)
```

8.2.4 stdin, stdout et stderr

Il s'agit de trois pseudo fichiers utilisés pour lire des entrées au clavier, pour écrire des messages standards ou pour écrire des messages d'erreur. Ainsi, la fonction *input()* va lire un message dans *stdin* et la fonction *print()* va écrire un message dans *stdout*. *stderr* sera par exemple utilisé pour les exceptions.

Le listing 8.9 contient un exemple de redéfinition de *sys.stdout*.

Listing 8.9 – Redéfinition de *stdout*

```
import sys

# Création d'un fichier de sortie
f = open('fichier.out', 'w')
# Sauvegarde de l'ancienne valeur de stdout
old_stdout = sys.stdout
# Remplacement de stdout par le fichier
sys.stdout = f

```

```
# À partir de maintenant, les affichages se font dans le fichier

print('Hello !')
print(list(range(10)))
# stderr n'a pas été redéfini : ce message s'affiche dans le terminal
print('Erreur', file=sys.stderr)
# Les exceptions s'affichent également dans le terminal
# raise Exception("Message d'erreur !")

# restoration du stdout initial
sys.stdout = old_stdout
f.close()
```

8.2.5 Taille de la pile d'appel

La fonction *sys.getrecursionlimit()* indique la taille maximale de la pile d'appel. La fonction *sys.setrecursionlimit(limit)* permet de modifier cette valeur.

Remarque : les cas dans lesquels il est intéressant de modifier la taille de la pile sont très rares. La valeur par défaut est presque toujours satisfaisante.

Listing 8.10 – Accès à la taille de la pile d'appel

```
>>> sys.getrecursionlimit()
1000
```

8.2.6 Accès au prompt

Il est possible de redéfinir le prompt de l'interprète python grâce aux variables *sys.ps1* et *sys.ps2*. Comme illustré dans le listing 8.11, la valeur par défaut de *ps1* est `'>>> '`, et la valeur par défaut de *ps2* est `'... '`.

Listing 8.11 – Modification du prompt dans python3

```
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1='%%% '
%%% sys.ps2 = '___ '
%%% def f():
___     pass
%%%
```

La gestion du prompt dans l'interprète ipython est similaire. Dans ce cas, *ps1* vaut `'In : '`, *ps2* vaut `'...: '`, et il existe une variable *ps3* pour les sorties qui vaut `'Out: '`.

Listing 8.12 – Modification du prompt dans ipython3

```
In [1]: import sys

In [2]: sys.ps1
Out[2]: 'In : '

In [3]: sys.ps2
Out[3]: '...: '
```

```
8
9  In [4]: sys.ps3
10 Out[4]: 'Out: '
```

8.2.7 Informations générales

Il existe quelques variables et fonctions qui permettent de se renseigner sur la version de l'interprète Python. Elles permettent de s'assurer que la version de Python utilisée correspond bien aux attentes du programmeur. Par exemple, pour s'assurer que la version de Python utilisée est au moins la version 3.3, on pourra écrire le listing 8.13.

Listing 8.13 – Test de la version de Python

```
1 if (sys.version_info.major==3 and sys.version_info.minor>=3) or sys.
    version_info.major>3:
2   ...
```

Le listing 8.14 recense quelques variables et fonctions de *sys* fournissant des informations utiles sur la version de l'interprète Python utilisé.

Remarque : il est toujours préférable que le programme Python soit le plus générique possible et qu'il fonctionne de la même manière quelque soit le système utilisé [2].

Listing 8.14 – Informations sur l'environnement

```
1  >>> sys.version
2  '3.4.2 (default, Oct  8 2014, 10:45:20) \n[GCC 4.9.1]'
3  >>> sys.version_info
4  sys.version_info(major=3, minor=4, micro=2, releaselevel='final', serial
     =0)
5  >>> sys.version_info.major
6  3
7  >>> sys.platform
8  'linux'
9  >>> sys.getdefaultencoding()
10 'utf-8'
11 >>> sys.getfilesystemencoding()
12 'utf-8'
13 >>> sys.implementation
14 namespace(_multiarch='x86_64-linux-gnu', cache_tag='cpython-34',
     hexversion=50594544, name='cpython', version=sys.version_info(major=3,
     minor=4, micro=2, releaselevel='final', serial=0))
```

8.3 Gestion de la date et de l'heure : datetime

Le module *datatime* permet de manipuler simplement des dates et des heures. Ce module contient 6 classes : *date*, *datetime*, *time*, *timedelta*, *timezone* et *tzinfo*. Nous décrivons ici brièvement les classes *datetime* et *timedelta*. La documentation complète de ce module est disponible en ligne [3].

date est une classe contenant la date [4] qui s'appuie sur le calendrier Grégorien.

2. On rappelle toutefois que les versions 2 et 3 de Python ne sont pas compatibles.
3. https://docs.python.org/3/library/datetime.html
4. année, mois et jour.

datetime est une classe contenant à la fois la date et l'heure qui s'appuie également sur le calendrier Grégorien. Elle suppose que chaque jour dure exactement $24 * 60 * 60$ secondes [5].

time est une classe contenant l'heure du jour. Contrairement aux classes *date* et *datetime*, il n'est pas possible de calculer la différence de deux objets de type *time* ; on préférera donc généralement *datetime* à *time*.

timedelta est une classe qui représente la différence entre deux dates ou deux heures. On peut ajouter ou retrancher une valeur de type *timedelta* à un objet de type *datetime* ou *date*. Cette classe s'associe particulièrement bien à la classe *datetime*.

8.3.1 timedelta

La classe *timedelta* permet de manipuler très simplement et intuitivement les différences de dates. La syntaxe générale pour créer un objet de type *timedelta* est :

```
timedelta(days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0,
  hours=0, weeks=0)
```

Dans la représentation interne de *timedelta* les données sont converties en jours [6], secondes [7] et microsecondes [8].

La classe *timedelta* possède trois constantes :

min : l'écart de temps de plus négatif qui vaut `timedelta(-999999999)` ;

max : l'écart de temps de plus positif qui vaut `timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)` ;

resolution : le plus petit écart de temps qui vaut `timedelta(1)`.

Le listing 8.15 contient quelques exemples d'utilisation de la classe *timedelta*.

Listing 8.15 – Exemple d'utilisation de *timedelta*

```
>>> from datetime import timedelta
>>> timedelta(weeks=2, minutes=5)
datetime.timedelta(14, 300)
>>> timedelta.max
datetime.timedelta(999999999, 86399, 999999)
>>> timedelta.max + timedelta.resolution
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: days=1000000000; must have magnitude <= 999999999
```

La table 8.1 décrit les opérations supportées par les objets de type *timedelta*. Les variables `t1`, `t2` et `t3` sont de type *timedelta*, `f` est de type *float*, et `i` est de type *int*.

TABLE 8.1: Opérations supportées par *timedelta*

Opération	Résultat
`t1 = t2 + t3`	Somme de t2 et t3
`t1 = t2 - t3`	Différence entre t2 et t3
`t1 = t2 * i` ou `t1 = i * t2`	Produit d'un *timedelta* par un entier.
...	

5. Cette hypothèse est très souvent vraie, mais le 2 juillet 2015 a duré 86401s.
6. $-999999999 \leqslant j \leqslant 999999999$
7. $0 \leqslant s < 24 * 60 * 60$
8. $0 \leqslant \mu s < 1000000$

...	
`t1 = t2 * f` ou `t1 = f * t2`	Produit d'un *timedelta* par un réel. Le résultat est arrondi à la valeur la plus proche.
`f = t2 / t3`	Quotient entre deux *timedelta*. Le résultat est un réel.
`t1 = t2 / f` ou `t1 = t2 / i`	Quotient entre un *timedelta* et un entier ou un réel. Le résultat est arrondi à la valeur la plus proche.
`t1 = t2 // i`	Quotient entier entre un *timedelta* et un entier.
`i = t1 // t2`	Quotient entier entre deux *timedelta*.
`t1 = t2 % t3`	reste de la division entre deux *timedelta*.
`q,r= divmod(t1, t2)`	Quotient et reste de la division entre deux *timedelta*. `q` est un entier et `r` est un *timedelta*.
`+t1`	Retourne un *timedelta* identique à `t1`.
`-t1`	Retourne l'opposé de `t1`. Équivalent à `timedelta(-t1.days, -t1.seconds, -t1.microseconds)`, et à `t1* -1`.
`abs(t1)`	Équivalent à `+t` si `t.days >= 0`, et à `-t` si `t.days < 0`.

8.3.2 datetime

La classe *datetime* permet de manipuler des dates et des heures. La syntaxe générale pour créer un objet de ce type est :

```
datetime(year, month, day, hour=0, minute=0, second=0, microsecond=0,
  tzinfo=None)
```

Avec :

- MINYEAR $\leqslant$ year $\leqslant$ MAXYEAR
- $1 \leqslant$ month $\leqslant 12$
- $1 \leqslant$ day $\leqslant$ nombre de jours du mois et de l'année donnés
- $0 \leqslant$ hour < 24
- $0 \leqslant$ minute < 60
- $0 \leqslant$ second < 60
- $0 \leqslant$ microsecond < 1000000

Le listing 8.16 contient quelques exemples d'utilisation de la classe *datetime*.

Listing 8.16 – Exemple d'utilisation de *datetime*

```
>>> from datetime import datetime, MINYEAR, MAXYEAR
>>> print(MINYEAR, MAXYEAR)
1 9999
>>> datetime.now()
datetime.datetime(2015, 7, 15, 11, 29, 24, 681162)
>>> datetime(year=2015, month=1, day=1)
datetime.datetime(2015, 1, 1, 0, 0)
>>> datetime.now().replace(year=42)
datetime.datetime(42, 7, 15, 11, 30, 3, 205719)
```

La table 8.2 recence les opérations disponibles avec des objets de type *datetime*. Les variables *dt1* et *dt2* sont de type *datetime*, et la variable *td* est de type *timedelta*.

TABLE 8.2: Opérations supportées par *datetime*

Opération	**Résultat**
`dt2 = dt1 + td`	Décale *dt1* de l'intervalle *td*
`dt2 = dt1 - td`	Décale *dt1* de l'opposé de l'intervalle *td*
...	

	...
`td = dt1 - dt2`	La différence de deux dates est une valeur de type *timedelta*.
`dt1 < dt2`	Il est possible de connaître l'antériorité d'une date par rapport à l'autre.

strftime et strptime

La classe *datetime* possède deux méthodes complémentaires permettant de gérer le format des dates :

strftime permet de convertir un objet de type *datetime* en chaîne de caractères ;

strptime permet de convertir une chaîne de caractères en objet de type *datetime*.

Un extrait des codes utilisables par ces fonctions est représenté dans la table 8.3 ; pour la liste complète des codes, consulter la documentation du module *datetime* de Python : https://docs.python.org/3.6/library/datetime.html#strftime-strptime-behavior.

TABLE 8.3: Format de *strptime* et *strftime*

Code	Signification
%d	Jour du mois sur 2 chiffres (de 01 à 31)
%m	Mois de l'année sur 2 chiffres (entre 01 et 12)
%y	Deux derniers chiffres de l'année
%Y	Année sur 4 chiffres
%H	Heure du jour sur 2 chiffres (de 00 à 23)
%M	Minutes sur 2 chiffres (entre 00 et 59)
%S	Secondes sur 2 chiffres (entre 00 et 59)
%f	Microsecondes sur 6 chiffres (entre 000000 et 999999)

8.3.3 Exemple

Le listing 8.17 illustre le fonctionnement des classes *datetime* et *timedelta*.

Listing 8.17 – Utilisation de datetime et timedelta

```
>>> from datetime import datetime, timedelta
# Date et heure courante en heure UTC
>>> datetime.utcnow()
datetime.datetime(2015, 7, 9, 6, 46, 15, 789037)
# Date et heure courante en heure locale
>>> d0 = datetime.now()
>>> print(d0)
2015-07-09 08:46:20.532864
# La différence de deux datetime est de type timedelta
>>> dt = d0 - datetime(2015,1,1,10,42,36)
>>> print(dt, type(dt))
188 days, 22:03:44.532864 <class 'datetime.timedelta'>
# Durée totale de dt exprimée en secondes
>>> print(dt.total_seconds())
16322624.532864
>>>
>>> nj = 3  # Nombre de jours
>>> nh = 2  # Nombre d'heures
# On peut ajouter ou retrancher un timedelta à un datetime
>>> d1 = d0 - timedelta(days=nj, hours=nh)
```

```
>>> print('Dans {0} jours et {1} heures, nous serons le : {2}'.format(nj,
    nh, d1))
Dans 3 jours et 2 heures, nous serons le : 2015-07-06 06:46:20.532864
# Il est possible de créer un objet de type datetime à partir
# d'une chaîne de caractères
>>> d2 = datetime.strptime('2014-04-01 10:04:20', '%Y-%m-%d %H:%M:%S')
# Utilisation de format pour l'affichage de la date
>>> print("Le {0:%d} {0:%B} de l'année {0:%Y} est un {0:%A}.".format(d1))
Le 06 juillet de l'année 2015 est un lundi.
```

8.4 Unittest et le test logiciel

Il est très difficile d'écrire un logiciel exempt de bug. Plus le logiciel est complexe, plus la présence de bug est probable. Une stratégie pour limiter les bugs consiste à tester le logiciel.

Une approche systématique du test présente plusieurs avantages :

— le développeur peut s'assurer à chaque instant du bon fonctionnement de son logiciel ; il évite ainsi le stress de dernière minute ;

— le développeur n'hésite pas à modifier et optimiser les différentes fonctions du logiciel : les tests permettent de s'assurer de la non régression [9] du logiciel ;

— le produit final est de meilleure qualité.

Parmi les types de test, on distinguera les tests unitaires [10], les tests d'intégration [11] et les tests systèmes [12]. Dans cette partie, nous parlerons uniquement des tests unitaires.

8.4.1 Le test unitaire

Le test unitaire est une procédure permettant de repérer les erreurs d'un programme au travers de scénarios de test, également appelés *cas de test*. Le principe d'un cas de test est d'initialiser des variables, d'appeler une fonction de traitement, et de comparer son résultat à un résultat attendu.

L'utilisation de tests unitaires permet d'augmenter la fiabilité d'un logiciel, en particulier s'il est soumis à de nombreuses modifications. Les tests sont au cœur des méthodes de développement rapide comme *l'extreme programming*.

Ils sont également fondamentaux pour les logiciels critiques, comme par exemple en avionique. Ainsi, la norme DO-178C [DO11] requiert des tests unitaires avec couverture totale du code pour tout le logiciel critique [13].

Remarque : un test unitaire permet de prouver la présence de bugs, mais jamais leur absence [14].

8.4.2 Test unitaire et programmation orientée objet

Dans le cas de la programmation orientée objet, les tests unitaires s'écrivent dans des classes de test. En général, une classe de test est associée à chaque classe à tester. Dans ce cas, il faut s'assurer que la classe soit testable, c'est-à-dire qu'il y ait toujours possibilité de vérifier la justesse du résultat. Il est alors souvent nécessaire d'avoir accès à la représentation interne des objets, autrement dit à leurs variables d'instance.

9. c'est-à-dire que ce qui fonctionnait avant modification fonctionne toujours.
10. Chaque module est testé indépendamment des autres.
11. On teste l'interaction entre différents modules
12. On teste la totalité du système.
13. Logiciel dont la panne peut avoir des conséquences majeures, dangereuses ou catastrophiques.
14. Citation d'Edsger Dijkstra : « Program testing can be used to show the presence of bugs, but never to show their absence! »

8.4.3 Utilisation de *unittest*

Unittest est un *framawork* de tests unitaires très simple à mettre en œuvre. Les principe est d'écrire une classe héritant de `unittest.TestCase` par classe à tester. Cette classe doit contenir un ensemble de méthodes dont le nom commence par *test*. Ces méthodes sont automatiquement appelées par la commande `unittest.main()`.

Chaque méthode fait appel à des assertions chargées de vérifier le résultat des méthodes de la classe testée. Si toutes les assertions sont satisfaites, le test est concluant ; sinon, la classe testée contient des erreurs.

Méthode	Assertion testée
`assertEqual(a, b)`	`a == b`
`assertNotEqual(a, b)`	`a != b`
`assertTrue(x)`	`bool(x) is True`
`assertFalse(x)`	`bool(x) is False`
`assertIs(a, b)`	`a is b`
`assertIsNot(a, b)`	`a is not b`
`assertIsNone(x)`	`x is None`
`assertIsNotNone(x)`	`x is not None`
`assertIn(a, b)`	`a in b`
`assertNotIn(a, b)`	`a not in b`
`assertIsInstance(a, b)`	`isinstance(a, b)`
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`

TABLE 8.4 – Liste des assertions

Les fonctions de *unittest* sont décrites dans la documentation de Python : https://docs.python.org/3.5/library/unittest.html.

La liste des assertions qu'il est possible de tester est définie dans la table 8.4. À cette liste, il faut ajouter la fonction `assertRaises()` qui permet de vérifier qu'une exception est levée dans la méthode.

8.4.4 Couverture de code

Les normes concernant les logiciels critiques imposent de tester les logiciels ; de plus, selon le niveau de criticité du logiciel, une couverture du code testé peut être imposée. Par exemple, en avionique, 5 niveaux de criticité sont définis :

Niveau A : un défaut du système ou sous-système étudié peut provoquer un problème catastrophique – sécurité du vol ou atterrissage compromis – crash de l'avion ;

Niveau B : un défaut du système ou sous-système étudié peut provoquer un problème majeur entraînant des dégâts sérieux voire la mort de quelques occupants ;

Niveau C : un défaut du système ou sous-système étudié peut provoquer un problème sérieux entraînant un dysfonctionnement des équipements vitaux de l'appareil ;

Niveau D : un défaut du système ou sous-système étudié peut provoquer un problème mineur réduisant légèrement la sécurité du vol ;

Niveau E : un défaut du système ou sous-système étudié peut provoquer un problème sans effet sur la sécurité du vol.

Les contraintes sur le logiciel dépendent de son niveau de criticité. Les contraintes concernant le test et la couverture de logiciel sont les suivantes :

Niveau E : aucune contrainte ;

Niveau D : le code doit être testé, et une couverture fonctionnelle est requise : toutes les fonctions doivent être testées ;

Niveau C : la couverture structurelle du code est requise : chaque instruction doit être couverte par le test ;

Niveau B : la couverture de code niveau « décision » est requise : dans chaque test, chaque condition doit prendre toutes les valeurs possibles ;

Niveau A : la couverture de code niveau « condition / décision » est requise : chaque sous-condition de chaque test doit prendre toutes les valeurs possibles.

Listing 8.18 – Condition complexe

```
1 if (a or b) and c:
```

Pour illustrer la différence entre les niveaux A et B, considérons le listing 8.18. Pour le niveau B, il faut que l'expression booléenne `(a or b) and c` prenne les deux valeurs possibles. On peut par exemple donner à a, b et c les valeurs :

— a=TRUE, b=TRUE, c=TRUE
— a=FALSE, b=FALSE, c=FALSE

Pour le niveau A, chaque sous-condition doit prendre toutes les valeurs possibles. Quatre configurations sont alors nécessaires :

— a=FALSE, b=FALSE, c=TRUE
— a=TRUE, b=FALSE, c=TRUE
— a=FALSE, b=TRUE, c=TRUE
— a=FALSE, b=TRUE, c=FALSE

En Python, le module *coverage*[15] permet de vérifier la couverture structurelle du code.

Remarque : la couverture structurelle de code à 100% n'est demandée que pour les logiciels critiques. Dans le cas général, on ne teste que les fonctions intéressantes. Ainsi, dans la section 8.4.5, la méthode `__str__` n'est pas testée.

8.4.5 Exemple

Reprenons l'exemple du listing 3.12 (section 3.3.1 p. 81). Remarquons que la classe n'est pas facilement testable en l'état : il n'y a pas d'accès immédiat aux variables d'instance. Pour rendre le code facilement testable, il faut ajouter les accesseurs ; on obtient alors la classe du listing 8.19.

Listing 8.19 – Classe Rationnel testable

```
1  class Rationnel:
2      """Type abstrait nombre rationnel.
3      """
4
5      def __init__(self, num, den=1):
6          """Creation d'un rationnel.
7          Si le dénominateur est nul, alors le rationnel est non défini.
8          On s'assure lors de la création que le dénominateur soit positif.
9          """
10         if den == 0:
11             raise Exception('Invalid number')
12         elif den < 0:
13             self.__num = -num
14             self.__den = -den
15         else:
```

15. https://pypi.python.org/pypi/coverage

```
16              self.__num = num
17              self.__den = den
18
19      @property
20      def num(self):
21          return self.__num
22
23      @property
24      def den(self):
25          return self.__den
26
27      def __eq__(self, other):
28
29          """Test d'égalité entre deux rationnels.
30          """
31          return self.__num * other.__den == other.__num * self.__den
32
33      def __add__(self, other):
34          """Calcule la somme de deux rationnels.
35          Le résultat est un rationnel.
36          """
37          n = self.__num * other.__den + self.__den * other.__num
38          d = self.__den * other.__den
39          return Rationnel(n, d)
40
41      def __mul__(self, other):
42          """Calcule le produit de deux rationnels.
43          Le résultat est un rationnel.
44          """
45          n = self.__num * other.__num
46          d = self.__den * other.__den
47          return Rationnel(n, d)
48
49      def __str__(self):
50          """Représentation textuelle d'un rationnel.
51          Permet de les afficher simplement.
52          """
53          return '{0}/{1}'.format(self.__num, self.__den)
54
55  if __name__ == '__main__':
56      # Test : création de rationnels
57      r1 = Rationnel(1, -2)
58      r2 = Rationnel(3, 5)
59      # Affichage des rationnels et de leur produit
60      print(r1, r2, r1*r2)
61      # Test d'égalité de rationnels
62      print(r1 == Rationnel(-2, 4))
```

Le principe du test est de créer une classe *TestRationnel* effectuant les tests unitaires de la classe *Rationnel*. À chaque méthode de *Rationnel* correspond au moins une méthode de test. Le listing 8.20 est un exemple de test de la classe *Rationnel*. Le listing 8.21 contient le résultat de son exécution dans le cas où la classe *Rationnel* ne contient pas d'erreur, et le listing 8.22 correspond au cas où la méthode `__add__` contient une erreur.

Listing 8.20 – Tests unitaires de la classe Rationnel

```
1  import unittest
```

```
2  from rationnel import Rationnel
3
4  class TestRationnel(unittest.TestCase):
5      """Classe de test unitaire. Toutes les méthodes dont le nom
6      commence par test sont exécutées automatiquement."""
7      def setUp(self):
8          """Méthode d'initialisation appelée avant chaque test.
9          Initialise quelques variables utilisables dans toutes les mé
             thodes.
10         """
11         self.__zero = Rationnel(0)
12         self.__one = Rationnel(1)
13         self.__two = Rationnel(2)
14
15     def testInit(self):
16         """Test de la méthode d'initialisation.
17         """
18         r = Rationnel(1, -1)
19         self.assertEqual(r.num, -1)
20         self.assertEqual(r.den, 1)
21
22     def testInitUndef(self):
23         """Test de l'initialisation de rationnel non défini.
24         """
25         with self.assertRaises(Exception):
26             r = Rationnel(1, 0)
27
28     def testEq(self):
29         """Test de l'égalité entre deux rationnels.
30         """
31         for num in range(-3, 4):
32             for den in range(1, 3):
33                 r1 = Rationnel(num, den)
34                 r2 = Rationnel(2*num, 2*den)
35                 self.assertEqual(r1, r2)
36
37     def testMult0(self):
38         """Test du produit d'un rationnel par 0.
39         """
40         for num in range(-3, 4):
41             for den in range(1, 3):
42                 r1 = Rationnel(num, den)
43                 r2 = r1 * self.__zero
44                 self.assertEqual(r2, self.__zero)
45
46     def testMult1(self):
47         """Test d'un produit d'un rationnel par 1.
48         """
49         for num in range(-3, 4):
50             for den in range(1, 3):
51                 r1 = Rationnel(num, den)
52                 r2 = r1 * self.__one
53                 self.assertEqual(r2, r1)
54
55     def testAdd(self):
56         """Tets de la somme de deux rationnels.
```

```
57          """
58          r1 = Rationnel(1, 2)
59          r2 = Rationnel(-1, 3)
60          r3 = Rationnel(1, 6)
61          self.assertEqual(r1+r2, r3)
62
63      def testAddMult(self):
64          """Comparaison de la somme de deux rationnels et
65          du produit d'un raionnel par 2.
66          """
67          for num in range(-3, 4):
68              for den in range(1, 3):
69                  r1 = Rationnel(num, den)
70                  r2 = r1 + r1
71                  r3 = r1 * self.__two
72                  self.assertEqual(r2, r3)
73
74  if __name__ == '__main__':
75      unittest.main()
```

Listing 8.21 – Exécution du test

```
1  $ python3 test_rationnel.py
2  .......
3  ----------------------------------------------------------------------
4  Ran 7 tests in 0.001s
5
6  OK
```

Listing 8.22 – Erreurs dans l'exécution du test

```
1  $ python3 test_rationnel.py
2  FF.....
3  ======================================================================
4  FAIL: testAdd (__main__.TestRationnel)
5  ----------------------------------------------------------------------
6  Traceback (most recent call last):
7    File "test_rationnel.py", line 47, in testAdd
8      self.assertEqual(r1+r2, r3)
9  AssertionError: <rationnel.Rationnel object at 0x7f92b7a831d0> != <
     rationnel.Rationnel object at 0x7f92b7a832e8>
10
11 ======================================================================
12 FAIL: testAddMult (__main__.TestRationnel)
13 ----------------------------------------------------------------------
14 Traceback (most recent call last):
15   File "test_rationnel.py", line 55, in testAddMult
16     self.assertEqual(r2, r3)
17 AssertionError: <rationnel.Rationnel object at 0x7f92b7a83208> != <
     rationnel.Rationnel object at 0x7f92b7a83128>
18
19 ----------------------------------------------------------------------
20 Ran 7 tests in 0.001s
21
22 FAILED (failures=2)
```

Le test de couverture de code est illustré par le listing 8.23. On remarque que la couverture n'est pas de 100% : la méthode `__str__` et le *main* ne sont pas testés. Il est possible d'indiquer à *coverage* qu'il ne faut pas prendre en compte le *main* en créant un fichier de configuration *.coveragerc* (voir listing 8.24) au même endroit que le code à tester. En ce qui concerne la méthode `__str__`, on peut considérer qu'elle n'est pas intéressante à tester.

Listing 8.23 – Test de couverture

```
$ coverage run test_rationnel.py
.......
----------------------------------------------------------------------
Ran 7 tests in 0.001s

OK
$ coverage report
Name              Stmts   Miss  Cover
-----------------------------------
rationnel            30      5    83%
test_rationnel       46      0   100%
-----------------------------------
TOTAL                76      5    93%
$ coverage report -m
Name              Stmts   Miss  Cover   Missing
-----------------------------------------------
rationnel            30      5    83%   53, 57-62
test_rationnel       46      0   100%
-----------------------------------------------
TOTAL                76      5    93%
$
```

Listing 8.24 – Fichier *.coveragerc*

```
[report]

exclude_lines =
    if __name__ == .__main__.:
```

8.5 Classes abstraites : abc

Le module *abc* fournit l'environnement nécessaire pour la définition de classes de base abstraites [16]. Il se conforme aux spécifications définies dans [VT18]. Il est généralement incontournable lors de l'application de motifs de conception (voir section 5.4 p. 120). Il contient deux classes et plusieurs décorateurs.

8.5.1 Classe *ABCMeta*

Cette métaclasse permet de créer des classes abstraites. L'intérêt principal de cette métaclasse est de donner accès aux décorateurs décrits dans la section 8.5.3.

Listing 8.25 – Utilisation d'*ABCMeta*

```
from abc import ABCMeta

class MaClasse(metaclass=ABCMeta):
```

16. En anglais *Abstract Base Classes*.

```
4       pass
```

8.5.2 Classe *ABC*

Cette classe est disponible depuis la version 3.4 de Python. Il s'agit d'une classe vide qui s'appuie sur la métaclasse *ABCMeta*. Elle permet de remplacer l'écriture du listing 8.25 par celle du listing 8.26.

Listing 8.26 – Utilisation d'*ABC*

```
1 from abc import ABC
2
3 class MaClasse(ABC):
4     pass
```

8.5.3 Décorateurs

Les décorateurs définis dans le module *abc* sont :

abstractmethod : signale que la méthode est abstraite. Il n'est pas possible d'instancier une classe dans laquelle au moins une méthode est abstraite.

abstractclassmethod : obsolète depuis la version 3.3 de Python ; il est maintenant possible d'utiliser le double décorateur `@abstractmethod` et `@classmethod`.

abstractstaticmethod : obsolète depuis la version 3.3 de Python ; il est maintenant possible d'utiliser le double décorateur `@abstractmethod` et `@staticmethod`.

abstractproperty : obsolète depuis la version 3.3 de Python ; il est maintenant possible d'utiliser le double décorateur `@abstractmethod` et `@property`.

Le listing 8.27, extrait de la documentation du module *abc*, illustre les différents décorateurs dans leur forme conseillée.

Listing 8.27 – Extrait de la documentation de *abc* : décorateurs

```
1  class C(metaclass=ABCMeta):
2      @abstractmethod
3      def my_abstract_method(self, ...):
4          ...
5      @classmethod
6      @abstractmethod
7      def my_abstract_classmethod(cls, ...):
8          ...
9      @staticmethod
10     @abstractmethod
11     def my_abstract_staticmethod(...):
12         ...
13
14     @property
15     @abstractmethod
16     def my_abstract_property(self):
17         ...
18     @my_abstract_property.setter
19     @abstractmethod
20     def my_abstract_property(self, val):
21         ...
```

Le listing 8.28 est un exemple de création de méthode abstraite. Le résultat de l'instanciation des classes ainsi définies est représenté dans le listing 8.29. Il n'est pas possible d'instancier `ClasseAbstraite` car `une_methode` est abstraite. `SousClasse1` hérite de `ClasseAbstraite` et ne redéfinit pas la méthode, donc il n'est pas possible de l'instancier. Par contre, `SousClasse2` redéfinit `une_methode` ; il est donc possible de l'instancier.

Listing 8.28 – Méthode abstraite

```
from abc import ABCMeta, abstractmethod

class ClasseAbstraite(metaclass=ABCMeta):
    @abstractmethod
    def une_methode(self):
        """Méthode abstraite. Il n'est pas possible d'instancier
        la classe.
        """
        pass

class SousClasse1(ClasseAbstraite):
    """La méthode abstraite n'est pas redéfinie.
    L'instanciation est impossible.
    """
    pass

class SousClasse2(ClasseAbstraite):
    def une_methode(self):
        """La méthode est redéfinie. Il est maintenant possible
        d'instancier la classe.
        """
        pass
```

Listing 8.29 – Instanciation de classe abstraite

```
>>> ClasseAbstraite()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class ClasseAbstraite with abstract
   methods une_methode
>>> SousClasse1()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class SousClasse1 with abstract
  methods une_methode
>>> SousClasse2()
<__main__.SousClasse2 object at 0x7f6346ce5be0>
```

8.6 Expressions rationnelles : re

Le module `re` implémente en Python les *expressions rationnelles*[17], qui analysent une chaîne de caractère pour tester si celle-ci contient un certain motif.

`compile(str)` transforme une chaîne de caractères en un *objet expression rationnelle*, qui est spécialisé dans la recherche de ce motif particulier (et donc plus rapide à exécuter).

17. *Regular expressions* en anglais.

`search(str)` **ou** `search(motif, str)` : vrai si la chaîne contient le motif compilé ou le motif passé en paramètre. Renvoie un objet de type *MatchObject*.

`motif.match(str)` **ou** `search(motif, str)` : vrai si la chaîne correspond dans son intégralité au motif compilé ou au motif passé en paramètre. Renvoie un objet de type *MatchObject*.

MatchObject.`group(k)` : Renvoie la chaîne de caractères correspondant aux k^{e} parenthèses de l'expression rationnelle. Le groupe numéro 0 correspond à l'expression complète.

Utiliser une chaîne de caractères contenant juste un mot, sans caractère spécial, revient à faire un détecteur de ce mot. Le programme suivant, appelé par `python3 finder.py file.tex`, aura exactement le même effet que `grep Python file.tex`.

Listing 8.30 – Trouver les lignes contenant Python

```
import sys, re

testPython = re.compile("Python")
fichier_entree = open(sys.argv[1])
for ligne_courante in fichier_entree:
    mo = testPython.search(ligne_courante)
    if mo:
        print(ligne_courante, end="")
```

Toutefois, cet objectif aurait pu être atteint par un simple `"Python" in ligne_courante` ou `ligne_courante.find("Python")`. En général, il est préférable d'utiliser les méthodes et opérateurs du type *str* lorsque c'est possible, et de réserver les expressions rationnelles aux situations un peu plus complexes.

8.6.1 Caractères spéciaux

Les expressions rationnelles permettent d'utiliser des jokers, des quantificateurs, de définir des classes de caractères et de former des groupes. Cela permet de réaliser l'analyse *lexicale* d'un texte, c'est-à-dire d'en identifier les mots [18] ou plus généralement les briques élémentaires. Dans ce sens, les séparateurs entre les mots ne sont pas que les blancs, ni tous les blancs.

(...) : définit un groupe. Il peut être accompagné de contraintes de quantification.

? : ce qui précède peut être présent 0 ou 1 fois.

+ : ce qui précède peut être présent au moins une fois.

***** : ce qui précède peut être présent un nombre quelconque de fois.

{m, M} : ce qui précède doit être présent entre `m` et `M` fois inclus. Les deux entiers peuvent être égaux, et se confondre en `{m}`. Si `m` est omis (par `{,M}`) cela se traduit par au plus `M` fois, et si `M` est omis (par `{m,}`) cela se traduit par au moins `m` fois.

m1|m2 : le motif est vérifié si le sous-motif `m1` est vérifié ou si le sous-motif `m2` est vérifié.

[...] : représente une classe de caractères. Ainsi `[aeiouy]` est n'importe quelle voyelle minuscule. Il est possible de définir des segments de lettres par `-`. Les classes `[A-Za-z]` ou `[0-9]` sont explicites, mais `[%-,]` l'est nettement moins (l'ordre des caractères usuels est donné en annexe E). Il convient d'échapper les caractères '`[`', '`]`' et '`-`' dans une classe de caractères, mais les autres caractères spéciaux peuvent être écrits normalement ; ainsi, la classe des quatre opérateurs de base est `[+\-*/]`. Le caractère `^` crée une négation, et doit être utilisé en début de classe : `[^0-9]` représente « tout sauf un chiffre ».

Des classes de caractères sont prédéfinies :

18. Les étages suivants sont l'analyse syntaxique, pour organiser les mots selon une grammaire (ce qui le plus souvent fabrique un arbre) puis l'analyse sémantique, pour en extraire le sens (ce problème-là est largement ouvert).

`\d` : chiffre décimal. Son complémentaire est `\D`.

`\w` : caractère alphanumérique ; correspond à `[a-zA-Z0-9_]`. Son complémentaire est `\W`.

`\s` : caractère blanc, qui peut être une espace, une tabulation ou un saut de ligne. Ce sont aussi les séparateurs de mots standards pour la méthode *split* de *str*.

`^` : désigne un début de ligne.

`$` : désigne une fin de ligne.

Certains caractères peuvent être spéciaux pour le type *str* de Python. Les caractères ayant un sens particulier pour *re* sont `. ^ $ * + ? { } [ ] \ | ( )`. Pour faire référence à un caractère spécial de l'expression rationnelle, il convient aussi de l'échapper par '\'. Ainsi '`.`' représente n'importe quel caractère, mais '`\.`' représente un point, et doit être écrit `"\\."` dans la chaîne de caractères Python. Le caractère '\' est également spécial pour le type *str* de Python. Il convient alors de l'échapper par '\\' (en Python), et donc d'écrire `"\\\\"` pour représenter un caractère '\'.

Voici quelques exemples :

`"bb|[^b]{2}"` : *two b or not two b.*

`"\\\\((sub){0,2}section|chapter)"` : trouve les marqueurs de structure de documents LaTeX, qui sont les niveaux `\chapter`, `\section`, `\subsection` et `\subsubsection`.

`"[0-9]+"` ou `"\d+"` : nombres décimaux.

`"0[oO][0-7]+"` : nombres octaux.

`"0[xX][0-9a-fA-F]+"` : nombres hexadécimaux.

`"(\d{1,3}\\.){3}\d{1-3}"` : adresse IPv4 (comme 172.20.27.47). Ne vérifie pas que chaque nombre appartient à l'intervalle $[0, 255]$.

`"^\s*[a-z]"` : utilisé avec *search*, trouve les lignes commençant par une minuscule, éventuellement précédée de blancs.

`"fin\\.$"` : utilisé avec *search*, trouve les lignes finissant par *"fin."*.

8.6.2 Quantificateurs gloutons

Les programmes 8.31 et 8.32 semblent équivalents : contenir un 4, puis n'importe quoi, puis un 2 sur une même ligne est exactement la même chose que demander que la ligne soit exactement composée de n'importe quoi, puis un 4, puis n'importe quoi, puis un 2, et se termine par n'importe quoi.

Listing 8.31 – Trouver ce qui est entre 4 et 2 sur chaque ligne de l'entrée standard

```
1 import sys, re
2
3 test42 = re.compile("4(.*)2")
4 while True:
5     s = sys.stdin.readline()
6     mo = test42.search(s)
7     if mo:
8         print("Between 4 and 2 lies ", mo.group(1))
```

Listing 8.32 – Trouver ce qui est entre 4 et 2 sur chaque ligne de l'entrée standard

```
1 import sys, re
2
3 test42 = re.compile(".*4(.*)2.*")
4 while True:
5     s = sys.stdin.readline()
6     mo = test42.match(s)
7     if mo:
8         print("Between 4 and 2 lies ", mo.group(1))
```

En fait, le moteur d'expression rationnelle va essayer de rendre `".*"` aussi grand que possible tout en vérifiant ses contraintes. Dans la version avec *search*, il n'y a qu'un seul groupe de ce type, et face à la chaîne `"4242"` elle utilisera le premier `4` et le second `2` pour valider les contraintes, laissant `24` pour le groupe numéro 1. La version avec *match* utilise trois groupes de ce type, et le premier, qui n'est pas paranthésé, récupère donc le premier `42`. Le second `4` et le second `2` valident les contraintes ; il ne reste rien, donc `""` pour le groupe numéro 1. Ainsi les mêmes lignes sont détectées par les deux programmes, mais ce ne sont pas les mêmes groupes qui sont créés.

8.7 Interactions avec le système d'exploitation

L'une des interactions usuelles d'une programme avec le système d'exploitation est le lancement d'autres programmes, et l'utilisation de leur production.

8.7.1 os

Le module `os` donne accès au système d'exploitation.

`system(cmd)` : exécute la commande unix/linux définie par la chaîne de caractères `cmd` et renvoie son code de sortie.

`popen(cmd)` : exécute la commande unix/linux `cmd` et renvoie sa sortie standard. On peut alors parcourir celle-ci par une boucle `for`. Les fonctions `popen2` et `popen3` permette la redirection de l'entrée standard et de la sortie d'erreur.

`environ, getenv, putenv` : permettent de consulter et de modifier les variables d'environnement.

Listing 8.33 – Invoquer *ls* et lire le résultat

```
1 import os
2
3 sortie = os.popen("ls -rtla")
4 for ligne in sortie:
5     print(ligne.rstrip())
```

Le module `os` permet également de gérer les fichiers et les répertoires, en particulier grâce aux fonctions suivantes [19] :

`walk(top)` : parcours tous les fichiers et répertoires en dessous du répertoire *top*. Retourne un ensemble de triplets.

`path.dirname(name), path.basename(name), path.split(name)` : permet d'extraire le nom de fichier et le nom de répertoire d'un chemin.

`path.splitext(name)` : permet d'extraire l'extension d'un nom de fichier.

`path.join(dir, file)` : permet de fusionner un nom de répertoire et un nom de fichier.

Le listing 8.34 illustre quelques fonctions de manipulation de noms de fichiers et de répertoires. Il est possible de remplacer la plupart de ces fonctions par des opérations sur les chaînes de caractères, mais notons qu'ici les spécificités du système d'exploitation sont prises en compte, et que les cas particuliers sont gérés.

Listing 8.34 – Utilisation de `os.path`

```
1 >>> import os
2 >>> name = '/usr/share/gmt/conf/gmt.conf'
3 # Test d'existance d'un fichier ou répertoire
```

19. Ne réinventez pas la roue, utilisez les fonctions existantes !

```
>>> os.path.exists(name)
True
# Extraction du répertoire
>>> os.path.dirname(name)
'/usr/share/gmt/conf'
# Extraction du nom de fichier
>>> os.path.basename(name)
'gmt.conf'
# Extraction du répertoire et du nom de fichier
>>> os.path.split(name)
('/usr/share/gmt/conf', 'gmt.conf')
# Extraction de l'extension d'un nom de fichier
>>> f = 'gmt.conf.old'
>>> os.path.splitext(f)
('gmt.conf', '.old')
# fusion d'un nom de répertoire et d'un nom de fichier
>>> dir = '/ens2/inf/unix/exos'
>>> file = 'exo01.sh'
>>> os.path.join(dir, file)
'/ens2/inf/unix/exos/exo01.sh'
```

Le listing 8.35 illustre l'utilisation de `os.walk` pour parcourir une arborescence.

Listing 8.35 – Parcours d'une arborescence avec `os.walk`

```
import os

BASEDIR = '/etc/network'
for root, dirs, files in os.walk(BASEDIR):
    # Pour chaque répertoire dans BASEDIR, crée une liste
    # de répertoires et de fichiers
    print('Base : ', root)
    # Parcours de la liste de répertoires
    print('Répertoires :')
    for d in dirs:
        dirname = os.path.join(root, d)
        print(dirname)
    # Parcours de la liste de fichiers
    print('Fichiers :')
    for f in files:
        filename = os.path.join(root, f)
        print(filename)
```

Le listing 8.35 combine l'utilisation de `os.walk` et des expressions rationnelles [20] pour sélectionner les fichiers à traiter. Notons que dans cet exemple, la liste des répertoires est ignorée.

Listing 8.36 – Parcours d'une arborescence avec `os.walk` et une expression rationnelle

```
import os
import re

# Répertoire de base
BASEDIR = '/etc/gmt'
# Expression rationnelle correspondant au nom de fichier
PATTERN = r'^gmt.*\.conf'
expr = re.compile(PATTERN)
for root, _, files in os.walk(BASEDIR):
```

20. Voir section 8.6

```
10      # Parcours de la liste de fichiers
11      for f in files:
12          # Tester si le fichier correspond au motif
13          test = expr.match(f)
14          # Si le fichier correspond au motif, afficher son nom
15          if test:
16              filename = os.path.join(root, f)
17              print(filename)
```

8.7.2 subprocess

Il est désormais conseillé d'utiliser le module `subprocess` plutôt que `os`, plus proche de la structure du système d'exploitation réel, mais à la syntaxe plus lourde [21].

`call(cmd, stdin, stdout, stderr, shell, timeout)` : remplace `os.system(cmd)`, en permettant une meilleure gestion des entrées-sorties du processus et du contexte d'exécution.

`Popen(args, stdout=PIPE, ...)` : remplace `os.popen(cmd)`, et quelques autres. Renvoit un objet de type *Popen*, qui contient notamment les tuyaux vers els entrées-sorties standard de la commande. Le paramètre `args` est une liste des mots à placer sur la ligne de commande à exécuter : `args[0]` est le nom de la commande en l'absence de shell, et le nom du shell à utiliser sinon (par exemple, `args[0]="/bin/bash"`). Le paramètre `universal_newlines=True` place le système en mode texte ; il faut decoder la sortie sinon, avec la méthode `decode()` de *str*.

`call(args, ...)` : disponible à partir de Python 3.5. Remplace à la fois `call` et la plupart des utilisations directes de `Popen`.

Listing 8.37 – Invoquer *ls* et lire le résultat

```
1 import subprocess
2 from subprocess import PIPE, Popen
3
4 proc = Popen(["ls", "-rtla"], stdout=PIPE, universal_newlines=True)
5 sortie = proc.stdout
6 for ligne in sortie:
7     print(ligne.rstrip())
```

8.8 Gestion du temps : time

Le module `time` fournit quelques outils pour interagir avec le temps du système. La représentation du temps à destination des humains se fait mieux par `datetime` (section 8.3).

`clock()` : temps processeur consommé par le processus depuis sa création, en secondes.

`time()` : temps du système, en secondes. En général, il s'agit du nombre de secondes depuis le 1er janvier 1970. Permet de faire un chronomètre : la précision est souvent de l'ordre de la microseconde.

`sleep(sec)` : met le processus en pause pendant le temps indiqué, en secondes. Le paramètre peut être un flottant. Ce temps peut être raccourci si le processus reçoit un signal (via la commande unix *kill*, par exemple) ou allongé si le processeur est surchargé par d'autres tâches : `sleep` ne lui assure pas une priorité supérieure au réveil.

21. ...et que l'on ne comprend qu'une fois que l'on sait comment dont gérées les entrées-sorties et les redirections par le système.

Listing 8.38 – Utilisation du module time

```
>>> import time
>>> t1 = time.time()
>>> t2 = time.time()
>>> print(t1, t2, t2-t1)
1441717912.883906 1441717914.923495 2.0395891666412354
```

8.9 Interroger le web : urllib et html.parser

Le module `urllib` permet de récupérer des ressources sur internet (le web, mais aussi selon d'autres protocoles comme FTP). Lorsque ces ressources sont des pages web, au format HTML, le module `html.parser` fournit une boîte à outils pour en travailler la structure, définie par un ensemble de balises. Il peut s'appliquer à tout document de cette forme, donc XML.

8.9.1 urllib

Le module `urllib` permet d'accéder à des pages web. Il est découpé en sous-modules :

`request` : le plus important, qui réalise l'interrogation.

`response` : pour gérer la page retournée comme un fichier (cf. section 7.3).

`parse` : pour gérer l'encodage des caractères spéciaux dans l'url (l'espace s'écrit `%20`) et les paramètres passés à la page.

`error` : parce que les serveurs web et les réseaux ont quantité de façon de refuser de faire ce qu'on attend d'eux. Permet de gérer cela comme une exception (section 7.2).

`robotparser` : dans la mesure où le programme Python est un robot, teste si la page demandée est autorisée selon le fichier *robots.txt* du serveur.

La fonction `urlopen(url)` de `request` prend comme paramètre la chaîne de caractère contenant l'url demandée (y compris `"http://"`) et renvoie la page web correspondante. Il convient de la lire comme un ensemble de lignes. En pratique les connections HTTP ou HTTPS renvoient un *http.client.HTTPResponse*, qui est itérable. Elle dispose de plusieurs paramètres optionnels, comme *timeout*, dont la valeur par défaut est en général un peu trop longue, ou *context*, *cafile* et *capath*, qui permettent de s'assurer de la sécurité d'une connection HTTPS.

Le module `urllib` interroge les variables d'environnement du système pour découvrir un éventuel proxy.

Le code 8.39 s'exécute en ligne de commande :

```
$ python3 acquerir_url.py poestories.com/read/houseofusher
```

Listing 8.39 – Lire une page web fournie en paramètre

```
import urllib.request
import sys

if __name__ == '__main__':
    if len(sys.argv)>1:
        # Si pas de protocole spécifié, utiliser http
        if (sys.argv[1][:7] == 'http://') or \
           (sys.argv[1][:8] == 'https://'):
            url = sys.argv[1]
        else:
            url = 'http://'+sys.argv[1]
        # ouvrir la page
        page = urllib.request.urlopen(url, timeout=3)
```

```
14          for ligne in page:
15              # écrire chaque ligne
16              print(ligne.decode("utf-8"), end="")
```

8.9.2 html.parser

La page web est livrée par `urllib.request.urlopen` avec ses balises HTML. Pour exploiter la structure du code HTML, on peut utiliser la classe `HTMLParser` définie dans le module `html.parser`. Si on préfère analyser la page web comme un texte quelconque, on peut se limiter aux fonctionnalités de la classe `str` ou de la bibliothèque `re`.

Le module *html.parser* définit une classe *HTMLParser* dont le rôle est de traiter les informations contenues dans le fichier HTML. Les méthodes adéquates du parser sont appelées lorsque des balises, des données, ou d'autres éléments HTML sont rencontrés. Par défaut, *HTMLParser* ne fait rien dans ces cas : il convient de définir une classe-fille de *HTMLParser* qui s'en charge.

Le code 8.40 s'exécute en ligne de commande :

```
$ python3 extr_html.py http://www.sesamath.net/index.php?page=professiondefoi
```

Listing 8.40 – Extraire la structure d'une page web

```
1  import sys
2  import urllib.request
3  from html.parser import HTMLParser
4
5  class MyHTMLParser(HTMLParser):
6      def __init__(self):
7          super().__init__()
8          # Compte le niveau d'imbrication des balises
9          self.increment = 0
10     def handle_starttag(self, tag, attrs):
11         print("  "*self.increment, tag, "[")
12         self.increment += 1         # ouverture de balise
13     def handle_endtag(self, tag):
14         self.increment -= 1         # fermeture de balise
15                                     # (on ne vérifie pas les vis-à-vis)
16         print("  "*self.increment, "]")
17     def handle_data(self, data):
18         """ afficher le contenu avec la bonne indentation """
19         # supprime espaces et retour à la ligne éventuels
20         s = data.strip()
21         if s:                       # ne pas afficher de ligne vide
22             print("  "*self.increment, s)
23
24 if __name__ == "__main__":
25     # Création du parser
26     parser = MyHTMLParser()
27
28     url = sys.argv[1]      # On ne vérifie pas la validité des paramètres
       ...
29     page = urllib.request.urlopen(url)
30     for ligne in page:
31         # Alimentation du parser
32         parser.feed(ligne.decode("utf-8"))
```

8.10 Bases de données

Un système de gestion de bases de données (SGBD) est un programme chargé de stocker de l'information structurée sur un serveur, ainsi que de l'interroger et la mettre à jour.

Les SGBD libres les plus courants sont MySQL [22], MariaDB [23], PostgreSQL [24] ou SQLite ; les SGBD commerciaux les plus courants sont Access et Oracle.

Tous sont accessibles depuis la bibliothèque `pyodbc` (*Python Open DataBase Connectivity*) et/ou `pypyodbc`. Ces bibliothèques sont génériques, et la configuration de l'accès à un SGBD en particulier peut être un peu absconse. Des bibliothèques spécialisées, comme `mysql.connector` pour MySQL, permettent des accès plus simples. Elles ont toutes la même logique de *connexion* et de *curseur*.

Les données d'un SGBD relationnel sont structurées sous forme de tables, dont les colonnes sont fortement typées. Chaque ligne correspond à un enregistrement dans la table (une personne, un cours, une facture, ...). Le serveur est interrogé en [25] SQL (Structured Query Langage).

8.10.1 Connexion

La première opération est de se connecter au serveur par la fonction `connect()`, en renseignant le *driver* à utiliser, c'est-à-dire le type de SGBD, ainsi que la machine sur laquelle il se trouve, et l'identité du client. On récupère ensuite un curseur sur cette connexion qui permettra d'envoyer des requêtes SQL au serveur.

Une connexion à un serveur MySQL installé sur la machine locale [26] pourrait être :

```
import mysql.connector
cnxn = mysql.connector.connect(user='username', password='strongenough',
  host='127.0.0.1', database='test')
cursor = cnxn.cursor()
```

8.10.2 Bases de données sans serveur : SQLite

La bibliothèque `sqlite3`, incluse dans Python par défaut, permet d'utiliser le disque dur comme une base de données. Tant que la base de donnée reste d'une taille modérée – *i.e.* que sa taille totale soit inférieure à la taille maximale d'un fichier sur le disque – il s'agit d'un moyen efficace de stocker et d'interroger l'information.

Le typage de SQLite est dynamique, et sa méthode de stockage de l'information ne s'appuie pas sur le fait que les champs des tables aient une occupation mémoire fixe, contrairement à la plupart des SGBD relationnels. Il y a donc nettement moins de types de données : `TEXT`, `INTEGER`, `REAL`, `BLOB` et `NONE`.

Listing 8.41 – Création et interrogation d'une BdD SQLite

```
import sqlite3

# Utilise ou crée la base dans un fichier nommé ecole.db
conn = sqlite3.connect('ecole.db')

# c est un curseur sur la base ecole
c = conn.cursor()
```

22. Suite au rachat de MySQL AB par Sun Microsystems, puis au rachat de Sun Microsystems par Oracle Corporation, le SGBD MySQL est disponible sous une double licence GPL et propriétaire.
23. Il s'agit d'un clone de MySQL entièrement libre, édité sous licence GPL.
24. PostgreSQL est disponible sous une licence libre de type BSD.
25. Chaque SGBD « parle » son propre idiome de SQL, se référer à la documentation du SGBD cible pour écrire les requêtes.
26. localhost : 127.0.0.1

```
8
9  # Création d'une table etudiants
10 c.execute('''CREATE TABLE etudiants
11     (id INTEGER PRIMARY KEY, nom TEXT, prenom TEXT, promo INTEGER)''')
12 conn.commit()
13
14 # Ajout de deux étudiants
15 inscrits = [(12, 'Karpov', 'Antoine', 2017),
16             (121, 'Carlsen', 'Grand', 2020)]
17 c.executemany('INSERT INTO etudiants VALUES (?,?,?,?)', inscrits)
18 conn.commit()
19
20 # Recherche des étudiants nommés K...
21 c.execute('SELECT prenom, nom FROM etudiants WHERE nom LIKE "K%"')
22 for row in c.fetchall():
23     print("%s %s"%row)
24
25 conn.close()
```

8.10.3 Requêtes SQL

Une interrogation SQL commence par `SELECT` et renvoie une ou plusieurs lignes composées d'une ou plusieurs colonnes[27].

```
SELECT colonne(s) FROM table(s)
  [WHERE condition(s)]
  [[GROUP BY colonne(s)] [HAVING condition(s)]]
  [ORDER BY colonne(s)];
```

La requête est transmise par la méthode `execute()` du curseur. L'information renvoyée par le SGBD est disponible au travers de ce même curseur. Il est possible de l'utiliser comme un itérateur :

```
1 query = ("SELECT * FROM etudiants WHERE promo > 2016 ORDER BY promo DESC,
      nom")
2 cursor.execute(query)
3 for (id, nom, prenom, promo) in cursor:
4     print(id, nom, prenom, promo)
```

Il est possible de récupérer l'information ligne par ligne par `fetchone()`. La ligne est de type `Row` ; ses colonnes peuvent être extraites par leur numéro de colonne, comme un `tuple`, mais aussi par le nom de la colonne dans la requête.

```
1 while True:
2     row = cursor.fetchone()
3     if not row:
4         break
5     print('Nom :', row.nom, '\tNote :', row[1])
```

La méthode `fetchall` permet de récupérer toute l'information renvoyée dans une unique structure, que l'on peut parcourir par `for`.

```
1 query = ("SELECT * FROM etudiants WHERE promo > 2016 ORDER BY promo DESC,
      nom")
2 cursor.execute(query)
```

27. Se référer au cours de l'UV 2.5 pour la composition de requêtes.

```
rows = cursor.fetchall()
for row in rows:
    print('Nom :', row.nom, '\tNote :', row[1])
```

Une requête peut aussi être une modification de la table. La méthode `commit()` rend les modifications définitives sur le serveur.

```
sqlreq = "INSERT INTO etudiants VALUES (64, 'Kasparov', 'Jerry', 2018);"
cursor.execute(sqlreq)
cnxn.commit()
```

8.11 Analyse de données : pandas

`pandas` est une bibliothèque Python libre dédiée à la manipulation et à l'analyse de données. Elle est spécialisée dans la manipulation de grands volumes de données et permet de lire facilement les tableaux de données au format CSV, Excel, HTML, HDF5, ...

La bibliothèque `pandas` est basée sur `numpy` mais elle possède également des structures de haut niveau, ce qui en fait un outil à la fois efficace et simple d'usage.

Elle est bien plus efficace que `numpy` pour lire des données sous forme de texte. Le listing 8.42 représente un exemple de lecture d'un fichier texte de plus de deux millions de lignes avec `numpy` et `pandas`. Dans ce cas, la lecture est approximativement 25 fois plus rapide avec `pandas.read_csv` qu'avec `numpy.loadtxt`.

Listing 8.42 – Performances de Pandas / numpy

```
>>> import numpy as np
>>> import pandas
>>> import time
>>> a=time.clock() ; X=np.loadtxt('coquilles.txt') ; b=time.clock()
>>> c=time.clock() ; Y=pandas.read_csv('coquilles.txt', sep=' ', header=
  None) ; d=time.clock()
>>> print(b-a, d-c)
31.432743999999996 1.259732999999997
```

La documentation de `pandas` est accessible sur le site :
http://pandas.pydata.org/pandas-docs/stable/index.html.

8.11.1 Structures de données pandas

Type Series

Le type `Series` permet de gérer les données monodimensionnelles ; il contient un ensemble de valeurs et un index associé.

Le listing 8.43 est un exemple d'utilisation de `pandas`. La variable de type `Series` s'appuie sur un tableau `numpy`, et elle se comporte de manière similaire. Notons toutefois qu'elle possède des méthodes supplémentaires, comme par exemple `describe`.

Notons enfin que le type `Series` permet de gérer les valeurs non défines (NaN).

Listing 8.43 – Type Series en pandas

```
import numpy as np
import pandas

a = np.random.randn(1000)
b = pandas.Series(a)

```

```
print('Moyenne a et b :', np.mean(a), np.mean(b))
print('Description de b\n', b.describe())

print("\nAjout d'un NaN à a et b")

a[42] = float('nan')
print('Moyenne a et b :', np.mean(a), np.mean(b))
print('Description de b\n', b.describe())
```

Le listing 8.44 contient le résultat de l'exécution du script 8.43. On peut remarquer la différence de comportement entre les types `pandas.Series` et `numpy.array` lorsque les données contiennent des *NaN*.

Listing 8.44 – Utilisation de pandas

```
Moyenne a et b : -0.0181407001393 -0.0181407001393
Description de b
count    1000.000000
mean       -0.018141
std         0.993980
min        -2.969224
25%        -0.684338
50%        -0.016392
75%         0.651308
max         2.763677
dtype: float64

Ajout d'un NaN à a et b
Moyenne a et b : nan -0.019186855653
Description de b
count    999.000000
mean      -0.019187
std        0.993927
min       -2.969224
25%       -0.684589
50%       -0.016452
75%        0.650177
max        2.763677
dtype: float64
```

Par défaut, `pandas` indexe les données avec des entiers commençant par 0, mais il est possible de modifier ce comportement. Le listing 8.45 illustre cette possibilité.

Listing 8.45 – Gestion des index

```
>>> test = pandas.Series([4, 7, -5, 3], index=['a', 'b', 'c', 'd'])
>>> test
a    4
b    7
c   -5
d    3
dtype: int64
>>> test['a':'c']
a    4
b    7
c   -5
dtype: int64
>>> 'd' in test
```

```
14 True
```

Type DataFrame

Un objet de type `DataFrame` est une collection de colonnes, chacune étant de type `Series`. Chaque colonne peut contenir des données d'un type particulier. Les fonctions `pandas` permettant de lire un fichier retournent une variable de type `DataFrame`.

Considérons par exemple le fichier 8.46 qui contient une première ligne d'en-tête et des données sous 3 colonnes séparées par des espaces ou des tabulations.

Listing 8.46 – Fichier de données

```
1 date     heure    hauteur
2 28/02/2013 23:50:00 0.961
3 01/03/2013 00:00:00 0.928
4 01/03/2013 00:10:00 0.934
5 01/03/2013 00:20:00 0.958
6 01/03/2013 00:40:00 1.190
```

Le listing 8.47 illustre une première version de lecture de fichier texte grâce à `pandas`. Lors de la lecture, il faut préciser que le délimiteur est un caractère blanc. `pandas` a automatiquement reconnu le type de données : chaînes de caractères pour *date* et *heure*, et réel pour *hauteur*.

Listing 8.47 – Lecture de fichier avec pandas, version 1

```
1 >>> import pandas
2 >>> data = pandas.read_csv('brest_maregraphe.txt', delim_whitespace=True)
3 >>> print(type(data))
4 <class 'pandas.core.frame.DataFrame'>
5 >>> print(data.columns)
6 Index(['date', 'heure', 'hauteur'], dtype='object')
7 >>> print(type(data['date']))
8 <class 'pandas.core.series.Series'>
9 >>> print(type(data['date'][0]), type(data['heure'][0]), type(data['
    hauteur'][0]))
10 <class 'str'> <class 'str'> <class 'numpy.float64'>
```

Le listing 8.48 est une évolution de 8.47 dans lequel la date est décodée. La variable *data* contient alors une colonne nommée *datetime* de type `TimeStamp`. Dans cet exemple, `pandas` a reconnu automatiquement le format des dates [28]. Notons cependant que laisser `pandas` déterminer automatiquement le format des dates est pénalisant en terme de temps, et que le décodage peut être ambigu [29]. Il est donc conseillé de préciser le format de date attendu. Ce format est identique à celui de *datetime* (voir section 8.3.2, p. 178).

Listing 8.48 – Lecture de fichier avec pandas, version 2

```
1 >>> import pandas
2 >>> data = pandas.read_csv('brest_maregraphe.txt', delim_whitespace=True,
     parse_dates={'datetime': ['date', 'heure']}, infer_datetime_format=
    True)
3 >>> data
4               datetime  hauteur
5 0 2013-02-28 23:50:00    0.961
6 1 2013-03-01 00:00:00    0.928
```

28. Le paramètre `infer_datetime_format=True` précise que `pandas` doit essayer de déterminer automatiquement le format de la date.

29. Ainsi, la deuxième date du fichier pourrait être le 1er mars 2013 ou le 3 janvier 2013.

```
2 2013-03-01 00:10:00     0.934
3 2013-03-01 00:20:00     0.958
4 2013-03-01 00:40:00     1.190
>>> data.columns
Index(['datetime', 'hauteur'], dtype='object')
>>> type(data['datetime'][0])
<class 'pandas.tslib.Timestamp'>
```

La fonction `read_csv` possède de nombreuses options. La table 8.5 en décrit quelques unes ; consulter la documentation pour l'ensemble des options.

TABLE 8.5: Quelques options de *read_csv*

Option	Signification
sep	Chaîne de caractères servant de séparateur de colonnes.
delim_whitespace	Utiliser les blancs (par exemple espace ou tabulation) comme séparateurs. *False* par défaut.
header	Numéro de ligne contenant le nom des colonnes. 0 par défaut. Utiliser *None* s'il n'y a pas d'en-tête.
parse_dates	Tentative d'analyser les données en dates–heures. Par défaut *False*. Peut valoir *True* ou une liste de colonnes.
date_parser	Fonction utilisée pour décoder la date.
infer_datetime_form	Tentative de décodage automatique du format des dates. *False* par défaut.
dayfirst	Lors de l'analyse des dates, décoder le jour avant le mois. *False* par défaut.
comment	Chaîne de caractères indiquant le début d'un commentaire.
nrows	Nombre de lignes à lire. Permet de tester un programme avec une partie du fichier uniquement. *None* par défaut (lire le fichier entièrement).

Gestion des dates

Le listing 8.49 décrit la manière de procéder pour spécifier le format des dates à décoder. Il illustre également la possibilité d'indexer les données selon la date. Ainsi, la variable *data* est indexée selon des indices, et la variable *data2* est indexée selon la date. Le listing 8.50 représente ces deux variables. Une fois les données indexées selon la date, il est possible de les ré-échantillonner temporellement. Plusieurs stratégies de ré-échantillonnage sont disponibles : laisser la donnée vide, copier la donnée d'avant ou d'après, interpoler linéairement les valeurs. Le listing 8.51 illustre le résultat de l'application de deux de ces stratégies.

Listing 8.49 – Lecture de fichier avec pandas, gestion de la date

```
def dateparse(ymd, hms):
    date = ymd + ' ' + hms
    return pandas.datetime.strptime(date, '%d/%m/%Y %H:%M:%S')

data = pandas.read_csv('brest_maregraphe.txt', parse_dates={'datetime':
  ['date', 'heure']}, date_parser=dateparse, comment='#',
  delim_whitespace=True)
# Indexation par la colonne datetime puis suppression de la colonne
  datetime
data2 = data.set_index(data['datetime']).drop('datetime', axis=1)
print('Resample 1')
```

```
print(data2.resample('10 min').asfreq())
print('Resample 2')
print(data2.resample('10 min').interpolate())
```

Listing 8.50 – Ré-indexation

```
Data :
               datetime  hauteur
0 2013-02-28 23:50:00     0.961
1 2013-03-01 00:00:00     0.928
2 2013-03-01 00:10:00     0.934
3 2013-03-01 00:20:00     0.958
4 2013-03-01 00:40:00     1.190
Data2 :
                      hauteur
datetime
2013-02-28 23:50:00     0.961
2013-03-01 00:00:00     0.928
2013-03-01 00:10:00     0.934
2013-03-01 00:20:00     0.958
2013-03-01 00:40:00     1.190
```

Listing 8.51 – Ré-échantillonnage de données

```
Resample 1
                      hauteur
datetime
2013-02-28 23:50:00     0.961
2013-03-01 00:00:00     0.928
2013-03-01 00:10:00     0.934
2013-03-01 00:20:00     0.958
2013-03-01 00:30:00       NaN
2013-03-01 00:40:00     1.190
Resample 2
                      hauteur
datetime
2013-02-28 23:50:00     0.961
2013-03-01 00:00:00     0.928
2013-03-01 00:10:00     0.934
2013-03-01 00:20:00     0.958
2013-03-01 00:30:00     1.074
2013-03-01 00:40:00     1.190
```

Le listing 8.52 illustre les possibilités de `pandas` en terme de fusion de lots de données. Il est possible de fusionner deux lots de données sur des colonnes communes (par défaut la fusion se fait sur les index) et de paramétrer le comportement de la fusion. Par défaut, l'index du premier lot de données est conservé, mais le paramètre *how* permet de modifier ce comportement.

Listing 8.52 – Fusion de données

```
>>> import pandas
>>> A = pandas.DataFrame(index=['A', 'C', 'D'], data={'val1': [1, 3, 4]})
>>> B = pandas.DataFrame(index=['B', 'C', 'D'], data={'val2': [4, 6, 8]})
>>> A
   val1
A     1
C     3
```

```
D      4
>>> B
   val2
B      4
C      6
D      8
>>> A.join(B)
   val1  val2
A      1   NaN
C      3   6.0
D      4   8.0
>>> A.join(B, how='right')
   val1  val2
B   NaN     4
C   3.0     6
D   4.0     8
>>> A.join(B, how='outer')
   val1  val2
A   1.0   NaN
B   NaN   4.0
C   3.0   6.0
D   4.0   8.0
>>> A.join(B, how='inner')
   val1  val2
C      3     6
D      4     8
```

La bibliothèque **pandas** permet également de manipuler les séquences de dates. Un extrait des nombreuses possibilités est représenté dans le listing 8.53.

Listing 8.53 – Intervalles de temps

```
# Séquence définie par début, nombre et fréquence
>>> pandas.date_range('1 jul 2017', periods=6, freq='7D')
DatetimeIndex(['2017-07-01', '2017-07-08', '2017-07-15', '2017-07-22',
               '2017-07-29', '2017-08-05'],
               dtype='datetime64[ns]', freq='7D')
# Séquence définie par début, fin et fréquence
>>> pandas.date_range('1 jul 2017', end=pandas.to_datetime('8-7-2017',
  dayfirst=True), freq='D')
DatetimeIndex(['2017-07-01', '2017-07-02', '2017-07-03', '2017-07-04',
               '2017-07-05', '2017-07-06', '2017-07-07', '2017-07-08'],
               dtype='datetime64[ns]', freq='D')
# Date définie en époque (secondes depuis le 1er janvier 1970)
>>> pandas.to_datetime([0, 1498815025], unit='s')
DatetimeIndex(['1970-01-01 00:00:00', '2017-06-30 09:30:25'], dtype='
  datetime64[ns]', freq=None)
# Dates définies en nombres de jours depuis une origine
>>> pandas.to_datetime([0, 4242], unit='D', origin='1/1/2000')
DatetimeIndex(['2000-01-01', '2011-08-13'], dtype='datetime64[ns]', freq=
  None)
```

J'ai craqué une allumette. Il a commencé à dessiner sur le mur gris et humide. Il a d'abord tracé un rectangle en hauteur pour encadrer le sujet. Après plusieurs traits adroits, le phare a commencé de naître. C'était surprenant, mais son habileté était intacte malgré sa folie. Les flammes se sont élevées, et ont éclairé le mur. Je me suis concentré sur la tour en faisant appel à mes souvenirs. J'ai reniflé quelque chose qui ressemblait à une brise salée. Le tableau devenait plus réel à mesure que je le fixais. J'ai fait un pas en avant. Mon pied ne s'est pas posé sur le feu.

Roger Zelazny, *les neuf princes d'Ambre*

9 Interface Homme-Machine

Sommaire

Ce chapitre définit les mécanismes usuels des interfaces homme-machine, et présente la façon de les mettre en œuvre avec la bibliothèque PyQt. Le lecteur est invité à prolonger cette lecture avec la documentation en ligne de PyQt et Qt pour ses usages spécifiques.

Cette bibliothèque n'est pas, et de loin, la seule façon de créer une fenêtre et des widgets graphiques. Pour des usages peu complexes, le module `tkinter`[1] peut être suffisant.

9.1 Rôle et structure d'une IHM

Une Interface Homme-Machine (IHM), comme son nom l'indique, permet à un utilisateur d'accéder à un système informatique. Elle crée des moyens (appelés *widgets*) pour que l'humain agisse sur l'état interne du programme, et lui permet de consulter cet état interne.

On appellera *Modèle* la partie du programme qui contient les données permettant de faire les traitements. On appellera *Vue* la partie du programme – et de l'interface – qui permet à l'utilisateur de connaître l'état du modèle. On appellera *Contrôleur* la partie du programme – et de l'interface – qui permet à l'utilisateur d'agir sur le modèle. La frontière entre vue et contrôleur est souvent ténue d'un point de vue informatique, un même widget, comme un slider ou une checkbox, pouvant relever de ces deux catégories.

L'outil graphique de création d'une IHM permet de positionner les éléments graphiques dans une fenêtre principale, d'identifier les actions possibles au sein de l'interface, et de créer les

1. Voir la documentation standard de Python : https://docs.python.org/3.5/library/tkinter.html

prototypes des méthodes qui permettront aux widgets contrôleurs d'agir sur le modèle.

La programmation d'IHM se décompose en trois phases :

1. Assemblage des éléments graphiques (widgets) constituant l'IHM.
2. Écriture des traitements à déclencher par les actions de l'utilisateur. Idéalement, ces traitements sont des méthodes du modèle, et peuvent être testés sans l'IHM.
3. Association des ces traitements aux actions de l'utilisateur.

Le lancement du programme muni de son IHM se fait par le code généré, qui contient la boucle principale (méthode *exec_()* de l'objet *QApplication*).

Il existe de nombreuses bibliothèques et outils pour créer des interfaces homme-machine. Certains sont spécifiques à un langage, comme Java/Swing, mais ce sont souvent des solutions transverses.

TABLE 9.1: toolkits graphiques

toolkit	**langage**	**cibles**	**outils**	**OS**	**license**
Qt	C++	Python (PyQt, PySide, PythonQt), Ruby, Ada, C#, Java, Perl, ...	Qt Designer, Qt Creator	Linux, Windows, OS X	GPL, LGPL, comm.
GTK+	C	Python (PyGTK), C++, Perl, Ruby, C#, Ada, ...	Glade	Linux, Windows, OS X	LGPL
wxWidgets	C++	Python (wxPython), C, C++, Ruby, Perl, Java, .NET Framework, Ada, ...	VisualWx, Boa Constructor, PythonCard, ...	Linux, Windows, OS X	sim. LGPL
Tk	C, Tcl	Python (Tkinter), Ruby, Perl, Ada, ...			BSD
Swing	Java	Java	Eclipse, NetBeans	cross plateform	

Ainsi, l'ensemble d'outils graphiques (*toolkit*) Qt est écrit en C++, compilé pour la plupart des systèmes d'exploitation, et de nombreux langages, dont Python par l'intermédiaire de PyQt, disposent de points d'accès à ces outils.

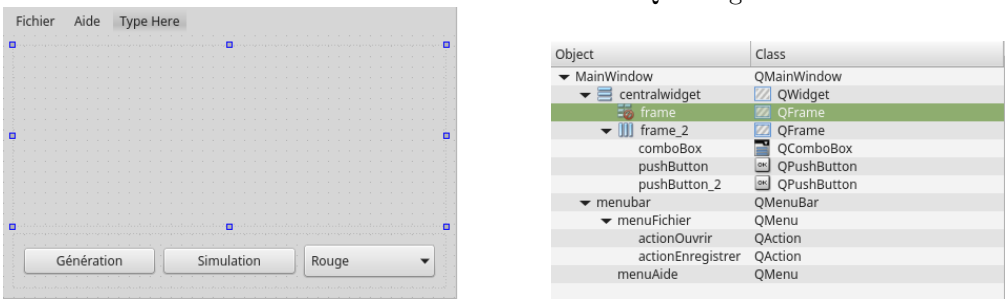
9.2 Bibliothèque Qt

La bibliothèque PyQt est organisée en modules. Ceux-ci sont chargés séparément, selon les besoins du programme. Le module `QtWidgets` définit les composants graphiques (*widgets*) permettant de construire une interface homme-machine usuelle. Le module `QtCore` définit les fonctionnalités de base, non graphiques, et notamment le mécanisme de signaux et de slots. Le module `QtGui` définit les fonctionnalités graphiques de base.

Il y a deux grandes catégories de widgets : les *containers* (conteneurs), qui peuvent contenir d'autres widgets, et les *components* (composants) qui ne le peuvent pas. Certains containers sont dit de "haut niveau", car ils ne peuvent pas être contenus dans un autre container ; c'est notamment le cas des fenêtres. Il est possible (et recommandé) d'associer un *layout* (agencement) à un conteneur pour gérer la façon dont les composants et autres conteneurs qu'il contient sont organisés [2].

2. La pratique du franglais dans le domaine des IHM est à la fois variée et massive. Si les termes "composant" et "component" sont à peu près aussi utilisés l'un que l'autre, "container" est plus courant que "conteneur", et "agencement" n'est presque jamais utilisé ; seul "layout" l'est. Le terme "widget" ne dispose pas de terme équivalent

FIGURE 9.1 – Fenêtre définie sous Qt Designer



conteneurs de haut niveau : QMainWindow, QDialog

conteneurs : QFrame, QGroupBox, QScrollArea, ...

composants :

Boutons : QPushButton, QRadioButton, QCheckBox, ...
Menus : QMenuBar, QPopMenu, ...
Visualisateurs : QListWidget, QtreeView, QTableView, QColumnView
Entrées : QTextEdit, QSlider, QScrollBar, QdateTimeEdit, ...
Autres : Spacer (horizontal and vertical), QLabel, QCalendarWidget, ...

agencements : QHBoxLayout, QVBoxLayout, QGridLayout, ...

Lors de la création d'un objet de la bibliothèque Qt, donc d'un descendant de *QObject*, il est possible de spécifier un paramètre *parent*. Ce paramètre n'a rien à voir avec une relation d'héritage en programmation objet. La destruction de l'objet parent entraîne la destruction récursive des objets qui lui sont attachés. Cela permet de supprimer efficacement et simplement tous les widgets qui constituent une fenêtre de dialogue ou d'alerte lors de sa fermeture.

L'ensemble de la bibliothèque Qt est documentée sur `http://doc.qt.io/qt-5`, et son implémentation Python sur `http://pyqt.sourceforge.net/Docs/PyQt5/`. Le livre [Sum12] constitue une approche progressive des concepts de la programmation d'IHM en Python avec PyQt.

9.3 Outil Qt Designer

L'outil Qt Designer permet de composer une fenêtre et d'y placer conteneurs et composants. Il s'agit d'un programme graphique, qui génère un code `.ui`. Ce code permet de générer du code source correspondant à cette interface graphique dans tout langage-cible de Qt, dont Python (*cf.* table 9.1). Ce fichier .ui est enregistré dans un format xml [3]. L'arborescence des objets graphiques est reprise dans l'outil *Object Inspector*. Les icônes à gauche des noms des objets rappellent le layout associé aux conteneurs.

Il est possible de tester le comportement de la fenêtre créée à partir de Qt Designer. Cela permet notamment de s'assurer que les objets graphiques réagissent comme attendu aux variations de la taille de la fenêtre.

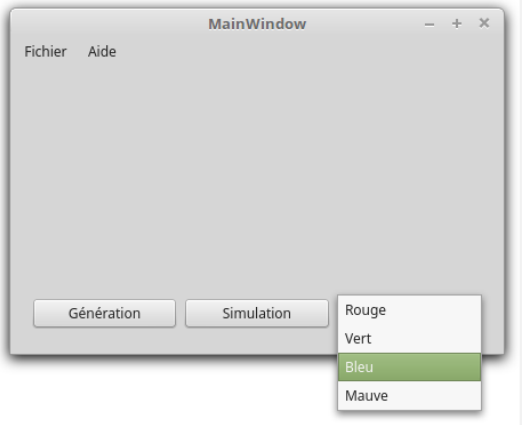
Il suffit de générer un fichier python à partir du fichier .ui à l'aide de `pyuic5`, en utilisant :

```
pyuic5 -x colors.ui > colors.py
```

en français ; il faut noter que ce n'est pas non plus un terme anglais.... Ce chapitre est assez fidèle à ces usages, même s'il est difficile de les cautionner.

3. Il s'agit d'un format texte modérément agréable à lire pour un humain, et trop verbeux pour être écrit sans une interface adaptée. Toutefois, il s'agit d'un format ouvert, et QtDesigner n'est pas nécessaire pour le modifier, juste utile.

FIGURE 9.2 – Fonctionnement de la fenêtre



L'option -x permet de rendre le fichier exécutable : il contient des actions à effectuer quand `__name__` est `"__main__"`. Sinon, il ne contient que la définition de la classe `Ui_MainWindow` ; il faut alors importer cette classe depuis un programme python, et créer une nouvelle fenêtre de la même manière que ce qui est fait dans les lignes 91 à 97. N'omettre ni l'appel à *show()* de l'objet *QMainWindow* (la fenêtre n'est pas visible, et ne reçoit donc pas les évènements de la souris via le gestionnaires de fenêtre du système d'exploitation) ni l'appel à *exec_()* de l'objet *QApplication* (il n'y a pas de boucle d'attente d'évènement, la fenêtre est détruite à l'instant de son affichage, et le programme se termine).

Listing 9.1 – Programme python généré par PyQt

```
1  # -*- coding: utf-8 -*-
2
3  # Form implementation generated from reading ui file 'ch9_colors.ui'
4  #
5  # Created by: PyQt5 UI code generator 5.5.1
6  #
7  # WARNING! All changes made in this file will be lost!
8
9  from PyQt5 import QtCore, QtGui, QtWidgets
10
11 class Ui_MainWindow(object):
12     def setupUi(self, MainWindow):
13         MainWindow.setObjectName("MainWindow")
14         MainWindow.resize(432, 278)
15         self.centralwidget = QtWidgets.QWidget(MainWindow)
16         self.centralwidget.setObjectName("centralwidget")
17         self.verticalLayout = QtWidgets.QVBoxLayout(self.centralwidget)
18         self.verticalLayout.setObjectName("verticalLayout")
19         self.frame = QtWidgets.QFrame(self.centralwidget)
20         sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Preferred,
             QtWidgets.QSizePolicy.Expanding)
21         sizePolicy.setHorizontalStretch(0)
22         sizePolicy.setVerticalStretch(0)
23         sizePolicy.setHeightForWidth(self.frame.sizePolicy().hasHeightForWidth())
24         self.frame.setSizePolicy(sizePolicy)
25         self.frame.setFrameShape(QtWidgets.QFrame.StyledPanel)
26         self.frame.setFrameShadow(QtWidgets.QFrame.Raised)
27         self.frame.setObjectName("frame")
28         self.verticalLayout.addWidget(self.frame)
29         self.frame_2 = QtWidgets.QFrame(self.centralwidget)
30         sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Preferred,
```

```
            QtWidgets.QSizePolicy.Preferred)
        sizePolicy.setHorizontalStretch(0)
        sizePolicy.setVerticalStretch(0)
        sizePolicy.setHeightForWidth(self.frame_2.sizePolicy().hasHeightForWidth()
            )
        self.frame_2.setSizePolicy(sizePolicy)
        self.frame_2.setFrameShape(QtWidgets.QFrame.StyledPanel)
        self.frame_2.setFrameShadow(QtWidgets.QFrame.Raised)
        self.frame_2.setObjectName("frame_2")
        self.horizontalLayout = QtWidgets.QHBoxLayout(self.frame_2)
        self.horizontalLayout.setObjectName("horizontalLayout")
        self.pushButton = QtWidgets.QPushButton(self.frame_2)
        self.pushButton.setObjectName("pushButton")
        self.horizontalLayout.addWidget(self.pushButton)
        self.pushButton_2 = QtWidgets.QPushButton(self.frame_2)
        self.pushButton_2.setObjectName("pushButton_2")
        self.horizontalLayout.addWidget(self.pushButton_2)
        self.comboBox = QtWidgets.QComboBox(self.frame_2)
        self.comboBox.setObjectName("comboBox")
        self.comboBox.addItem("")
        self.comboBox.addItem("")
        self.comboBox.addItem("")
        self.comboBox.addItem("")
        self.horizontalLayout.addWidget(self.comboBox)
        self.verticalLayout.addWidget(self.frame_2)
        MainWindow.setCentralWidget(self.centralwidget)
        self.menubar = QtWidgets.QMenuBar(MainWindow)
        self.menubar.setGeometry(QtCore.QRect(0, 0, 432, 27))
        self.menubar.setObjectName("menubar")
        self.menuFichier = QtWidgets.QMenu(self.menubar)
        self.menuFichier.setObjectName("menuFichier")
        self.menuAide = QtWidgets.QMenu(self.menubar)
        self.menuAide.setObjectName("menuAide")
        MainWindow.setMenuBar(self.menubar)
        self.actionOuvrir = QtWidgets.QAction(MainWindow)
        self.actionOuvrir.setObjectName("actionOuvrir")
        self.actionEnregistrer = QtWidgets.QAction(MainWindow)
        self.actionEnregistrer.setObjectName("actionEnregistrer")
        self.menuFichier.addAction(self.actionOuvrir)
        self.menuFichier.addAction(self.actionEnregistrer)
        self.menubar.addAction(self.menuFichier.menuAction())
        self.menubar.addAction(self.menuAide.menuAction())

        self.retranslateUi(MainWindow)
        QtCore.QMetaObject.connectSlotsByName(MainWindow)

    def retranslateUi(self, MainWindow):
        _translate = QtCore.QCoreApplication.translate
        MainWindow.setWindowTitle(_translate("MainWindow", "MainWindow"))
        self.pushButton.setText(_translate("MainWindow", "Génération"))
        self.pushButton_2.setText(_translate("MainWindow", "Simulation"))
        self.comboBox.setItemText(0, _translate("MainWindow", "Rouge"))
        self.comboBox.setItemText(1, _translate("MainWindow", "Vert"))
        self.comboBox.setItemText(2, _translate("MainWindow", "Bleu"))
        self.comboBox.setItemText(3, _translate("MainWindow", "Mauve"))
        self.menuFichier.setTitle(_translate("MainWindow", "Fichier"))
        self.menuAide.setTitle(_translate("MainWindow", "Aide"))
        self.actionOuvrir.setText(_translate("MainWindow", "Ouvrir"))
        self.actionEnregistrer.setText(_translate("MainWindow", "Enregistrer"))


if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    MainWindow = QtWidgets.QMainWindow()
```

```
94      ui = Ui_MainWindow()
95      ui.setupUi(MainWindow)
96      MainWindow.show()
97      sys.exit(app.exec_())
```

Les fenêtres de type *QMainWindow* sont riches mais complexes : elles sont adaptées à une interaction multiforme avec l'utilisateur, le réseau et/ou le disque. S'il s'agit de paramétrer une petite simulation et d'afficher ses résultats, une fenêtre de type *QDialog* sera souvent suffisante.

9.4 Layouts

Une variable de type layout (*i.e.* descendant de la classe *QLayout*) gère la façon dont sont agencés au sein d'un conteneur les éléments graphiques qui lui sont attribués. L'attribution se fait par la méthode `addWidget()` du layout ; le lien avec le conteneur se fait par la méthode `setLayout()` du conteneur, ou en passant le conteneur comme paramètre au constructeur du layout (*cf.* lignes 17 et 38 du listing 9.1).

La classe *QBoxLayout* positionne les éléments ajoutés en les juxtaposant selon une direction définie au départ (de gauche à droite, de droite à gauche, de haut en bas, ou de bas en haut). Ses descendants *QHBoxLayout* et *QVBoxLayout* permettent d'omettre le paramètre de direction, qui est *LeftToRight* pour le premier, et *TopToBottom* pour le second.

La classe *QGridLayout* définit un tableau dans lequel ranger des widgets. Sa taille n'est pas définie à la construction, mais évolue pour contenir toutes les coordonnées demandées en cours d'utilisation. Il faut spécifier l'emplacement du widget dans le layout lors de l'appel à la méthode `addWidget()`, avec les paramètres *row* et *column*. Un widget occupe par défaut une unique case, mais il est possible de l'élargir avec le paramètre *rowSpan* ou le rendre plus haut avec *columnSpan*. Avec une valeur de -1, le widget est étendu jusqu'au bord droit ou bas du container.

9.5 Signaux et slots

Un *signal* est un message envoyé lors d'un évènement concernant un widget. Cela peut être une action de l'utilisateur (clic de la souris sur un bouton, survol d'une zone spécifique, ...) ou un évènement généré par l'objet lui-même (*timer* arrivant à la fin de son délai).

Un *slot* est une méthode appelée lorsqu'un certain évènement se produit. Un slot appartient souvent à la partie Modèle du programme : c'est ainsi que l'on peut agir sur l'état interne du programme par l'intermédiaire de l'interface graphique.

À partir de PyQt 4.5 (2009), on connecte un signal à un slot par la méthode `connect()` du signal concerné. Ici, `emetteur` est l'objet sur lequel se produit une action, `signal` est le nom du signal utilisé, `connect()` est une méthode de ce signal, et `slot` est la méthode appelée lorsque le signal est émis :

```
1   emetteur.signal.connect(slot)
```

Les lignes 38 à 40 du listing 9.2 illustrent ce mécanisme.

Un *signal* maintient une liste de *slots* auxquels il est connecté : un même signal peut participer à un nombre quelconque de connections. Chaque slot sera invoqué à chaque fois que le signal est émis. De même, un même slot peut être connecté à plusieurs signaux. La déconnection peut se faire par la méthode `disconnect()`. Si l'un des objets concernés (slot ou signal) par une connection entre signal et slot est détruit, la connection est également supprimée.

Un `emetteur` est l'objet sur lequel se produit l'évènement qui doit déclencher l'action. Il peut être lui-même à l'origine de cet évènement, comme dans le cas d'un timer.

Le `signal` est construit avec le nom du changement d'état de l'objet émetteur. Ce peut être `"valueChanged"` pour un slider, `"pressed"` ou `"clicked"` pour un bouton. La syntaxe des signaux est proche de celle des fonctions, mais il s'agit d'un mécanisme propre à Qt : on ne peut pas appeler un signal, mais on peut l'émettre avec la méthode `emit` du signal, comme en ligne 27 du listing 9.2.

Les signaux ne sont émis que lorsqu'il y a effectivement un changement d'état : réécrire une valeur 42 dans un *QDial* qui contenait une valeur 42 ne déclenche pas de signal *valueChanged*. Cela évite de créer des boucles infinies entre signaux *valueChanged* et slots *setValue* lors de la propagation d'une modification dans l'interface. Il est recommandé de reprendre cette sécurité pour l'émission de nouveaux signaux...

Le *slot* peut être n'importe quelle fonction ou méthode Python [4], ou une méthode identifiée comme slot au sein de la bibliothèque Qt [5]. Pour la réception d'un signal, il suffit de passer le slot sous la forme `recepteur.nomMethode` ou `nomFonction`.

Il est possible qu'un slot accepte plus de paramètres qu'un signal n'en fournit. Dans ce cas, les valeurs transmises sont affectées aux premiers paramètres, et les autres reçoivent leur valeur par défaut. Il est impossible de fournir plus de paramètres que prévu à un slot.

Souvent, il est préférable que la méthode ciblée comme *slot* interroge son environnement pour disposer des informations dont elle a besoin, plutôt que de lui passer cette information comme paramètre. Si la méthode appartient à un objet dont une variable d'instance cible le modèle, elle peut l'interroger (ligne 47 du listing 9.2). Sinon, elle peut interroger l'émetteur du signal, qui est `self.sender` à l'intérieur du slot.

Le programme suivant crée un *QDial* [6] et une *QSpinBox* [7]. Il les relie par des signaux pour synchroniser les valeurs des deux objets. Pour obtenir un affichage et un programme fonctionnel, il faut ajouter ces objets à la fenêtre [8], l'afficher, et lancer une *QApplication*.

```
1  class LinkedValues(QDialog):
2    def __init__(self, Parent=None):
3      dial = QDial()
4      spinbox = QSpinBox()
5      dial.valueChanged.connect(spinbox.setValue)
6      spinbox.valueChanged.connect(dial.setValue)
```

Il est possible de connecter un widget à une méthode python classique, et de générer manuellement un signal. Dans l'exemple 9.2 le modèle numérique est défini dans une classe *LotkaV*. Elle descend de *QObject*, ce qui la rend capable de connecter des signaux en émission : dans *setValue*, elle émet `valueChanged`. Les méthodes *run* et *randinit* servent de slots lorsqu'on clique sur les boutons.

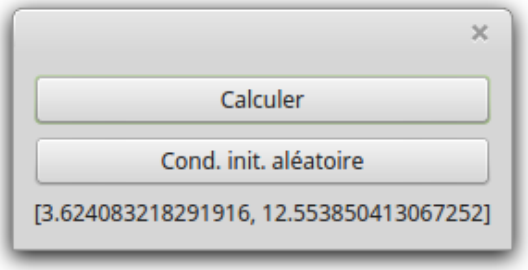
Listing 9.2 – Simulateur minimal d'EDO

```
1  import random
2  import PyQt5
3  from PyQt5.QtCore import QObject, pyqtSignal
4  from PyQt5.QtWidgets import QPushButton, QLabel, QVBoxLayout, QApplication,
     QDialog
5
6  class LotkaV(QObject):
```

4. Les interfaces Qt d'autres langages sont en général plus restrictives.
5. Bien lire la documentation Qt ; toutefois les méthodes qu'il est naturel d'invoquer en cas de changement d'état du programme sont des slots.
6. Une sorte de potentiomètre, cousin de QSlider
7. Boîte de dialogue pour donnée entière, avec boutons pour incrémenter et/ou décrémenter la valeur
8. La classe *LinkedValue* descend de *QDialog*, c'est donc une fenêtre, même si elle ne descend pas de *QMainWindow*. Il faut donc affecter un layout à `self`, y ajouter `dial` et `spinbox`, puis rendre la fenêtre visible avec `self.setVisible(True)`.

```
7       valueChanged = pyqtSignal()
8       def __init__(self, x0, tmax, dt):
9           super(LotkaV, self).__init__()
10          self.__val = [0, 0]
11          self.x0 = x0
12          self.dt = dt
13          self.tmax = tmax
14      def run(self):
15          t = 0
16          x = self.x0
17          while t<self.tmax:
18              x = [x[0] + self.dt*(x[0]*(1-0.5*x[1])),
19                   x[1] - self.dt*(x[1]*(1-0.5*x[0]))]
20              t += self.dt
21          self.setValue(x)
22      def randinit(self):
23          self.x0 = [random.randint(1, 20), random.randint(1, 20)]
24      def setValue(self, x):
25          if x != self.__val:
26              self.__val = x
27              self.valueChanged.emit()
28      def getValue(self):
29          return self.__val
30
31  class Simulateur(QDialog):
32      def __init__(self, parent=None):
33          super(QDialog, self).__init__(parent)
34          self.buttonStart = QPushButton("Calculer")
35          self.buttonInit = QPushButton("Cond. init. aléatoire")
36          self.result = QLabel("Unknown")
37          self.modele = LotkaV(x0 = [1, 12], tmax = 10, dt=0.01)
38          self.buttonStart.clicked.connect(self.modele.run)
39          self.buttonInit.clicked.connect(self.modele.randinit)
40          self.modele.valueChanged.connect(self.updateText)
41          layout = QVBoxLayout()
42          layout.addWidget(self.buttonStart)
43          layout.addWidget(self.buttonInit)
44          layout.addWidget(self.result)
45          self.setLayout(layout)
46      def updateText(self):
47          self.result.setText(str(self.modele.getValue()))
48
49  if __name__ == "__main__":
50      app = QApplication([])
51      sw = Simulateur()
52      sw.show()
53      app.exec_()
```

FIGURE 9.3 – Simulateur d'équation de Lotka-Volterra



9.5.1 Timer

Un objet *QTimer* permet de piloter le fonctionnement d'une application. Il émet un signal *timeout* au bout de chaque intervalle de temps défini : il suffit de connecter ce signal aux méthodes que l'on souhaite invoquer avec cet intervalle de temps.

```
game = Dungeon()                        # à écrire...
clock = QTimer()
clock.timeout.connect(game.moveMonsters)
game.death.connect(clock.stop) # death est un signal émis par un Dungeon
                               # à la mort du joueur ; les monstres
                               # et l'horloge s'arrêtent
clock.start(40)         # intervalle de 40ms, soit 25Hz
```

9.6 Dessiner

Chaque *QWidget* (ainsi que les autres descendants de *QPaintDevice*) est capable de se dessiner à l'écran. Pour cela, sa méthode *paintEvent()* est appelée. Éventuellement, cette méthode reçoit un paramètre de type *QPaintEvent* qui renseigne la zone de l'objet qui nécessite d'être redessinée. Si ce paramètre n'est passé, il faut redessiner l'intégralité de la zone couverte par le widget. Le slot *update* entraîne l'appel à *paintEvent* dès que Qt le juge approprié (pour éviter le scintillement ou rester raisonnable en temps de calcul). Le slot *repaint* entraîne un appel immédiat à *paintEvent*, ce qui est en général une moins bonne idée qu'*update*, sauf si l'animation est au cœur du programme (jeu temps réel, animation). Pour faire évoluer un widget en continu, sans action de l'utilisateur, une solution élégante est de connecter le signal *timeout* d'un *QTimer* à son slot *update*.

Lorqu'on crée un nouveau widget en héritant d'un widget existant, il est courant de modifier sa façon de s'afficher en redéfinissant sa méthode *paintEvent()*. L'outil permettant d'accéder aux fonctions de dessin élémentaires est *QPainter*. Les méthodes les plus courantes sont *drawEllipse*, *drawLine*, *drawRect*, *setFont*, *fillPath*, *setBrush*, *drawText*, *setFont*, *drawPixmap*, ...

Il convient de créer un *QPainter* adapté au widget à redessiner : il dispose alors du "bon" système de coordonnées. Attention, en informatique, l'origine du repère est dans le coin en haut à gauche, et le repère est indirect. L'unité de mesure est le pixel.

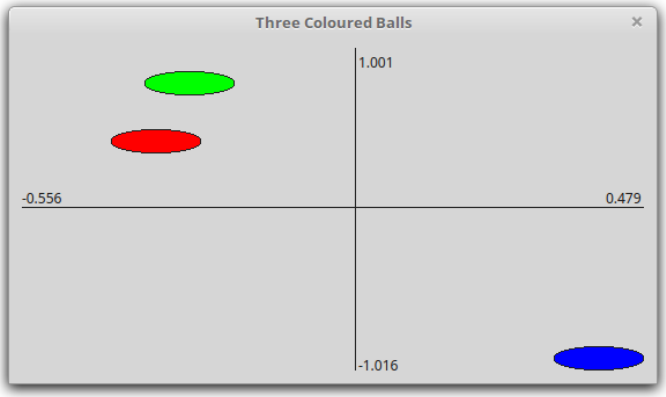
Il est *fortement recommandé* de ne pas utiliser les coordonnées utilisées pour l'affichage dans les calculs et le stockage d'information du programme : la trigonométrie ne "marcherait" plus, les nombres entiers sont en général peu adaptés, des fonctions de zoom sur une partie du modèle seraient pénibles à implémenter, ... Les méthodes *width()* et *height()* permettent de connaître la taille du widget, afin d'adapter la transformation des coordonnées du modèle vers celles du widget. Si la taille du widget change, *resizeEvent()* est appelée, qui invoque *paintEvent()*.

Listing 9.3 – Afficheur spécialisé

```
from random import random
import PyQt5
from PyQt5.QtCore import QObject, Qt
from PyQt5.QtGui import QPainter
from PyQt5.QtWidgets import QDialog, QWidget, QApplication, QVBoxLayout

class Balls:
    def __init__(self):
        self.ray = 0.15
        self.red = self.randcoord()
        self.green = self.randcoord()
        self.blue = self.randcoord()
    def randcoord(self):
        return (-1+2*random(), -1+2*random())
    def __str__(self):
```

FIGURE 9.4 – Fenêtre redimensionnée de *ShowBalls*




```
16          return "(%.2f %.2f) (%.2f %.2f) (%.2f %.2f) %.2f"%(
17              self.red + self.green + self.blue +(self.ray,))
18
19  class ShowBalls(QDialog):
20      def __init__(self, balls=None, parent=None):
21          super(QDialog, self).__init__(parent)
22          self.b = Balls()
23          layout = QVBoxLayout()
24          bw = BallsWidget(b)
25          layout.addWidget(bw)
26          self.setLayout(layout)
27          self.setMinimumSize(300, 300)
28          self.setWindowTitle("Three Coloured Balls")
29
30  class BallsWidget(QWidget):
31      def __init__(self, balls=None):
32          super(QWidget, self).__init__()
33          if balls:
34              self.balls = balls
35          else:
36              self.balls = Balls()
37          print(self.balls)          # pour débogage
38
39      def paintEvent(self, event = None):
40          # calcul de la zone d'intérêt du modèle
41          minx = min(0, self.balls.red[0], self.balls.green[0],
42                     self.balls.blue[0])-self.balls.ray
43          maxx = max(0, self.balls.red[0], self.balls.green[0],
44                     self.balls.blue[0])+self.balls.ray
45          miny = min(0, self.balls.red[1], self.balls.green[1],
46                     self.balls.blue[1])-self.balls.ray
47          maxy = max(0, self.balls.red[1], self.balls.green[1],
48                     self.balls.blue[1])+self.balls.ray
49          dx = maxx-minx
50          dy = maxy-miny
51
52          # outil dans les coordonnées du widget
53          qp = QPainter(self)
54          w = self.width()
55          h = self.height()
56
57          # tracé des axes
58          qp.drawLine(0, h+miny/dy*h, w, h+miny/dy*h)
59          qp.drawLine(-minx/dx*w, 0, -minx/dx*w, h)
```

```
60
61          # tracé des balles
62          for (pt, col) in [(self.balls.red, Qt.red), (self.balls.green, Qt.green),
63                            (self.balls.blue, Qt.blue)]:
64              qp.setBrush(col)
65              qp.drawEllipse((pt[0]-minx)/dx*w, h-(pt[1]-miny)/dy*h,
66                             self.balls.ray/dx*w, self.balls.ray/dy*h)
67
68          # tracé des coordonnées
69          qp.drawText(0, h+miny/dy*h-3, "%.3f"%minx)
70          txtwidth = self.fontMetrics().boundingRect("%.3f"%maxx).width()
71          qp.drawText(w-txtwidth, h+miny/dy*h-3, "%.3f"%maxx)
72          qp.drawText(-minx/dx*w+3, h, "%.3f"%miny)
73          txtheight = self.fontMetrics().boundingRect("%.3f"%maxy).height()
74          qp.drawText(-minx/dx*w+3, txtheight, "%.3f"%maxy)
75
76
77  if __name__ == "__main__":
78      app = QApplication([])
79      b = Balls()
80      sb = ShowBalls(b)
81      sb.show()
82      app.exec_()
```

La différence essentielle entre un objet qui peut connaître une défaillance et un objet qui ne peut absolument pas connaître la moindre défaillance est que lorsqu'un objet qui ne peut absolument pas connaître la moindre défaillance connaît une défaillance, il s'avère généralement impossible à remplacer ou réparer.

Douglas Adams

A Conventions

Sommaire

Pour qu'un programme puisse être facilement réutilisé, il est important que le développeur rencontre des programmes ayant toujours la même organisation et la même présentation. Ce chapitre présente quelques conventions généralement utilisées lors de l'écriture de programmes Python. Le langage ne nous impose pas de les suivre, mais, étant suivies par la plupart des programmeurs, elles constituent des habitudes intéressantes à prendre. Ces conventions sont décrites plus en détail dans [VWC05]. L'outil PEP8 permet de vérifier qu'un programme Python respecte ces conventions.

Exemple A.1 (Exemple de source Python)**.** *Le listing suivant est un extrait de code source de* pep8.py*.*

Listing A.1 – Conventions en Python

```
#!/usr/bin/env python3
# pep8.py - Check Python source code formatting, according to PEP 8
# Copyright (C) 2006-2009 Johann C. Rocholl <johann@rocholl.net>
# Copyright (C) 2009-2013 Florent Xicluna <florent.xicluna@gmail.com>
#

"""
```

```
8  Check Python source code formatting, according to PEP 8:
9  http://www.python.org/dev/peps/pep-0008/
10
11 This program and its regression test suite live here:
12 http://github.com/jcrocholl/pep8
13 """
14 __version__ = '1.4.6'
15
16 import os
17 import sys
18 import re
19
20 ##############################################################################
21 # Plugins (check functions) for physical lines
22 ##############################################################################
23
24
25 def tabs_or_spaces(physical_line, indent_char):
26     """
27     Never mix tabs and spaces.
28
29     The most popular way of indenting Python is with spaces only.  The
30     second-most popular way is with tabs only.  Code indented with a mixture
31     of tabs and spaces should be converted to using spaces exclusively.  When
32     invoking the Python command line interpreter with the -t option, it issues
33     warnings about code that illegally mixes tabs and spaces.  When using -tt
34     these warnings become errors.  These options are highly recommended!
35
36     Okay: if a == 0:\n        a = 1\n        b = 1
37     E101: if a == 0:\n        a = 1\n\tb = 1
38     """
39     indent = INDENT_REGEX.match(physical_line).group(1)
40     for offset, char in enumerate(indent):
41         if char != indent_char:
42             return offset, "E101 indentation contains mixed spaces and tabs"
43
44
45 ##############################################################################
46 # Framework to run all checks
47 ##############################################################################
48
49
50 class Checker(object):
51     """
52     Load a Python source file, tokenize it, check coding style.
53     """
54
55     def __init__(self, filename=None, lines=None,
56                  options=None, report=None, **kwargs):
57         if options is None:
58             options = StyleGuide(kwargs).options
59         else:
60             assert not kwargs
61
62     def readline(self):
63         """
64         Get the next line from the input buffer.
65         """
66         self.line_number += 1
67         if self.line_number > len(self.lines):
68             return ''
69         return self.lines[self.line_number - 1]
```

A.1 Organisation des fichiers

Chaque fichier Python doit rester de taille raisonnable. Il est conseillé d'éviter les fichiers de plus de 1000 lignes. L'exemple A.1 est un extrait du code de *pep8.py* qui respecte les conventions.

Un fichier source est organisé de la manière suivante :

— il commence par une zone de commentaires décrivant le contenu du fichier ;
— il contient éventuellement des importations ;
— il contient ensuite une déclaration de variables, de fonctions ou de classes.

Un exemple de commentaire de début de fichier est décrit dans l'exemple A.1.

A.2 Règles générales

A.2.1 Indentation

L'indentation préconisée par PEP8 est de 4 espaces par niveau. Si une instruction est écrite sur plusieurs lignes, l'indentation doit en permettre une lecture aisée.

Le listing A.2 représente des exemples d'indentation correcte, et le listing A.3 des exemples incorrects.

Listing A.2 – Indentation correcte

```
# Alignement sur l'opérateur ouvrant : ici la parenthèse
res = une_jolie_fonction(var_un, var_deux,
                         var_trois, var_quatre)

# Indentation permettant de distinguer la fonction de
# la suite du programme
def une_jolie_fonction(
        var_un, var_deux, var_trois,
        var_quatre):
    print(var_un)
```

Listing A.3 – Indentation incorrecte

```
# La seconde ligne d'arguments n'est pas alignée avec la première
res = une_jolie_fonction(var_un, var_deux,
    var_trois, var_quatre)

# On ne distingue pas la déclaration de la fonction de
# la suite du programme
def une_jolie_fonction(
    var_un, var_deux, var_trois,
    var_quatre):
    print(var_un)
```

Des règles similaires sont applicables aux listes et aux appels de fonctions. Dans ce cas, le caractère de fermeture de la liste ou de la fonction peut être aligné avec le début de l'énumération (voir listing A.4) ou avec l'instruction (voir listing A.5).

Listing A.4 – Alignement de liste version 1

```
ma_liste = [
    1, 2, 3,
    4, 5, 6,
    ]
res = une_fonction_avec_plusieurs_parametres(
```

```
    'a', 'b', 'c',
    'd', 'e', 'f',
    )
```

Listing A.5 – Alignement de liste version 2

```
ma_liste = [
    1, 2, 3,
    4, 5, 6,
]
res = une_fonction_avec_plusieurs_parametres(
    'a', 'b', 'c',
    'd', 'e', 'f',
)
```

A.2.2 Longueur des lignes

Il est conseillé de limiter la taille des lignes à 79 caractères. Les lignes de plus de 80 caractères doivent être coupées, par exemple à l'intérieur des parenthèses comme indiqué dans les exemples A.4 et A.5.

Dans le cas où il n'y a pas de parenthèse, crochet ou accolade à l'intérieur duquel faire la coupure, on utilisera le caractère antislash (\) comme représenté dans le listing A.6.

Listing A.6 – Utilisation de l'antislash

```
with open('/chemin/vers/le/fichier/a/lire') as fichier_1, \
     open('/chemin/vers/le/fichier/a/ecrire', 'w') as fichier_2:
    fichier_2.write(fichier_1.read())
```

De manière générale, il faut veiller à couper les lignes pour que le programme soit le plus lisible possible. En particulier, il faut couper les lignes après les opérateurs et pas avant. Le listing A.7 illustre ces bonnes pratiques.

Listing A.7 – Coupure de ligne

```
class Rectangle(Blob):

    def __init__(self, width, height,
                 color='black', emphasis=None, highlight=0):
        if (width == 0 and height == 0 and
                color == 'red' and emphasis == 'strong' or
                highlight > 100):
            raise ValueError("sorry, you lose")
        if width == 0 and height == 0 and (color == 'red' or
                                           emphasis is None):
            raise ValueError("I don't think so -- values are %s, %s" %
                             (width, height))
        Blob.__init__(self, width, height,
                      color, emphasis, highlight)
```

A.2.3 Gestion des espaces

Pour augmenter la lisibilité du code, il est conseillé d'entourer les opérateurs binaires d'un espace de chaque côté. Si une expression combine des opérateurs de priorités différentes, ajouter des espaces autour des opérateurs de priorité la plus faible. De manière générale, il faut faire en sorte que l'expression soit la plus facile à lire possible.

Le listing A.8 représente une bonne gestion, et le listing A.9 une mauvaise gestion des espaces.

Listing A.8 – Espaces bien gérés

```
i = i + 1
valeur += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

Listing A.9 – Espaces mal gérés

```
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

Attention, il ne faut pas entourer le signe = d'espaces dans le cas des paramètres par défaut (voir listing A.10).

Listing A.10 – Valeur par défaut

```
# Exemple correct
def complex(real, imag=0.0):
    return magic(r=real, i=imag)

# Exemple incorrect
def complex(real, imag = 0.0):
    return magic(r = real, i = imag)
```

A.3 Règles de nommage

Le langage Python laisse de grandes libertés dans le choix des différents noms. Cependant, certaines conventions permettent de rendre un programme plus lisible et facilitent l'identification des classes, variables et méthodes.

A.3.1 Règles générales

De manière générale, il est conseillé d'utiliser principalement des minuscules. Si un nom (de variable, de fonction, de méthode) est composé de plusieurs mots, les séparer par un tiret bas [1] (caractère _). Dans le cas d'un nom de classe, mettre la première lettre de chaque mot en majuscule.

Exemples : *un_joli_nom_de_variable*, *UnSuperbeNomDeClasse*.

Il est possible de nommer les différentes entités en français ou en anglais, mais il est déconseillé de mélanger les deux. Si votre programme doit être repris par d'autres personnes, utiliser de préférence l'anglais. Si vous choisissez le français, il est d'usage de ne pas utiliser la version accentuée des caractères.

A.3.2 Classes

Voici les règles générales de nommage des classes :

— les noms de classes doivent être des substantifs (des noms) et commencer par une majuscule ;

1. *underscore* en anglais, touche 8 sur les claviers français.

— il est conseillé d'employer des noms complets et d'éviter acronymes et abréviations (sauf pour des acronymes très répandus comme par exemple HTML).

Exemples de noms de classes : *ListeSimplementChainee*, *Insecte*.

A.3.3 Méthodes

Voici quelques contraintes dans le choix des noms de méthodes :
— utiliser des verbes, à l'infinitif ;
— utiliser uniquement des minuscules.
Exemples : *empiler*, *manger*, *extract_color*.

Certains noms de méthodes sont encadrés par `__`. Ce sont les méthodes qui permettent de définir le comportement des opérateurs et de certains mots-clefs du langage Python lorsqu'ils sont appliqués à un objet de cette classe. À moins de développer la version 4.0 du langage, *il ne faut pas* créer de méthode utilisant un tel nom, mais il tout à fait raisonnable de redéfinir des méthodes portant de tels noms déjà définis dans le langage.
Exemples : `__str__`, `__len__`, `__add__`, `__le__`.

A.3.4 Variables

Lors du choix de noms de variables, il est conseillé de suivre les règles suivantes :
— utiliser des substantifs ;
— utiliser des noms courts mais significatifs ;
— utiliser uniquement des minuscules ;
— n'utiliser les noms d'un seul caractère que dans le cas des variables temporaires (indices de boucles par exemple) ; pour éviter les confusions, ne pas utiliser les lettres l (L minuscule), O (o majuscule) et I (i majuscule).
Exemples : *i, j* pour des entiers, *x, y* pour des réels, *c* pour un caractère, *abscisse*, *nombre_elements*.

On pourra faire commencer un nom de variable par `_` pour représenter une variable *privée*. Son accès n'est pas restreint de façon informatique (cf. accesseurs et décorateurs section 5.3.2) mais son nom recommande la plus grande prudence lorsqu'on la lit ou l'écrit directement. Il existe au moins une méthode permettant d'accéder à l'information de cette variable, et il est recommandé de l'utiliser. Lors de l'importation d'un module avec `from le_module import *`, les noms commençant par `_` ne sont pas pris en compte.
Exemples : *_flags*, *_index*, *_passwd_hashcode*

Précéder le nom de variable par `__` fournit une protection plus explicite (le nom réel n'est pas celui-là, et la variable `__x` de la classe `Point` s'appelle en fait `_Point__x`) et permet d'éviter d'éventuels conflits de nommage.

A.3.5 Constantes

Comme dans le cas des variables, il est conseillé de nommer les constantes en utilisant des substantifs. Il est également conseillé d'employer les règles suivantes :
— les noms des constantes sont entièrement en majuscule ;
— si un nom de constante comporte plusieurs mots, les séparer par des *underscore*.
Exemples : *NOMBRE_MAX*, *ETAT_INITIAL*.

Quarante-deux, dit Compute-Un, avec infiniment de calme et de majesté.

Douglas Adams

B

Génération de documentation

L'outil *sphinx* permet de générer une documentation HTML associée à un programme python. Il se base sur les commentaires du programme.

B.1 Génération de documentation

L'utilisation de *sphinx* demande quelques étapes. Le site http://sphinx-doc.org/tutorial.html décrit la procédure à suivre pour le mettre en œuvre.

Partant d'un programme commenté tel que le listing B.5 dans un répertoire *prog*, la procédure pour générer la documentation est la suivante :

- lancer le programme *sphinx-quickstart*. Un extrait de cette partie est décrit dans le listing B.1.
- mettre à jour le fichier *source/conf.py* ; en particulier, préciser le chemin des sources Python grâce à la fonction *sys.path.insert*. Pour que sphinx supporte les commentaires au format *numpy*, il faut également charger l'extension *napoleon*. Le listing B.2 recense ces modifications.
- utiliser la commande *sphinx-apidoc* pour générer les fichiers *rst* correspondant au module.
- compiler la documentation grâce à la commande *make*, comme indiqué dans le listing B.4.

Listing B.1 – Exécution du *quickstart*

```
$ ls -l
total 4
drwxrwxr-x 2 roderic roderic 4096 juin  20 17:21 prog
$ ls -l prog
total 4
-rw-r--r-- 1 roderic roderic 3578 juin  20 17:18 example.py
$ sphinx-quickstart
Welcome to the Sphinx 1.3.6 quickstart utility.

Please enter values for the following settings (just press Enter to
accept a default value, if one is given in brackets).
$
```

Listing B.2 – Fichier conf.py

```
# If extensions (or modules to document with autodoc) are in another
# directory, add these directories to sys.path here. If the directory
# is relative to the documentation root, use os.path.abspath to make
# it absolute, like shown here.
#sys.path.insert(0, os.path.abspath('.'))
sys.path.insert(0, os.path.abspath('../../prog'))

# Add any Sphinx extension module names here, as strings. They can be
# extensions coming with Sphinx (named 'sphinx.ext.*') or your custom
# ones.
extensions = [
    'sphinx.ext.autodoc',
    'sphinx.ext.doctest',
    'sphinx.ext.mathjax',
    'sphinx.ext.napoleon',
]
```

Listing B.3 – Création des fichiers *rst*

```
$ sphinx-apidoc -o source ../prog
Creating file source/example.rst.
Creating file source/modules.rst.
$
```

Listing B.4 – Compilation de la documentation sphinx

```
$ cd doc
$ make html
sphinx-build -b html -d build/doctrees   source build/html
Running Sphinx v1.3.6
loading translations [fr]... done
loading pickled environment... done
building [mo]: targets for 0 po files that are out of date
building [html]: targets for 0 source files that are out of date
updating environment: 0 added, 0 changed, 0 removed
looking for now-outdated files... none found
no targets are out of date.
build succeeded.

Build finished. The HTML pages are in build/html.
$ ls -l build
total 8
drwxrwxr-x 2 moitiero roderic 4096 juin  21 08:40 doctrees
drwxrwxr-x 4 moitiero roderic 4096 juin  21 08:53 html
$
```

B.2 Exemple de code commenté

Exemple de commentaire complet extrait de la documentation de *numpy* : https://github.com/numpy/numpy/blob/master/doc/example.py

Listing B.5 – Exemple de commentaire *sphinx*

```
"""This is the docstring for the example.py module.  Modules names
```

```
should have short, all-lowercase names.  The module name may have
underscores if this improves readability.

Every module should have a docstring at the very top of the file.  The
module's docstring may extend over multiple lines.  If your docstring
does extend over multiple lines, the closing three quotation marks
must be on a line by itself, preferably preceeded by a blank line.

"""
from __future__ import division, absolute_import, print_function

import os # standard library imports first

# Do NOT import using *, e.g. from numpy import *
#
# Import the module using
#
#   import numpy
#
# instead or import individual functions as needed, e.g
#
#  from numpy import array, zeros
#
# If you prefer the use of abbreviated module names, we suggest the
# convention used by NumPy itself::

import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt

# These abbreviated names are not to be used in docstrings; users must
# be able to paste and execute docstrings after importing only the
# numpy module itself, unabbreviated.

#from my_module import my_func, other_func

def foo(var1, var2, long_var_name='hi') :
    r"""A one-line summary that does not use variable names or the
    function name.

    Several sentences providing an extended description. Refer to
    variables using back-ticks, e.g. `var`.

    Parameters
    ----------
    var1 : array_like
        Array_like means all those objects -- lists, nested lists, etc.
          --
        that can be converted to an array.  We can also refer to
        variables like `var1`.
    var2 : int
        The type above can either refer to an actual Python type
        (e.g. ``int``), or describe the type of the variable in more
        detail, e.g. ``(N,) ndarray`` or ``array_like``.
    Long_variable_name : {'hi', 'ho'}, optional
        Choices in brackets, default first when optional.
```

```
57
58      Returns
59      -------
60      type
61          Explanation of anonymous return value of type ``type``.
62      describe : type
63          Explanation of return value named `describe`.
64      out : type
65          Explanation of `out`.
66
67      Other Parameters
68      ----------------
69      only_seldom_used_keywords : type
70          Explanation
71      common_parameters_listed_above : type
72          Explanation
73
74      Raises
75      ------
76      BadException
77          Because you shouldn't have done that.
78
79      See Also
80      --------
81      otherfunc : relationship (optional)
82      newfunc : Relationship (optional), which could be fairly long, in
83                which case the line wraps here.
84      thirdfunc, fourthfunc, fifthfunc
85
86      Notes
87      -----
88      Notes about the implementation algorithm (if needed).
89
90      This can have multiple paragraphs.
91
92      You may include some math:
93
94      .. math:: X(e^{j\omega } ) = x(n)e^{ - j\omega n}
95
96      And even use a greek symbol like :math:`omega` inline.
97
98      References
99      ----------
100     Cite the relevant literature, e.g. [1]_.  You may also cite these
101     references in the notes section above.
102
103     .. [1] O. McNoleg, "The integration of GIS, remote sensing,
104        expert systems and adaptive co-kriging for environmental habitat
105        modelling of the Highland Haggis using object-oriented,
106        fuzzy-logic and neural-network techniques," Computers &
107        Geosciences, vol. 22, pp. 585-588, 1996.
108
109     Examples
110     --------
111     These are written in doctest format, and should illustrate how to
112     use the function.
```

```
113
114     >>> a=[1,2,3]
115     >>> print [x + 3 for x in a]
116     [4, 5, 6]
117     >>> print "a\n\nb"
118     a
119     b
120
121     """
122
123     pass
```

Ce n'est qu'en essayant continuellement que l'on finit par réussir. Autrement dit : plus ça rate, plus on a de chances que ça marche.

Proverbe Shadok

C
Pièges classiques

Sommaire

Il existe quelques pièges très classiques dans lesquels il est extrêmement facile de tomber. Le but de cet annexe est d'en recenser quelques-uns. Le programmeur débutant se fera sûrement piéger par au moins l'un d'entre eux, et il n'est pas rare qu'il les rencontre tous...

C.1 Effets de bord

Voici un exemple d'effet de bord dû à une utilisation maladroite de variable d'instance :

Listing C.1 – Problème avec un effet de bord

```
class EffetDeBord():
    def __init__(self):
        self.i = 0

    def boucle1(self):
        self.i = 0
        while self.i <10:
            print('boucle1 : ',self.i)
            self.i += 1

    def boucle2(self):
        self.i = 0
        while self.i < 10:
            print('boucle2 : ',self.i)
            self.boucle1()
            self.i += 1

if __name__ == "__main__":
```

```
    x = EffetDeBord()
    x.boucle2()
```

Dans cet exemple, la variable i est une variable d'instance, donc accessible depuis toutes les méthodes d'instance de la classe. Elle est en particulier accessible depuis *boucle1* et *boucle2*. Il est important de réaliser qu'il s'agit de la même variable accessible depuis les deux méthodes.

Étudions l'appel à *boucle2* : la variable i est initialisée à 0. Le test $i < 10$ est vrai, donc la boucle est exécutée. À l'intérieur de la boucle, *boucle1* est appelée. Cette méthode initialise i à 0, et exécute une action tant que $i < 10$. Après cette boucle, i vaut donc 10. L'exécution de *boucle2* continue, mais i vaut maintenant 10, et la boucle se termine !

Donc, notre programme va exécuter une seule fois *boucle1*.

Remarquons que si *boucle1* exécutait la boucle pour i allant de 1 à 5, alors *boucle2* ne se termine jamais.

C.2 Quelques perles

Cette section présente quelques exemples (écrits en différents langages ; ici du C, du Java et du Python) tirés de http://thedailywtf.com. Ce site recense quelques exemples de code à ne surtout pas suivre. Il s'agit de « vrai » code, c'est-à-dire de code effectivement utilisé dans des entreprises. Il s'agit également de code développé par des salariés, pas par des étudiants...

S'il est recommandé de commenter son programme, il ne faut pas tomber dans un excès de détails, comme dans l'exemple C.2. De même, donner des noms parlants aux méthodes constitue une bonne pratique, mais les noms des exemples C.3 et C.4 sont excessifs.

Listing C.2 – commentaire excessivement détaillé

```
SQWORD GetGlobalTime( const TCHAR* Filename )
  {
    //return greenwich mean time as expressed in nanoseconds since the
    //creation of the universe.  time is expressed in meters, so
    //divide by the speed of light to obtain seconds.  assumes the
    //speed of light in a vacuum is constant.  the file specified by
    //Filename is assumed to be in your reference frame, otherwise you
    //must transform the result by the path integral of the minkowski
    //metric tensor in order to obtain the correct result.

    return Time;
  }
```

Listing C.3 – des noms de méthodes très parlants

```
synchronized (surelyReachableObjectsWhichHaveToBeMarkedAsSuch) {
  waitRecommended =
    surelyReachableObjectsWhichShouldHaveBeenProcessedButWereLockContentedSize
      == surelyReachableObjectsWhichShouldHaveBeenProcessedButWereLockContented.size();

    surelyReachableObjectsWhichShouldHaveBeenProcessedButWereLockContentedSize =
      surelyReachableObjectsWhichShouldHaveBeenProcessedButWereLockContented.size();

    while(!surelyReachableObjectsWhichShouldHaveBeenProcessedButWereLockContented.isEmpty())
    {
      surelyReachableObjectsWhichHaveToBeMarkedAsSuch.push(
        surelyReachableObjectsWhichShouldHaveBeenProcessedButWereLockContented.getFirst());
    }
}
```

Listing C.4 – un nom de méthode assez clair ?

```
GetProfileCustomerEntityReceiverInformationReceiverAndProgrammingInformationListAccess
CardInformationProgrammingListProductDetails()
```

La plupart des langages de programmation possèdent la notion de boucle. Hélas, certains programmeurs l'oublient, comme dans l'exemple C.5.

Listing C.5 – déjà entendu parler de boucles ?

```
annualMileageChoice = new Choice();
annualMileageChoice.addItem(CHO_PLEASE_ENTER);
annualMileageChoice.addItem("500");
annualMileageChoice.addItem("1000");
annualMileageChoice.addItem("1500");
annualMileageChoice.addItem("2000");
annualMileageChoice.addItem("2500");
// ...
annualMileageChoice.addItem("44000");
annualMileageChoice.addItem("44500");
annualMileageChoice.addItem("45000");
```

Un problème récurrent en informatique est le *padding*, qui consiste à compléter un nombre en ajoutant des zéros à gauche. Ce problème est généralement résolu en une ligne. Le listing C.6 décrit une méthode différente... à ne pas suivre.

Listing C.6 – Une jolie méthode de padding

```
public static String zeroFill(String data) {
  int length = data.length();
  String newData = new String();
  if (length >= 100) {
    newData = data.substring(0, length);
  }
  if (length == 99) {
    newData = "0" + data.substring(length - 99, length);
  }
  if (length == 98) {
    newData = "00" + data.substring(length - 98, length);
  }
  if (length == 97) {
    newData = "000" + data.substring(length - 97, length);
  }
  /*
   * ... CODE coupé pour des raisons de lisibilité ...
   */
  if (length == 2) {
    newData = "00000000000000000000000" + "00000000000000000"
      + "00000000000000000" + "00000000000000000"
      + "00000000000000000000000" + data.substring(length - 2, length);
  }
  if (length == 1) {
    newData = "00000000000000000000000" + "00000000000000000"
      + "00000000000000000" + "00000000000000000"
      + "00000000000000000000000" + data.substring(length - 1, length);
  }
  return newData;
}
```

L'indentation étant signifiante en Python, les programmes sont généralement bien présentés. Cependant, un excès de structures de contrôle imbriquées peut avoir des effets intéressants sur la forme du programme. Le listing C.7 illustre cet effet et montre également que le programmeur ne connaît pas la notion de complexité algorithmique (voir l'annexe D pour une définition de la complexité).

Listing C.7 – Génération de mot de passe originale

```
def genpassword(wlc,maxchar,txt,List,verbose):
   word = ""
   i1 = i2 = i3 = i4 = i5 = i6 = i6 = i7 = i8 = i9 = i10 = i11 = i12 = i13 = i14 =
      i15 = 0
   txtfile = open(txt,'w')

   i = 0
   mc = int(maxchar) - 1
   lword = [0]
   for i in range(mc):
      lword += [0]

   for i1 in range(len(wlc)):
      for i2 in range(len(wlc)):
         for i3 in range(len(wlc)):
            for i4 in range(len(wlc)):
               for i5 in range(len(wlc)):
                  for i6 in range(len(wlc)):
                     for i7 in range(len(wlc)):
                        for i8 in range(len(wlc)):
                           for i9 in range(len(wlc)):
                              for i10 in range(len(wlc)):
                                 for i11 in range(len(wlc)):
                                    for i12 in range(len(wlc)):
                                       for i13 in range(len(wlc)):
                                          for i14 in range(len(wlc)):
                                             for i15 in range(len(wlc)):
                                                if int(maxchar) == 1 :
                                                   word = wlc[i15]
                                                if int(maxchar) == 2 :
                                                   word = wlc[i14] + wlc[i15]
                                                if int(maxchar) == 3 :
                                                   word = wlc[i13] + wlc[i14] +
                                                     wlc[i15]
                                                if int(maxchar) == 4 :
                                                   word =
                                                   wlc[i12] +
                                                   wlc[i13] +
                                                   wlc[i14] +
                                                   wlc[i15]
# ...
# CODE coupé pour des raisons de lisibilité
# ...
                                                if end == 1 : break
                                             if end == 1 : break
                                          if end == 1 : break
                                       if end == 1 : break
                                    if end == 1 : break
                                 if end == 1 : break
                              if end == 1 : break
                           if end == 1 : break
                        if end == 1 : break
                     if end == 1 : break
                  if end == 1 : break
               if end == 1 : break
            if end == 1 : break
         if end == 1 : break
      if end == 1 : break

   txtfile.close()
```

. . . toutefois, le déroulement de ce programme va me prendre un petit moment.
– Combien de temps ?
– Sept millions et demi d'années, répondit Compute-Un.

Douglas Adams

D

Notions de complexité algorithmique

Sommaire

D.1 Complexité en temps

La notion de complexité permet d'évaluer l'efficacité des algorithmes. Elle permet de répondre à la question : entre différents algorithmes réalisant une même tâche, quel est le plus rapide et dans quelles conditions ?
Une approche indépendante des facteurs matériels est nécessaire pour évaluer cette efficacité. Donald Knuth fut un des premiers à l'appliquer systématiquement dans [Knu68]. La complexité d'un algorithme permet d'évaluer ses performances. Elle représente son nombre d'opérations caractéristiques. Elle est généralement évaluée en fonction de la taille N des données d'entrée.

Elle peut être évaluée dans le pire des cas, c'est-à-dire pour la répartition des données provoquant le plus d'opérations mais il est également possible de calculer la complexité moyenne, c'est-à-dire la moyenne des complexités pour toutes les données d'entrée possibles.

Remarque : la complexité en moyenne est généralement beaucoup plus difficile à calculer que dans le pire des cas, mais elle donne souvent des informations plus pertinentes. Elle s'appuie toutefois sur une loi de probabilité sur les données qui peut être difficile à obtenir.

Exemple D.1 (Complexité du tri bulles)**.** *Considérons le cas du tri bulle (algorithme 3.2).*
Le nombre d'opérations effectuées par cet algorithme sur un tableau de taille N est :

$$\sum_{i=1}^{N-1} i = \frac{N(N-1)}{2} = \frac{N^2}{2} - \frac{N}{2} = \Theta(N^2)$$

D.2 Complexité en espace

On appelle complexité en espace la quantité de mémoire utilisée par un algorithme. Cette complexité devrait en toute rigueur s'évaluer en nombre d'octets, mais en pratique elle s'évalue souvent en nombre de mots mémoire (caractères, entiers, réels, ...)

Exemple D.2 (Complexité en espace du tri bulles)**.** *Toujours dans le cas du tri bulle (algorithme 3.2) :*
la quantité de mémoire nécessaire pour effectuer un tri bulle sur un tableau de taille N est $N+1$ (N variables pour le tableau, et une pour l'échange. Les indices ne sont pas pris en compte).

D.3 Ordre de grandeur

Il est généralement peu utile d'avoir l'expression exacte du temps de calcul d'un algorithme en fonction de la taille des données d'entrée. Dans la plupart des cas, seul l'ordre de grandeur de la complexité nous intéresse. On ne considérera donc que le terme dominant de la complexité. Par exemple dans le cas d'une complexité valant $an^2 + bn + c$, seul le terme an^2 sera pris en compte.
De même, on ignorera le coefficient multiplicateur constant du terme prépondérant puisque les facteurs constants sont moins importants que l'ordre de grandeur dans l'évaluation de l'efficacité d'un algorithme. Nous dirons donc que la complexité du tri bulle est de $\Theta(n^2)$.

Remarque : la notation Θ sera définie dans le paragraphe D.4.

D.4 Notation asymptotique

Cette partie définit les notations asymptotiques utilisées lors des calculs de complexité. Elles sont introduites par exemple dans [Cor+94] et bien entendu dans [Knu68].

D.4.1 Notation Θ

Pour une fonction $g(n)$ donnée, on note $\Theta(g(n))$ l'ensemble des fonctions suivant :

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2 \in \mathbb{R}^{+\star}, n_0 \in \mathbb{N} / \forall n \geqslant n_0, 0 < c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)\}$$

$\Theta(g(n))$ représente l'ensemble des fonctions de même ordre que $g(n)$. Une fonction $f(n)$ appartient à $\Theta(g(n))$ s'il existe deux constantes positives c_1 et c_2 telles que $f(n)$ puisse être encadrée par $c_1 g(n)$ et par $c_2 g(n)$.
La fonction $f(n)$ est donc égale à la fonction $g(n)$ à un facteur multiplicateur près.

Remarque : $\Theta(g(n))$ représente un ensemble, et il faut donc écrire en toute rigueur $f(n) \in \Theta(g(n))$. Il est cependant d'usage d'utiliser l'abus de notation $f(n) = \Theta(g(n))$.

Exemple D.3 (Notation Θ)**.** *Quelques exemples d'utilisation de* Θ *:*
- *la complexité du tri bulle s'écrit* $\Theta(n^2)$ *;*
- *la complexité du tri par segmentation est* $\Theta(n^2)$ *dans le pire des cas et* $\Theta(n \log n)$ *en moyenne ;*
- *une complexité indépendante du nombre de données en entrée s'écrit* $\Theta(n^0)$ *ou encore* $\Theta(1)$*. La deuxième notation est celle le plus couramment utilisée.*

D.4.2 Notation $\mathcal{O}$

La notation $\mathcal{O}$ est utilisée dans le cas où nous ne disposons que d'une majoration de la complexité. Partant d'une fonction $g(n)$, l'ensemble $\mathcal{O}(g(n))$ est défini par :

$$\mathcal{O}(g(n)) = \{f(n) : \exists c \in \mathbb{R}^{+\star}, n_0 \in \mathbb{N} / \forall n \geqslant n_0, 0 < f(n) \leqslant cg(n)\}$$

La notation $\mathcal{O}$ sert à majorer une fonction à un facteur constant près.

Remarque : contrairement à $\Theta(g(n))$, $\mathcal{O}(g(n))$ ne représente qu'une majoration de $g(n)$ et contient donc plus que les fonctions du même ordre que g.

On remarquera que :

$$f(n) \in \Theta(g(n)) \Rightarrow f(n) \in \mathcal{O}(g(n))$$

ce qui peut également s'écrire :

$$\Theta(g(n)) \subseteq \mathcal{O}(g(n))$$

Exemple D.4 (Utilisation de la notation $\mathcal{O}$)**.** *Soit* $f(n) = an + b$*. On montre facilement (la preuve est laissée à titre d'exercice) que* $f(n) \in \mathcal{O}(n)$ *mais également, ce qui peut paraître plus surprenant, que* $f(n) \in \mathcal{O}(n^2)$*.*

Remarque :

$$f(n) \in \Theta(g(n)) \Longleftrightarrow \begin{cases} f(n) \in \mathcal{O}(g(n)) \\ g(n) \in \mathcal{O}(f(n)) \end{cases}$$

D.4.3 Notation Ω

La notation Ω est comparable à la notation $\mathcal{O}$, mais elle définit une borne inférieure asymptotique au lieu d'une borne supérieure asymptotique.

Étant donnée une fonction $g(n)$, on définit $\Omega(g(n))$ par :

$$\Omega(g(n)) = \{f(n) : \exists c \in \mathbb{R}^{+}, n_0 \in \mathbb{N} / \forall n \geqslant n_0, 0 \leqslant cg(n) \leqslant f(n)\}$$

Remarque : soient deux fonctions $f(n)$ et $g(n)$.

$$f(n) = \Theta(g(n)) \Longleftrightarrow f(n) = \mathcal{O}(g(n)) \text{ et } f(n) = \Omega(g(n))$$

D.4.4 Opérations de base sur les fonctions

Cette partie présente quelques résultats élémentaires sur les calculs de complexité.

$$
\begin{aligned}
f(n) &= \Theta(f(n)) \\
c.\Theta(f(n)) &= \Theta(f(n)) \\
\Theta(f(n)) + \Theta(f(n)) &= \Theta(f(n)) \\
\Theta(\Theta(f(n))) &= \Theta(f(n)) \\
\Theta(f(n))\Theta(g(n)) &= \Theta(f(n)g(n)) \\
\Theta(f(n)g(n)) &= f(n)\Theta(g(n)) \\
\sum_{i=0}^{m} a_i n^i &= \Theta(n^m) \\
\forall \alpha \in \mathbb{R}^{+\star}, n^\alpha + \log n &= \Theta(n^\alpha) \\
\forall \alpha \in \mathbb{R}^{+\star}, \forall x > 1, n^\alpha + x^n &= \Theta(x^n) \\
\forall x \in \mathbb{R}^{+\star}, n! + x^n &= \Theta(n!) \\
2^{2^n} + n! &= \Theta(2^{2^n})
\end{aligned}
$$

D.4.5 Bons algorithmes

Un bon algorithme est un algorithme polynomial. Ou plus exactement, un mauvais algorithme est un algorithme non polynomial.

Taille / Complexité	20	50	100	200	500	1000
$10^3 n$	0.02 s	0.05 s	0.1 s	0.2 s	0.5 s	1 s
$10^3 n \log n$	0.09 s	0.3 s	0.6 s	1.5 s	4.5 s	10 s
$100n^2$	0.04 s	0.25 s	1 s	4 s	25 s	2 mn
$10n^3$	0.02 s	1 s	10 s	1 mn	21 mn	27 h
$n^{\log n}$	0.4 s	1.1 h	220 j	12500 ans	5.10^{10} ans	
$2^{\frac{n}{3}}$	10^{-4} s	0.1 s	2.7 h	3.10^6 ans		
2^n	1 s	36 ans				
3^n	58 mn	2.10^{11} ans				
$n!$	77100 ans					

TABLE D.1 – Temps de calcul pour différentes complexités

En effet, l'augmentation de la puissance des machines peut permettre d'étudier des algorithmes polynomiaux sur des tailles de données plus importantes, mais elle n'a pratiquement pas d'influence pour les algorithmes exponentiels.

Ainsi, pour passer d'un problème de taille 50 à un problème de taille 100 en restant à temps constant, il faut multiplier la puissance par 10 pour un problème en n^3, par 4800 pour un problème en $n^{\log n}$, et par 100000 pour un problème en $2^{\frac{n}{3}}$.

Le tableau D.1 présente les temps de calcul d'algorithmes de différentes complexités en fonction de la taille des données. Les cases non remplies ont une valeur supérieure à 1000 milliards d'années.

D.5 Classes de complexité

Machine de Turing

Une machine de Turing est un modèle abstrait du fonctionnement d'un ordinateur et de sa mémoire, créé par Alan Turing en vue de donner une définition précise au concept d'algorithme ou « procédure mécanique ». Ce modèle est toujours largement utilisé en informatique théorique, en particulier pour résoudre les problèmes de complexité algorithmique et de calculabilité. Une machine de Turing est composée des éléments suivants :

— une mémoire ;
— un lecteur ;
— des états internes.

La figure D.1 représente une machine de Turing.

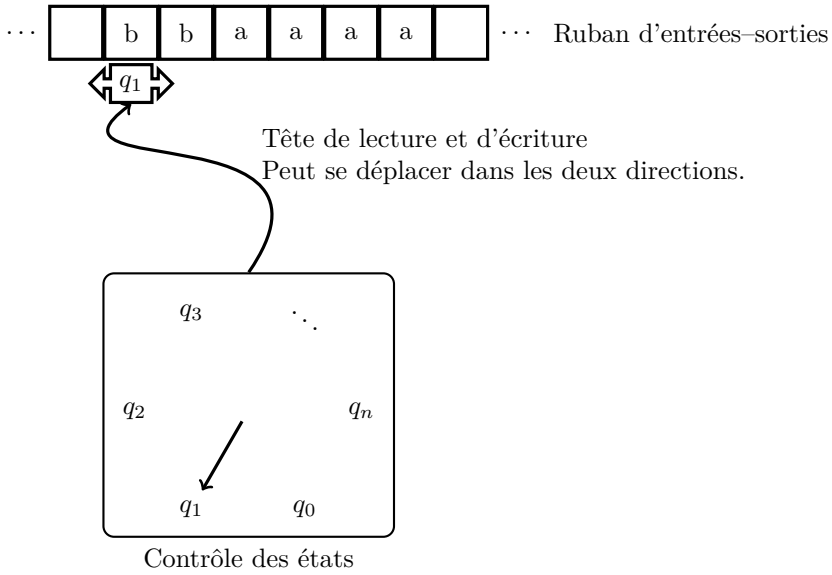


FIGURE D.1 – Machine de Turing

La mémoire : c'est là que s'inscrit l'information et que s'opèrent les altérations et transformations de celle-ci. Elle peut être représentée par un ruban de longueur infinie divisé en cases, chaque case pouvant recevoir une donnée. Les données font partie d'un alphabet $\mathcal{A} = (S_0, \ldots, S_n)$ pour lequel le symbole S_0 (noté $\square$) désigne le symbole blanc, ou l'absence de symbole.

Le lecteur : le ruban défile devant un lecteur qui peut lire une case mémoire à la fois. Le ruban peut se déplacer devant le lecteur d'une case vers la gauche ou d'une case vers la droite.

Les états internes : à chaque instant, le lecteur peut effectuer les actions suivantes sur la mémoire :

1. rien ;
2. effacer le symbole lu ;
3. effacer le symbole lu et écrire un autre symbole à la place ;
4. avancer le ruban d'une case vers la gauche ;
5. avancer le ruban d'une case vers la droite ;

6. s'arrêter.

L'action à effectuer dépend du symbole lu sur le ruban, mais également de l'état du système. On suppose que le système peut prendre un nombre fini d'états noté $\mathcal{Q} = (q_0, \ldots, q_n)$. q_0 représente l'état initial, et q_n l'état final.

Fonctionnement d'une machine de Turing : la machine de Turing est initialisée avec un certain ruban, le lecteur placé sur une certaine case. Elle est initialement dans l'état q_0. Elle commence alors à fonctionner en lisant le symbole du ruban en face du lecteur, en effectuant l'action correspondante et en passant dans un nouvel état.

Exemple de machine de Turing : considérons une machine de Turing permettant de multiplier un nombre par 2. Dans notre exemple, nous comptons en base 1, c'est-à-dire que l'alphabet de la machine contient uniquement le symbole 1.

Le principe de l'algorithme est d'effacer les 1 du nombre initial un par un, et de les remplacer par deux 1 dans le nombre final. Le nombre initial et le nombre final sont séparés par un □.

L'algorithme utilisé est alors :

— on se place sur le premier 1 du nombre à multiplier par 2 ;
— on remplace le 1 par □ ;
— on décale la tête de lecture vers la droite jusqu'à arriver sur □ ;
— on se place à la fin du nombre final ;
— on écrit deux fois 1 ;
— on se replace au début du nombre initial.

La machine traduisant cet algorithme est représentée figure D.2.

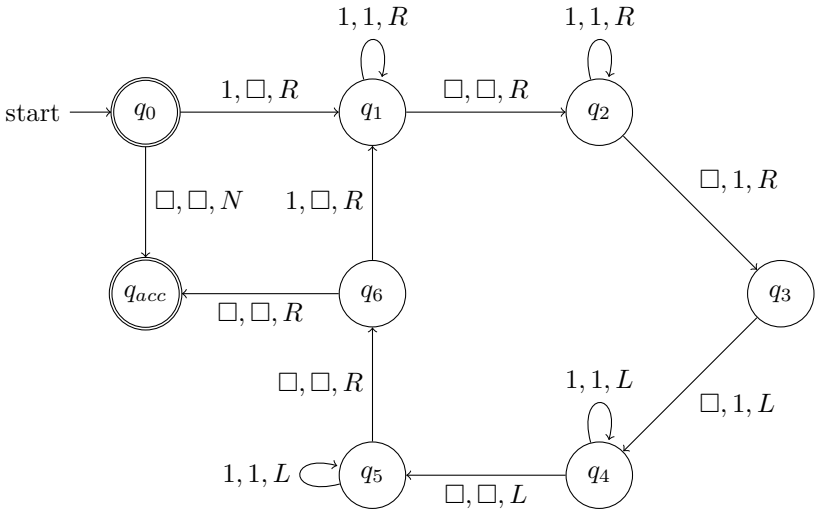


FIGURE D.2 – Machine de Turing calculant $\lambda x[2x]$

Langage Turing-complet : le terme Turing-complet désigne en informatique un système formel ayant au moins le pouvoir d'expression des machines de Turing.

Un langage de programmation est dit Turing-complet s'il permet de représenter toutes les fonctions calculables au sens de Turing (à la finitude de la mémoire des ordinateurs actuels près). La plupart des langages usuels de programmation (C, C++, Java, Python, ...) sont Turing-complets (HTML, SQL ne sont pas Turing-complets).

La thèse de Church-Turing, des noms des mathématiciens Alonzo Church et Alan Turing, affirme que tout traitement réalisable mécaniquement peut être accompli par une machine de Turing. Tout programme d'ordinateur (qui peut être vu comme une suite de traitements réalisables mécaniquement) peut donc être traduit en une machine de Turing.

D'autre part, certaines machines de Turing, dites universelles, peuvent effectuer tous les traitements possibles avec une machine de Turing quelconque. Tous les langages Turing-complets auraient donc, sur un ordinateur disposant d'une mémoire infinie, les possibilités de calcul d'une machine de Turing universelle, de sorte que toutes les machines de Turing peuvent être simulées par un programme écrit dans l'un de ces langages. La thèse de Church-Turing affirme donc que n'importe quel langage de programmation (Turing-complet) permet d'exprimer tous les algorithmes.

Machine déterministe et machine non déterministe

Une machine déterministe est le modèle formel d'une machine telle que nous les connaissons. Nos ordinateurs ainsi que les machines de Turing sont des machines déterministes.

Une machine non déterministe est une machine déterministe (de Turing par exemple) à laquelle on a greffé un *oracle*. L'oracle peut générer une suite quelconque de bits (ou de mots de l'alphabet $\mathcal{A}$) et la machine doit se contenter de vérifier que la suite de bits en question est solution du problème donné.

Par exemple, le problème consistant à dire si un entier est factorisable (c'est-à-dire non premier) se résout sur machine non déterministe avec un algorithme de ce type :

```
demander à l'oracle p > 1, p < x
demander à l'oracle q > 1, q < x
renvoyer vrai si pq=x
```

La machine renvoie vrai s'il existe un p et un q tel que pq=x. Dans la pratique, on peut émuler une machine non déterministe sur machine déterministe. Dans notre exemple, il suffit de remplacer l'oracle par une fonction générant tout les couples (p, q) avec $p, q > 1$ et $p, q < x$. Évidemment un tel calcul prend un temps énorme comparé à une simple multiplication.

Résoudre un problème sur machine non déterministe équivaut donc à savoir vérifier si une solution proposée fonctionne.

Théorie de la complexité

En théorie de la complexité, un problème est représenté par un ensemble de données en entrée, et une question sur ces données (pouvant demander éventuellement un calcul). On ne traite que des problèmes de décision binaire, c'est-à-dire posant une question dont la réponse est oui ou non. Cependant on étend la notion de complexité aux problèmes d'optimisation (consistant à trouver la solution optimale d'un problème). En effet il est facile de transformer un problème d'optimisation en problème de décision. Si par exemple on cherche à optimiser une valeur n on traite le problème de décision qui consiste à comparer n à un certain k. En traitant plusieurs valeurs de k on peut déterminer une valeur optimale. On confondra souvent un problème d'optimisation et son problème de décision associé.

On considère que l'ensemble des instances d'un problème est l'ensemble des données que peut accepter ce problème en entrée, par exemple l'ensemble des permutations de n entiers à trier pour un algorithme de tri.

On distingue les classes de complexité suivantes :

L : un problème de décision qui peut être résolu par un algorithme en espace logarithmique par rapport à la taille de l'instance avec une machine de Turing déterministe.

NL : cette classe correspond à la précédente mais pour une machine de Turing non déterministe.

P : un problème de décision est dans P s'il peut être décidé par un algorithme en un temps polynomial par rapport à la taille de l'instance sur une machine de Turing déterministe. On qualifie alors le problème de polynomial.

NP : c'est la classe des problèmes de décision pour lesquels la réponse oui peut être décidée par un algorithme en un temps polynomial par rapport à la taille de l'instance sur une machine de Turing non déterministe.

Autre définition de la classe NP : la classe NP est formée des problèmes de décision qui possèdent un vérifieur polynomial, c'est-à-dire qu'il est possible de vérifier si une valeur est solution du problème en temps polynomial.

Co-NP : nom parfois donné pour l'équivalent de la classe NP avec la réponse non (*i.e.* le problème peut être résolu par un algorithme qui vérifie qu'une valeur n'est pas solution du problème en temps polynomial).

PSPACE : les problèmes décidables par un algorithme en espace polynomial par rapport à la taille de son instance.

NSPACE ou NPSPACE : les problèmes décidables par un algorithme en espace polynomial par rapport à la taille de son instance sur une machine de Turing non déterministe.

EXPTIME : les problèmes décidables par un algorithme en temps exponentiel par rapport à la taille de son instance.

On a les inclusions: $P \subseteq NP$ et Co-NP $\subseteq PSPACE = NPSPACE$.

Remarque :

— Il existe beaucoup d'autres classes de complexité qui ne seront pas détaillées dans ce cours. Vous pouvez en trouver un grand nombre à l'adresse suivante :
http://qwiki.caltech.edu/wiki/Complexity_Zoo

Problème NP-Complet

Soit C une classe de complexité (comme P, NP, etc.). Un problème est C-dur (ou C-difficile) si ce problème est plus dur que tout problème dans C. Formellement on définit une notion de réduction : soient p et q deux problèmes, p se réduit à q si p est une instance de q. Et donc p est C-dur (ou C-difficile) si pour tout problème q de C, q se réduit à p.

On dit qu'un problème est C-complet si :

— il est dans C ;

— il est C-dur (ou C-difficile).

Les problèmes complets les plus étudiés sont les problèmes NP-complets. Ceci parce que beaucoup de problèmes intéressants sont NP-complets et que l'on ne sait pas résoudre un problème NP-complet efficacement à cause du non déterminisme.

De manière intuitive, dire qu'un problème peut être décidé à l'aide d'un algorithme polynomial sur une machine de Turing non déterministe signifie qu'il est facile, pour une solution donnée, de vérifier en un temps polynomial si celle-ci répond au problème pour une instance donnée mais que le nombre de solutions à tester pour résoudre le problème est exponentiel par rapport à la taille de l'instance. Le non-déterminisme permet de masquer la quantité exponentielle des solutions à tester tout en permettant à l'algorithme de rester polynomial.

Il existe de nombreux problèmes NP-Complets célèbres, dont les problèmes suivants :

— le voyageur de commerce (NP-dur) ;

— recherche de cycle hamiltonien ;

— calcul de la clique maximum (NP-dur) ;

— colorations de graphes (pour au moins trois couleurs) ;

— recherche d'ensemble dominant dans un graphe ;

— recherche de couverture de sommets dans un graphe.

Remarque : P vs NP

On a trivialement $P \subseteq NP$ car un algorithme déterministe est un algorithme non déterministe particulier. Par contre la réciproque : $NP \subseteq P$, que l'on résume généralement à $P = NP$ du fait de la trivialité de l'autre inclusion, est l'un des problèmes ouverts les plus fondamentaux et intéressants en informatique théorique. Cette question a été posée en 1970 pour la première fois et celui qui arrivera à répondre à la question de l'égalité de P et NP recevra le prix Clay (plus de $1.000.000).

D.6 Complexité des algorithmes de tris

Lors de l'étude des différents algorithmes de tris, une question vient naturellement à l'esprit : *quel est le meilleur tri existant ?* Ou encore : existe-t-il une complexité minimale pour les algorithmes de tri ? Par meilleur algorithme de tri, on entend ici algorithme de tri qui minimise le nombre de comparaisons.

Pour simplifier le problème, nous supposerons que toutes les valeurs à trier sont distinctes. Afin de trier notre ensemble, il est nécessaire dans le pire des cas de comparer tous les éléments deux à deux. Si l'ensemble contient n éléments, il existe $n!$ permutations possibles. Comparer les éléments deux à deux revient à les placer dans un arbre binaire (voir figure D.3 : les comparaisons sont les nombres dans les cercles, les ordres finaux possibles sont dans les rectangles). Trier un ensemble revient à parcourir l'arbre binaire jusqu'à une feuille.

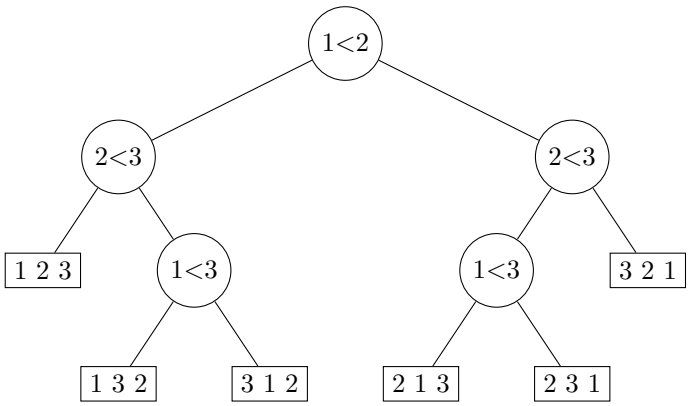


FIGURE D.3 – Comparaisons permettant de trier 3 éléments

Remarquons que cet arbre possède $f = n!$ feuilles (toutes les permutations possibles). De plus, si l'arbre est de hauteur h, alors

$$f \leqslant 2^h \tag{D.1}$$

Dans le pire des cas, l'algorithme ne sera capable de conclure qu'après un nombre de tests égal à la hauteur de l'arbre. Soit $c(n)$ le nombre de comparaisons nécessaire. Étant donné que ce nombre est un entier, on peut réécrire l'équation D.1 en :

$$c(n) \geqslant h = \lceil \log_2(n!) \rceil$$

En utilisant la formule de Stirling, on obtient :

$$\lceil \log_2(n!) \rceil = n \log_2(n) - \frac{n}{\ln(2)} + \frac{1}{2} \log_2(n) + \mathcal{O}(1)$$

C'est-à-dire :

$$c(n) = \Theta(n \log n)$$

Autrement dit : la complexité dans le pire des cas d'un algorithme de tri basé sur des comparaisons ne peut pas être meilleure que $\Theta(n \log n)$.

D.7 Complexité des opérations sur les structures de données classiques

Cette section, inspirée de https://wiki.python.org/moin/TimeComplexity, décrit la complexité en temps des opérations les plus courantes sur les structures de données classiques en Python.

Dans les expressions suivantes, n sera le nombre d'éléments dans la structure, et k sera soit la valeur d'un paramètre ou le nombre d'éléments du paramètre.

D.7.1 Classe *list*

La représentation interne de la liste s'appuie sur un tableau. La conséquence est le coût du redimentionnement d'une liste [1], ou de l'insertion en début de liste. La table D.2 recense le coût de diverses opérations sur une liste `l`.

Opération	Complexité moyenne	Complexité amortie dans le pire des cas
`l.copy()`	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`l.append(x)`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`l.insert(i, x)`	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`l[i]`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`l.remove(x)`	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`l[i:j]`	$\mathcal{O}(k)$	$\mathcal{O}(k)$
`del(l[i:j])`	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`l[i:j] = x`	$\mathcal{O}(n+k)$	$\mathcal{O}(n+k)$
`l.extend(k)`	$\mathcal{O}(k)$	$\mathcal{O}(k)$
`l.sort()`	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
`l * k`	$\mathcal{O}(nk)$	$\mathcal{O}(nk)$
`x in l`	$\mathcal{O}(n)$	

TABLE D.2 – Coût des opérations sur une liste

D.7.2 Classe *collections.deque*

La file [2] est représentée par une liste doublement chaînée, et elle permet un accès rapide à chaque extrémité. Cependant, l'accès aux éléments du milieu est plus long. La table D.3 recense le coût de diverses opérations sur une file `f`.

D.7.3 Classe *dict*

Dans la table D.4, `k` représentera une clé du dictionnaire.

D.7.4 Classe *set*

Le comportement de la classe *set* est similaire à celui de la classe *dict*. Dans la table D.5, `s` et `t` représentent des ensembles.

1. Car il est alors nécessaire de déplacer toutes les valeurs.
2. En anglais *deque*.

Opération	Complexité moyenne	Complexité amortie dans le pire des cas
`f.copy()`	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`f.append(x)`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`f.appendleft(x)`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`f.insert(i, x)`	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`f[i]`	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`f.pop(x)`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`f.popleft(x)`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`f.extend(k)`	$\mathcal{O}(k)$	$\mathcal{O}(k)$
`f.extendleft(k)`	$\mathcal{O}(k)$	$\mathcal{O}(k)$
`f.rotate(k)`	$\mathcal{O}(k)$	$\mathcal{O}(k)$
`f.remove(x)`	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`x in f`	$\mathcal{O}(n)$	

Table D.3 – Coût des opérations sur une file

Opération	Complexité moyenne	Complexité amortie dans le pire des cas
`d.copy()`	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`d[k]`	$\mathcal{O}(1)$	$\mathcal{O}(n)$
`d[k] = x`	$\mathcal{O}(1)$	$\mathcal{O}(n)$
`del(d[k])`	$\mathcal{O}(1)$	$\mathcal{O}(n)$
`x in d`	$\mathcal{O}(1)$	

Table D.4 – Coût des opérations sur un dictionnaire

Opération	Complexité moyenne	Complexité amortie dans le pire des cas
`x in s`	$\mathcal{O}(1)$	$\mathcal{O}(n)$
`s.add(x)`	$\mathcal{O}(1)$	$\mathcal{O}(n)$
`s.remove(x)`	$\mathcal{O}(1)$	$\mathcal{O}(n)$
`s | t`	$\mathcal{O}(len(s) + len(t))$	
`s & t`	$\mathcal{O}(\min(len(s), len(t)))$	
`s - t`	$\mathcal{O}(len(s))$	
`s ^ t`	$\mathcal{O}(len(s))$	

Table D.5 – Coût des opérations sur un ensemble

Par chance, reprit-il, vous êtes précisément venu frapper à la bonne porte avec votre intéressant problème, car le mot « impossible » ne figure pas dans mon dictionnaire. En fait, ajouta-t-il en brandissant le livre malmené, il semble que manque tout ce qui se situe entre « hareng » et « marmelade ».

Douglas Adams

E
ASCII et Unicode

Sommaire

Quels sont les caractères autorisés sur un ordinateur, et comment sont-ils représentés ? Pour répondre à ces questions, plusieurs normes ont été définies. De la nome ascii à l'unicode, voici un bref panorama de l'existant.

E.1 Norme ASCII

La texte suivant est extrait de wikipedia[1], l'encyclopédie libre.

La norme ascii (American Standard Code for Information Interchange), a longtemps été utilisée pour le codage de caractères en informatique. Elle a été inventée par l'américain *Bob Bemer* en 1961. Encore aujourd'hui, la table ascii est grandement utilisée, même si parfois complétée par une table étendue. L'usage a imposé la prononciation [æski ː], ou pour des francophones [aski ː]. C'est également la variante américaine du codage de caractères iso/cei 646.

L'ascii est très répandu dans la mesure où il s'agit du sous ensemble commun à la plupart de jeux de caractères (utf-8, windows-1252, iso-latin-15, iso-latin-1). L'ascii est également autosuffisant pour du texte anglais, langue la plus répandue sur internet.

L'ascii définit 128 caractères, codés en binaire de 0000000 à 1111111. À l'origine, (dans les années 1960) ce code mis au point pour la langue anglaise, ne contenait que 7 bits. Afin de pouvoir coder des caractères accentués ou spécifiques à une langue, le code ascii a été étendu à 8 bits (un octet) dans les années 1970.

Les caractères de 0 à 31 ainsi que le 127 ne sont pas affichables et correspondent à des directives de terminal. Le caractère 32 est l'espace blanc. Les autres correspondent aux chiffres arabes, aux lettres latines majuscules et minuscules et à quelques symboles de ponctuation.

1. http://fr.wikipedia.org

Le tableau E.1 représente les 128 caractères de la table ASCII (il est possible de l'obtenir sous linux par la commande : `man ascii`).

Code en base				Caractère	Signification
10	8	16	2		
0	0	00	0000000	NUL	Null (nul)
1	01	01	0000001	SOH	Start of Header (début d'entête)
2	02	02	0000010	STX	Start of Text (début du texte)
3	03	03	0000011	ETX	End of Text (fin du texte)
4	04	04	0000100	EOT	End of Transmission (fin de transmission)
5	05	05	0000101	ENQ	Enquiry (demande)
6	06	06	0000110	ACK	Acknowledge (accusé de reception)
7	07	07	0000111	BEL	Bell (caractère d'appel)
8	010	08	0001000	BS	Backspace (espacement arrière)
9	011	09	0001001	HT	Horizontal Tab (tabulation horizontale)
10	012	0A	0001010	LF	Line Feed (saut de ligne)
11	013	0B	0001011	VT	Vertical Tab (tabulation verticale)
12	014	0C	0001100	FF	Form Feed (saut de page)
13	015	0D	0001101	CR	Carriage Return (retour chariot)
14	016	0E	0001110	SO	Shift Out (fin d'extension)
15	017	0F	0001111	SI	Shift In (démarrage d'extension)
16	020	10	0010000	DLE	Data Link Escape
17	021	11	0010001	DC1	Device Control 1 à 4
18	022	12	0010010	DC2	
19	023	13	0010011	DC3	
20	024	14	0010100	DC4	
21	025	15	0010101	NAK	Negative Acknowledge (accusé de reception négatif)
22	026	16	0010110	SYN	Synchronous Idle
23	027	17	0010111	ETB	End of Transmission Block (fin du bloc de transmission)
24	030	18	0011000	CAN	Cancel (annulation)
25	031	19	0011001	EM	End of Medium (fin de support)
26	032	1A	0011010	SUB	Substitute (substitution)
27	033	1B	0011011	ESC	Escape (échappement)
28	034	1C	0011100	FS	File Separator (séparateur de fichier)
29	035	1D	0011101	GS	Group Separator (séparateur de groupe)
30	036	1E	0011110	RS	Record Separator (séparateur d'enregistrement)
31	037	1F	0011111	US	Unit Separator (séparateur d'unité)
32	040	20	0100000	SP	Space (espace blanc)
33	041	21	0100001	!	
34	042	22	0100010	"	
35	043	23	0100011	#	
36	044	24	0100100	$	
37	045	25	0100101	%	
38	046	26	0100110	&	
39	047	27	0100111	’	
40	050	28	0101000	(	
41	051	29	0101001	)	

42	052	2A	0101010	*	
43	053	2B	0101011	+	
44	054	2C	0101100	,	
45	055	2D	0101101	-	
46	056	2E	0101110	.	
47	057	2F	0101111	/	
48	060	30	0110000	0	
49	061	31	0110001	1	
50	062	32	0110010	2	
51	063	33	0110011	3	
52	064	34	0110100	4	
53	065	35	0110101	5	
54	066	36	0110110	6	
55	067	37	0110111	7	
56	070	38	0111000	8	
57	071	39	0111001	9	
58	072	3A	0111010	:	
59	073	3B	0111011	;	
60	074	3C	0111100	<	
61	075	3D	0111101	=	
62	076	3E	0111110	>	
63	077	3F	0111111	?	
64	0100	40	1000000	@	
65	0101	41	1000001	A	
66	0102	42	1000010	B	
67	0103	43	1000011	C	
68	0104	44	1000100	D	
69	0105	45	1000101	E	
70	0106	46	1000110	F	
71	0107	47	1000111	G	
72	0110	48	1001000	H	
73	0111	49	1001001	I	
74	0112	4A	1001010	J	
75	0113	4B	1001011	K	
76	0114	4C	1001100	L	
77	0115	4D	1001101	M	
78	0116	4E	1001110	N	
79	0117	4F	1001111	O	
80	0120	50	1010000	P	
81	0121	51	1010001	Q	
82	0122	52	1010010	R	
83	0123	53	1010011	S	
84	0124	54	1010100	T	
85	0125	55	1010101	U	
86	0126	56	1010110	V	
87	0127	57	1010111	W	

88	0130	58	1011000	`X`	
89	0131	59	1011001	`Y`	
90	0132	5A	1011010	`Z`	
91	0133	5B	1011011	`[`	
92	0134	5C	1011100	`\`	
93	0135	5D	1011101	`]`	
94	0136	5E	1011110	`^`	
95	0137	5F	1011111	`_`	
96	0140	60	1100000	`` ` ``	
97	0141	61	1100001	`a`	
98	0142	62	1100010	`b`	
99	0143	63	1100011	`c`	
100	0144	64	1100100	`d`	
101	0145	65	1100101	`e`	
102	0146	66	1100110	`f`	
103	0147	67	1100111	`g`	
104	0150	68	1101000	`h`	
105	0151	69	1101001	`i`	
106	0152	6A	1101010	`j`	
107	0153	6B	1101011	`k`	
108	0154	6C	1101100	`l`	
109	0155	6D	1101101	`m`	
110	0156	6E	1101110	`n`	
111	0157	6F	1101111	`o`	
112	0160	70	1110000	`p`	
113	0161	71	1110001	`q`	
114	0162	72	1110010	`r`	
115	0163	73	1110011	`s`	
116	0164	74	1110100	`t`	
117	0165	75	1110101	`u`	
118	0166	76	1110110	`v`	
119	0167	77	1110111	`w`	
120	0170	78	1111000	`x`	
121	0171	79	1111001	`y`	
122	0172	7A	1111010	`z`	
123	0173	7B	1111011	`{`	
124	0174	7C	1111100	`|`	
125	0175	7D	1111101	`}`	
126	0176	7E	1111110	`~`	
127	0177	7F	1111111	`DEL`	Delete (effacement)

TABLE E.1 – Table ASCII

Il est possible d'accéder à certains caractères non affichables grâce à des sequences d'échappement (*i.e.* des séquences de caractères commençant par « \ »). Quelques séquences d'échappement utilies sont représentées dans le tableau E.2.

Nom	Caractère	Échappement	Valeur décimale
audible alert (bell)	`BEL`	`\a`	7

backspace	BS	\b	8
horizontal tab	HT	\t	9
newline	LF	\n	10
vertical tab	VT	\v	11
formfeed	FF	\f	12
carriage return	CR	\r	13

TABLE E.2 – Séquences d'échappement

E.2 Norme iso8859

Alors que les 96 caractères ASCII imprimables suffisent à l'échange d'informations en anglais courant, la plupart des autres langues qui utilisent l'alphabet latin ont besoin de symboles additionnels non couverts par l'ASCII, tels que ß(allemand), å(suédois et d'autres langues nordiques), Ø(danois, norvégien). ISO 8859 a cherché à remédier à ce problème en utilisant le huitième bit de l'octet, pour donner de la place à 128 caractères supplémentaires (ce bit était jadis utilisé pour le contrôle de l'intégrité des données (bit de parité), ou était inutilisé). Cependant, il fallait plus de caractères qu'on n'en pouvait mettre dans un jeu de caractères 8 bits, aussi plusieurs tables de correspondances ont été développées, en incluant au moins 10 tables pour couvrir uniquement l'écriture latine.

Par exemple, la table ISO 8859-1 (latin-1 ou européen occidental) couvre la plupart des langues européennes occidentales. Cette table est disponible sous linux par la commande : `man iso_8859-1`.

E.3 Format UTF

UTF-8 [2] est un format de codage de caractères défini pour les caractères Unicode (UCS). Chaque caractère est codé sur une suite d'un à quatre octets. UTF-8 a été conçu pour être compatible avec certains logiciels originellement prévus pour traiter des caractères d'un seul octet.

Le numéro de chaque caractère est donné par le standard Unicode. Les caractères de numéro 0 à 127 sont codés sur un octet dont le bit de poids fort est toujours nul. Les caractères de numéro supérieur à 127 sont codés sur plusieurs octets. Dans ce cas, les bits de poids fort du premier octet forment une suite de 1 de longueur égale au nombre d'octets utilisés pour coder le caractère, les octets suivants ayant 10 comme bits de poids fort.

Représentation binaire	Signification
0xxxxxxx	1 octet codant 1 à 7 bits
110xxxxx 10xxxxxx	2 octets codant 8 à 11 bits
1110xxxx 10xxxxxx 10xxxxxx	3 octets codant 12 à 16 bits
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	4 octets codant 17 à 21 bits

Dans toute chaîne de caractères UTF-8, on remarque que :

— tout octet de bit de poids fort nul code un caractère US-ASCII sur un octet ;

— tout octet de bits de poids fort valant 11 est le premier octet d'un caractère codé sur plusieurs octets ;

— tout octet de bits de poids fort valant 10 est à l'intérieur d'un caractère codé sur plusieurs octets.

Avantages

Universalité : ce codage permet de représenter les milliers de caractères d'Unicode.

2. UCS Transformation Format 8 bits

Compatibilité avec us-ascii : un texte en US-ASCII est codé identiquement en UTF-8.

Réutilisabilité : de nombreuses techniques de programmation informatique valables avec les caractères uniformément codés sur un octet le restent avec UTF-8, notamment :
- la manière de repérer la fin d'une chaîne de caractères C, car l'octet 00000000 dans une chaîne de caractère codés en UTF-8 est toujours le caractère nul ;
- la manière de trouver une sous-chaîne est identique.

Fiabilité : il s'agit d'un codage auto-synchronisant (en lisant un seul octet on sait si c'est le premier d'un caractère ou non).
- une séquence décrivant un caractère n'apparaît jamais dans une séquence plus longue décrivant un autre caractère (cas de Shift-JIS) ;
- il n'existe pas de code « d'échappement » changeant l'interprétation de la suite d'une séquence d'octets.

Inconvénients

Un caractère UTF-8 a une taille variable, ce qui rend certaines opérations sur les chaînes de caractères plus compliquées : le calcul du nombre de caractères ; le positionnement à une distance donnée dans un fichier texte et en règle générale toute opération nécessitant l'accès au caractère de position N dans une chaîne.

Les idéogrammes (kanji, par exemple) utilisent 3 octets en UTF-8 contre 2 octets en UTF-16. Les textes chinois, coréens et japonais y occupent donc plus de mémoire.

Un programme mal écrit peut accepter un certain nombre de représentations UTF-8 et les convertir comme un seul et même caractère. Ceci pose un problème de sécurité : en effet l'analyseur syntaxique peut avoir besoin de rejeter une certaine chaîne de caractères. Par exemple une séquence prohibée pourrait être « /../ » codée en ASCII « 2F 2E 2E 2F » (notation hexadécimale) mais le principe du codage UTF-8 permet de le coder aussi avec « 2F C0 AE 2E 2F ». Si l'analyseur syntaxique n'est pas soigneusement écrit pour rejeter aussi cette chaîne, une brèche potentielle de sécurité est ouverte. Cet exemple est tiré d'un cas réel de virus attaquant des serveurs HTTP du Web en 2001.

F

Tables et bibliographie

Liste des Algorithmes

Listings

Index

Symboles

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z

Bibliographie

[Ada82] Douglas Adams. *Guide du routard galactique*. Denoël, 1982. ISBN : 9782207303405 (cf. p. 6).

[CCC12] Alexandre Casamayou-Boucau, Pascal Chauvin et Guillaume Connan. *Programmation en Python pour les mathématiques*. Dunod, 2012 (cf. p. 16).

[Cor+94] Thomas H. Cormen et al. *Introduction à l'algorithmique*. 2e éd. Dunod, 1994 (cf. p. 66, 234).

[DO11] RTCA DO. *178C, Software considerations in airborne systems and equipment certification*. 2011 (cf. p. 180).

[Fre+05] Eric Freeman et al. *Design patterns - Tête la première*. O'reilly, sept. 2005 (cf. p. 120).

[Gam+95] Erich Gamma et al. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, Professional Computing Series, 1995. ISBN : 0-201-63361-2 (cf. p. 120).

[Hun07] J. D. Hunter. "Matplotlib : A 2D graphics environment". In : *Computing In Science & Engineering* 9.3 (2007), p. 90–95 (cf. p. 96).

[Knu68] Donald E. Knuth. *The art of computer programming*. T. 1. Addison-Wesley, 1968 (cf. p. 233, 234).

[Knu69] Donald E. Knuth. *The art of computer programming*. T. 2. Addison-Wesley, 1969 (cf. p. 72).

[Knu73] Donald E. Knuth. *The art of computer programming*. T. 3. Addison-Wesley, 1973 (cf. p. 66, 70, 153).

[Le 12] Vincent Le Goff. *Apprenez à programmer en Python*. OpenClassrooms, 2012. URL : http://openclassrooms.com/courses/apprenez-a-programmer-en-python (cf. p. 16).

[Sum12] Mark Summerfield. *Rapid GUI Programming with Python and Qt*. Prentice Hall, 2012 (cf. p. 207).

[Swi12] Gérard Swinnen. *Apprendre à programmer avec Python 3*. Éditions Eyrolles, 2012. URL : http://inforef.be/swi/python.htm (cf. p. 16).

[VT18] Guido Van Rossum et Talin. *PEP 3119 – Introducing Abstract Base Classes*. 2007-04-18. URL : https://www.python.org/dev/peps/pep-3119/ (cf. p. 186).

[VWC05] Guido Van Rossum, Barry Warsaw et Nick Coghlan. *PEP 0008 – Style Guide for Python Code*. 2001-07-05. URL : https://www.python.org/dev/peps/pep-0008/ (cf. p. 217).