Algorithmique et Programmation 1

2018-2019

Département d'informatique
Première année de licence
Semestre 1

Table des matières

0 Introduction **9**

1 Logique **11**
1.1 Introduction . . . 11
1.2 Variables booléennes . . . 11
1.3 Fonctions et opérations logiques . . . 11
1.3.1 *Fonctions* logiques . . . 11
1.3.2 *Opérations* logiques . . . 12
1.4 *Opérations* logiques usuelles . . . 12
1.4.1 L'opération *ET* logique . . . 12
1.4.2 L'opération *OU* logique . . . 12
1.4.3 L'opération *NON* logique . . . 13
1.4.4 Fonctions logiques *NON ET* et *NON OU* . . . 13
1.5 Tables de vérité . . . 14
1.6 Pour aller plus loin . . . 14
1.6.1 L'Opération *Equivalence* logique . . . 14
1.7 Théorèmes et Propriétés . . . 15
1.7.1 Distributivité . . . 15
1.7.2 Théorème de Morgan . . . 16
1.7.3 Commutativité . . . 16
1.8 Logique et sens commun . . . 16
1.9 L'exemple d'un circuit électrique . . . 17

2 Types de base, Variables, Répétition **19**
2.1 Le langage *python* . . . 19
2.1.1 Les commentaires . . . 19
2.1.2 Données et expressions . . . 19
2.2 Les nombres : entiers et flottants . . . 20
2.2.1 Définitions et exemples . . . 20
2.2.2 Opérateurs . . . 20
2.2.3 L'ordre d'évaluation des opérateurs . . . 21
2.3 Les chaînes de caractères . . . 22
2.3.1 Définition . . . 22
2.3.2 La *concaténation* . . . 23
2.3.3 La *duplication* . . . 23

2.3.4 `len` : la longueur d'une chaîne . . . 23
2.3.5 Les séquences d'*échappement* . . . 23
2.4 nombres vs chaînes de caractères . . . 24
2.4.1 Les opérateurs + et * . . . 24
2.4.2 Les nombres et les textes-nombres . . . 25
2.4.3 Conversions explicites . . . 26
2.5 Les variables . . . 27
2.5.1 Définitions . . . 27
2.5.2 Dissymétrie . . . 27
2.5.3 Opérateurs . . . 28
2.5.4 Variables et chaînes de caractères . . . 28
2.5.5 Pourquoi utiliser des variables ? . . . 29
2.6 Les répétitions : la boucle `for` . . . 30
2.6.1 Syntaxe . . . 30
2.6.2 Le *compteur* de la boucle . . . 31
2.6.3 Boucles `for` imbriquées . . . 31

3 Booléens et conditionnelles **33**
3.1 Le type booléen . . . 33
3.2 Comparaisons d'expressions . . . 33
3.2.1 Opérateurs de comparaison . . . 33
3.2.2 Comparaison entre nombres de types différents . . . 35
3.2.3 Comparaison entre chaînes de caractères et Unicode . . . 35
3.3 Structure conditionnelle . . . 37
3.3.1 Condition *si* . . . 37
3.3.2 Condition *si ... sinon* . . . 38
3.3.3 Condition *si ... sinon si ... sinon* . . . 38
3.4 Boucle while . . . 40
3.5 Combinaison de propositions par opérateurs logiques . . . 41
3.6 Résumé . . . 43

4 Listes et Tuples **45**
4.1 Listes . . . 45
4.1.1 Qu'est ce qu'une liste ? . . . 45
4.1.2 Opérations sur les listes . . . 46
4.2 Tuples . . . 49
4.2.1 Qu'est-ce qu'un tuple ? . . . 49
4.2.2 Opérations sur les tuples . . . 50
4.3 Conversions entre chaines, listes, tuples . . . 51
4.4 Pour aller plus loin . . . 51
4.5 Exemples simples . . . 52
4.5.1 Parcours de liste . . . 52
4.5.2 Somme d'une liste de nombres . . . 53
4.5.3 Traitement d'une liste de tuples . . . 53

5 Structurer les grands programmes **55**
5.1 Un projet en plusieurs morceaux . . . 55
5.1.1 Exemple de projet . . . 55
5.1.2 Le calcul de e^x . . . 56
5.1.3 Comment assembler les programmes ? . . . 56
5.2 Problèmes soulevés par l'assemblage brut de programmes . . . 56
5.2.1 Lisibilité . . . 56
5.2.2 La redondance . . . 58
5.2.3 L'interférence entre les programmes . . . 58
5.2.4 Testabilité . . . 58
5.3 Solution : isoler/nommer/centraliser les morceaux de programme . . . 59
5.3.1 Isoler . . . 59
5.3.2 Nommer . . . 59
5.3.3 Centraliser . . . 59
5.4 Conséquences . . . 59
5.4.1 La *modularité* . . . 60
5.5 Exemple : les ensembles de *Mandelbrot* . . . 60
5.5.1 Définition . . . 61
5.5.2 Décomposition en blocs . . . 61

6 Fonctions et modules **63**
6.1 Vocabulaire : argument, valeur de retour, appel . . . 63
6.1.1 Les fonctions en mathématique . . . 63
6.1.2 Les fonctions en informatique . . . 63
6.2 Différentes sortes de fonctions en python . . . 64
6.2.1 Le nombre d'arguments . . . 64
6.2.2 La valeur de retour . . . 64
6.3 La définition de fonctions . . . 65
6.3.1 Syntaxe générale . . . 65
6.3.2 Vocabulaire : *en-tête, corps* et *paramètres* . . . 65
6.3.3 Quelques exemples . . . 66
6.3.4 L'ordre et le type des arguments d'une fonction . . . 66
6.3.5 Au sujet de l'instruction `return` . . . 67
6.3.6 Fonctions et expressions . . . 67
6.3.7 La portée des variables (locales ou globales) . . . 68
6.3.8 Documentation d'une fonction . . . 69
6.4 Les modules . . . 70
6.4.1 Qu'est-ce qu'un module ? . . . 70
6.4.2 Utiliser un module . . . 70
6.4.3 Utiliser plusieurs modules . . . 70
6.4.4 N'utiliser que partiellement un module . . . 71
6.4.5 Contenu d'un module . . . 71
6.4.6 Créer un module . . . 72
6.4.7 Docstrings . . . 72
6.4.8 Tester son module directement dans... le module . . . 73
6.5 Les Packages : qu'est-ce qu'un package . . . 74
6.6 Pour aller plus loin . . . 75

Appendices **76**
6.A Les 72 fonctions prédéfinies dans *Python3* 76
6.B Pour aller plus loin : astuces en lien avec `input` et `print` 76
6.B.1 Saisie d'informations numériques 76
6.B.2 Affichage d'informations numériques 77

7 Méthodologie de test et de debug **80**

8 Récursivité **81**
8.1 Principe de la récursivité 81
8.1.1 Exemple (récursif) 81
8.1.2 Exemple (itératif) 81
8.1.3 Récursif ou itératif ? 82
8.1.4 Fonctionnement général 82
8.2 Fonctions récursives en Python 82
8.2.1 Structure 82
8.2.2 Exemple (suite) 83
8.2.3 Déroulement 84
8.3 Objet de la récursion 85
8.3.1 Récursivité portant sur des nombres entiers 85
8.3.2 Exemple : la factorielle 86
8.3.3 Récursivité portant sur des listes 87
8.3.4 Définition mathématique et fonction récursive 87
8.4 Quelques exemples plus complexes 88
8.4.1 Récursivité multiple 88
8.4.2 Exemple 88
8.4.3 Fonction *wrapper* 89
8.4.4 Valeurs par défaut 90
8.4.5 Récursivité croisée 90
8.5 Exercices 91

9 Algorithmes et complexité **93**
9.1 Qu'est-ce qu'un algorithme ? 93
9.1.1 Faire abstraction des langages de programmation et du matériel 93
9.1.2 Définition 94
9.1.3 Les *tours de Hanoï* 94
9.1.4 L'équation du second ordre 95
9.2 Algorithmes récursifs et itératifs 95
9.2.1 Définition 95
9.2.2 Exemples 96
9.3 Généralité, Convergence 97
9.3.1 Trouver la solution de l'équation $f(x) = 0$ 97
9.4 Complexité 98
9.4.1 définition 98
9.4.2 Classes de complexité 98
9.4.3 Autres exemples 102

10 Les algorithmes de tri **103**

Chapitre 0

Introduction

Le projet *labyrinthe*

Pendant ce semestre, notre objectif sera d'écrire un programme permettant de construire, de visualiser un labyrinthe, puis de trouver un chemin entre deux points de ce labyrinthe. Chaque chapitre du cours permettra d'aller de l'avant dans ce projet.

L'algorithmique

L'objectif sous-jacent de ce cours est l'apprentissage de l'*algorithmique*, c'est à dire l'acquisition de compétences nécessaires à tous les programmeurs quelque soit le langage qu'ils utilisent. Pour mettre en œuvre ces compétences, nous utiliserons le langage *python*.

L'objectif de ce cours n'est donc pas l'apprentissage des spécificités du langage *python*. En particulier, nous n'étudierons ni les classes ni les dictionnaires ni les *comprehensive lists*. Dans certains cas, lorsque cela est explicitement spécifié, celles et ceux d'entre vous qui connaissent déjà ces éléments pourront les utiliser dans leur programme.

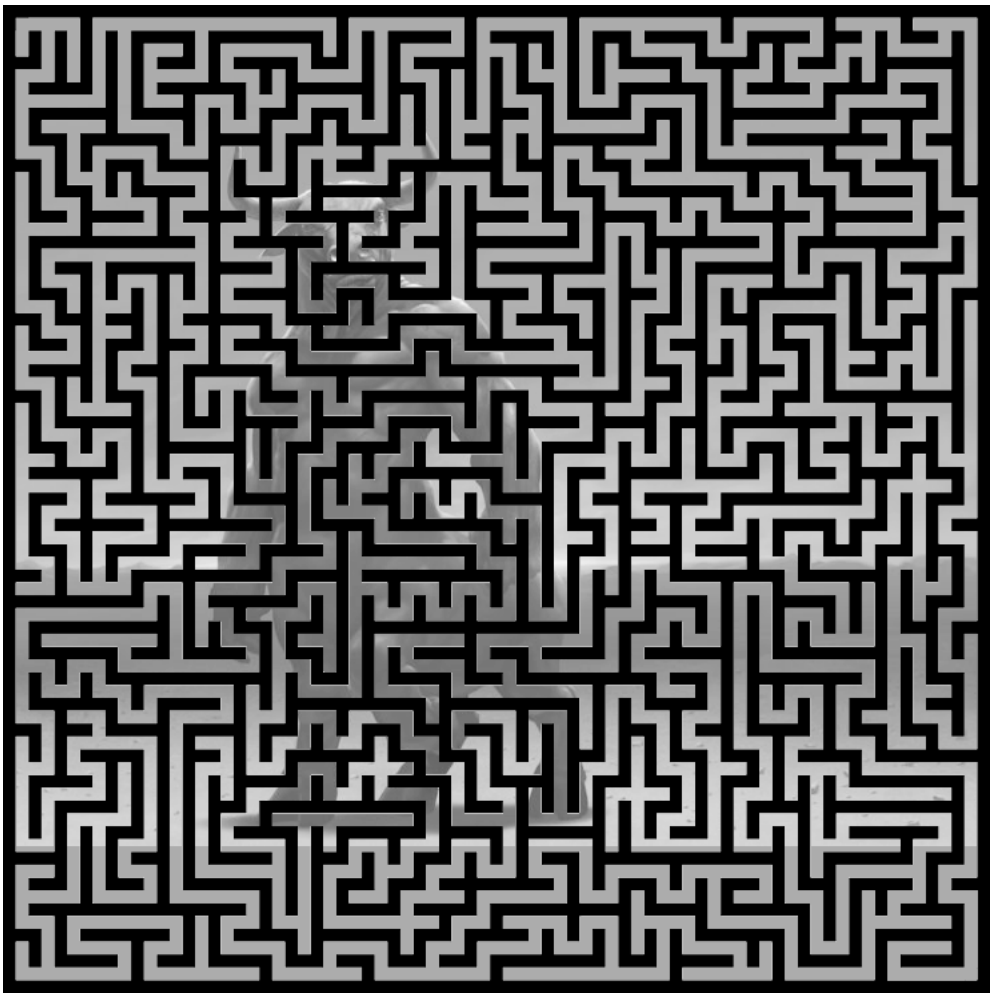
L'articulation cours/TD/TP

Nous travaillerons en salle machine et en salle de cours. En salle machine, nous ferons des travaux qui seront liés de façon directe ou indirecte au projet *labyrinthe*. En salle de cours, nous ferons peu de cours. D'une séance de cours à la séance suivante, vos enseignants vous demanderont de lire une partie du présent document. À la séance suivante, vous pourrez poser toutes les questions qui auraient été soulevées à la lecture du cours. Après la séance de questions, vous pourrez être évalués sur cette lecture. Le reste de la séance sera consacrée aux exercices.

Les évaluations

Vous serez évalués de trois façons différentes sur les compétences que vous aurez acquises.

1. Comme évoqué plus haut, vous serez évalués pendant les séances de cours sur la lecture et l'assimilation du cours ;
2. Le vendredi 23 novembre vous passerez une épreuve écrite portant sur vos compétences en algorithmique
3. Le mardi 11 décembre vous passerez une épreuve pratique portant sur le projet labyrinthe. Ainsi, ce projet ne sera pas évalué en lui-même. Mais, vous vous en rendrez compte, si vous avez réalisé le projet, l'épreuve pratique vous paraîtra très facile.



Chapitre 1

Logique

1.1 Introduction

George Boole, mathématicien anglais (1815-1864), est le créateur de la logique moderne. Il a posé les fondements de l'algèbre Booléenne. Aujourd'hui, l'algèbre de Boole est indispensable en informatique et dans la conception des circuits électroniques.

Les mots ET, OU, NON, SI, associés au langage courant, ont un sens précis qui nous permet d'étayer notre raisonnement. En informatique, on utilise ce sens pour réaliser des opérations logiques qui sont à la base de tous les calculs effectués par un ordinateur.

1.2 Variables booléennes

Dans toute la suite de ce cours les qualificatifs *logique* et *booléen* seront utilisés de façon équivalente et interchangeable.

Une variable *booléenne* est une variable qui ne peut prendre que deux valeurs. On admet plusieurs dénominations pour ces deux valeurs :

- 1 ou 0 ;
- Vrai ou Faux ;
- True ou False.

Dans un premier temps, je vous invite à faire abstraction de la signification de *Vrai* et de *Faux* dans le langage naturel. Il s'agit simplement de deux valeurs différentes. Dans la suite de ce cours, nous choisirons la dénomination 1 et 0. La correspondance logique est : 1 pour Vrai et 0 pour Faux.

1.3 Fonctions et opérations logiques

1.3.1 *Fonctions* logiques

Une *fonction logique* est une fonction dont l'ensemble de départ est $\{0,1\}^n$ et dont l'ensemble d'arrivée est $\{0,1\}^m$ avec $n > 0$ et $m > 0$.

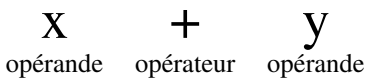
Autrement dit, une fonction logique admet n variables logiques et associe à ces variables m valeurs logiques.

Les fonctions logiques sont très importantes en informatique car, dans un ordinateur, les données sont représentées par des bits valant soit 0 soit 1. Donc tout traitement de données informatique s'apparente à une fonction logique.

1.3.2 *Opérations* logiques

Pour exprimer les fonctions logiques, on peut utiliser des combinaisons d'*opérations logiques* simples. Dans ce cours, nous verrons les opérations logiques *unaires* et *binaires*.

Les opérations binaires mettent en jeu deux *opérandes* entourant un symbole appelé l'*opérateur* (comme ci-contre). Les opérations unaires mettent en jeu une seule opérande et un opérateur. Dans les deux cas, les opérations logiques produisent une valeur logique.



1.4 *Opérations* logiques usuelles

Les principales opérations logiques sont les fonctions *ET*, *OU* et *NON*. À elles trois elles permettent de décrire toutes les fonctions logiques.

1.4.1 L'opération *ET* logique

Quelques soient les valeurs logiques *x* et *y*, l'opération $x \cdot y$ se lit *"x et y"*.
$x \cdot y = 1$ si les opérandes *x* et *y* valent 1 tous les deux ;
$x \cdot y = 0$ dans tous les autres cas.

Concrètement :
$0 \cdot 0 = 0$
$0 \cdot 1 = 0$
$1 \cdot 0 = 0$
$1 \cdot 1 = 1$

1.4.2 L'opération *OU* logique

Quelques soient les valeurs logiques *x* et *y*, l'opération $x + y$ se lit *"x ou y"*.
$x + y = 1$ si au moins l'une des opérandes vaut 1 ;
$x + y = 0$ dans tous les autres cas.

Concrètement :
$0 + 0 = 0$
$0 + 1 = 1$

$1 + 0 = 1$
$1 + 1 = 1$

Il est important de ne pas pousser trop loin l'analogie entre les notations des opérateurs logiques $\cdot$ et $+$ d'une part et la multiplication $*$ et l'addition + d'autre part. Ce sont des choses différentes, en effet, en logique *1 + 1 = 1* (*1 OU 1 = 1*) alors qu'en algèbre classique : *1 + 1 = 2*. Il est donc conseillé de toujours bien lire « 1 ou 1 » en logique.

1.4.3 L'opération *NON* logique

Quelque soit la valeur logique *x*, l'opération $\overline{x}$ se lit *"non x"*.
$\overline{x} = 1$ si $x = 0$
$\overline{x} = 0$ si $x = 1$

Il s'agit d'une opération unaire. Nous ne verrons pas d'autres opérations unaires dans ce chapitre. Concrètement :
$\overline{0} = 1$
$\overline{1} = 0$

1.4.4 Fonctions logiques *NON ET* et *NON OU*

Les opérateurs *NON ET* et *NON OU* (en anglais *NAND* et *NOR*) sont les compléments des opérations *ET* et *OU*.

Quelques soient les valeurs logiques *x* et *y* :
$NONET(x, y) = \overline{x \cdot y}$
$NONOU(x, y) = \overline{x + y}$

Ces fonctions jouent un rôle important, car on montre que toutes les fonctions logiques peuvent être obtenues par des combinaisons de fonctions $NAND$. (Même chose pour la fonction NOR.)

C'est d'autant plus intéressant que les opérations $NAND$ et NOR peuvent être réalisées par un circuit électronique très simple contenant seulement deux transistors. Grâce à cela, il est possible de réaliser n'importe quelle fonction logique avec un circuit électronique.

1.4.4.1 Opération *OU exclusif*

Quelques soient les valeurs logiques *x* et *y* :
$x \oplus y = 1$ si une et une seule des deux variables *x* et *y* vaut *1* (et l'autre 0) ;
$x \oplus y = 0$ si *x* et *y* valent tous les deux 0 ou tous les deux 1.

Concrètement :
0 $\oplus$ *0 = 0*
0 $\oplus$ *1 = 1*
1 $\oplus$ *0 = 1*

1 ⊕ 1 = 0

L'opération *OU* vue à la section 1.4.2 est souvent appelée l'opération *OU inclusif* par opposition à l'opération *OU exclusif* notée aussi XOR (de l'anglais « eXclusive OR »).

1.5 Tables de vérité

Une table de vérité est une table mathématique utilisée en logique pour représenter la valeur d'une fonction relativement à ses arguments. Elle est composée d'une colonne pour chaque variable (*x* et *y*) et d'une colonne où sont inscrits les résultats de l'opération logique représentée.

> La table de vérité d'une fonction logique définit entièrement l'action de cette fonction. Plus spécifiquement, si deux fonctions logiques ont la même table de vérité, alors ces deux fonctions sont identiques.

On appelle également les tables de vérité « tables de Karnaugh »(du nom de leur inventeur en 1953). Les tables de vérité des fonctions logiques sont données ci-dessous :

Opérateur *ET*		
x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

Opérateur *NON*	
x	$\overline{x}$
0	1
1	0

Opérateur *NON ET*		
x	y	$\overline{x \cdot y}$
0	0	1
0	1	1
1	0	1
1	1	0

Opérateur *OU*		
x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

Opérateur *NON OU*		
x	y	$\overline{x + y}$
0	0	1
0	1	0
1	0	0
1	1	0

Opérateur *OU exclusif*		
x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

1.6 Pour aller plus loin

Les opérations logiques ci-dessus font partie de la culture générale informatique. Je vous demande de les connaître par cœur. Les opérations suivantes peuvent également se révéler intéressantes. Mais si vous deviez les traiter dans le cadre d'un exercice, on vous en rappelerait la définition ou la table de vérité.

1.6.1 L'Opération *Equivalence* logique

La fonction « Equivalence » se note EQV (ou XNOR pour *NON OU exclusif*). On l'appelle également *ET inclusif*. L'opérateur associé à l'opération *équivalence logique* est le $(\odot)$.

Quelques soient les valeurs logiques *x* et *y*,

$x \odot y = 1$ si et seulement si les deux opérandes ont même valeur.

$x \odot y = 0$ si une et une seule des deux opérandes vaut 1 (et l'autre 0).

Concrètement :

0 ⊙ 0 = 1
0 ⊙ 1 = 0
1 ⊙ 0 = 0
1 ⊙ 1 = 1

1.6.1.1 L'opération *Implication*

L'opérateur associé à l'opération « Implication » est $\Rightarrow$.

$x \Rightarrow y$ se lit *"x implique y"*. On peut également la lire « "si x alors y " ». Cela peut prêter à confusion mais l'implication ne rend pas compte d'un lien de causalité, qui indiquerait comment la vérité de y découle de celle de x.

Quelques soient les valeurs logiques *x* et *y* :

$x \Rightarrow y$ vaut 0 si et seulement si *x* vaut 1 et *y* vaut 0.

$x \Rightarrow y$ vaut 1 dans tous les autres cas.

Concrètement :

$0 \Rightarrow 0 = 1$
$0 \Rightarrow 1 = 1$
$1 \Rightarrow 0 = 0$
$1 \Rightarrow 1 = 1$

1.6.1.2 L'opérateur *Inhibition*

L'inhibition est le complément de la fonction implication. L'opérateur associé est $\Rrightarrow$.

Quelques soient les valeurs logiques *x* et *y* :

$x \Rrightarrow y = 1$ si et seulement si x vaut 1 et y vaut 0.

$x \Rrightarrow y = 0$ dans tous les autres cas.

Concrètement :

$0 \Rrightarrow 0 = 0$
$0 \Rrightarrow 1 = 0$
$1 \Rrightarrow 0 = 1$
$1 \Rrightarrow 1 = 0$

Voici les tables de vérité des trois opérateurs précédents.

Opérateur *Equivalence*		
x	y	$x \odot y$
0	0	1
0	1	0
1	0	0
1	1	1

Opérateur *Implication*		
x	y	$x \Rightarrow y$
0	0	1
0	1	1
1	0	0
1	1	1

Opérateur *Inhibition*		
x	y	$x \Rrightarrow y$
0	0	0
0	1	0
1	0	1
1	1	0

1.7 Théorèmes et Propriétés

1.7.1 Distributivité

Comme avec les opérations mathématiques habituelles, il est possible de distribuer, cependant il faut faire attention, la distributivité en logique est différente de la distributivité avec les

opérateurs d'addition et de multiplication :

$$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$$

$$x + (y \cdot z) = (x + y) \cdot (x + z)$$

1.7.2 Théorème de Morgan

$$\overline{x \cdot y} = \overline{x} + \overline{y}$$

$$\overline{x + y} = \overline{x} \cdot \overline{y}$$

1.7.3 Commutativité

Les opérateurs *OU*, *ET* et *Equivalence* sont commutatifs :

$$x + y = y + x$$

$$x \cdot y = y \cdot x$$

$$x \odot y = y \odot x$$

Par contre les opérateurs *Implication* et *Inhibition* ne sont pas commutatifs :

$$x \Rightarrow y \neq y \Rightarrow x$$

$$x/y \neq y/x$$

1.8 Logique et sens commun

Jusqu'ici, nous avons étudié la logique de façon purement mathématique sans lien immédiat avec le langage courant et sans lien avec ce qu'on appelle *un raisonnement logique*. Le seul lien était donné par le fait que le 0 et le 1 logiques peuvent aussi être appelés *Vrai* et *Faux* et qu'en langage naturel, ces mots ont bien un sens.

Ainsi une proposition en langage naturel peut représenter une variable logique. Par exemple la proposition `Il fait beau` peut avoir deux valeurs : *Vrai* ou *Faux* selon qu'il fait effectivement beau ou non.

L'opération *ET* logique

Je peux écrire :

Je me promène	si et seulement si	il fait beau	ET	que j'ai fait mes devoirs
S	=	A	.	B

Cette phrase comporte trois propositions logiques correspondant à trois variables logiques :

- A la variable "Il fait beau"
- B la variable "J'ai fini mes devoirs"
- S la variable "Je me promène"

$S = A \cdot B$ traduit l'idée que si une des deux variables vaut *Faux*, *S* vaut *Faux* donc je ne vais pas me promener. Par contre si *A* est vraie et si *B* est vraie alors *S* est vraie et ... je vais me promener !

Et voici une illustration du théorême de DeMorgan. Si on applique ce théorème, on obtient : $\overline{S} = \overline{A} + \overline{B}$, on traduit alors cette relation en sens commun : « Si je ne me promène pas : soit il ne fait pas beau, soit je n'ai pas fini mes devoirs. »

L'opération *OU* logique

De même que précédemment, je peux écrire :

Je vais à la cafet'	si et seulement si	j'ai faim	OU	que j'accompagne un copain
S	=	A	+	B

- A est la variable logique "J'ai faim" ;
- B est la variable logique "J'accompagne un copain" ;
- S est la variable logique "Je vais à la cafet' ".

$S = A + B$ traduit l'idée que si une des deux variables vaut *Vrai*, *S* vaut *Vrai* donc je vais à la cafet' . Par contre si *A* est faux et si *B* est faux alors *S* est faux et ... je ne vais pas à la cafet'.

Si on applique le théorème de Morgan, $\overline{S} = \overline{A} \cdot \overline{B}$, on traduit alors cette relation en sens commun : « Si je ne vais pas à la cafet' c'est que je n'ai pas faim et que je n'y accompagne pas un copain. »

1.9 L'exemple d'un circuit électrique

Il est également possible d'illustrer la logique booléenne avec des analogies électriques. On considère qu'un interrupteur est :

- à l'état 1 lorsque l'interrupteur est fermé ;
- à l'état 0 lorsque l'interrupteur est ouvert.

Dans le circuit des figures suivantes on représente l'état de la lampe par la variable *S*.

L'opération *ET* logique

Si les variables *A* et *B*, qui schématisent les états des interrupteurs, sont à l'état 1 alors le courant peut passer et la lampe brille. Il suffit qu'un des interrupteurs soit ouvert pour que le courant ne passe pas et que la lampe ne brille pas. On peut donc représenter le fonctionnement de ce circuit par une opération logique *ET* : $S = A \cdot B$.

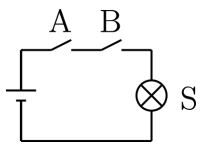


FIGURE 1.1 – Interrupteurs en série

Si on applique le théorème de Morgan, $\overline{S} = \overline{A} + \overline{B}$, on traduit alors cette relation par : « La lampe ne brille pas si l'interrupteur *A* ou si l'interrupteur *B* ne sont pas fermés. »

L'opération *OU* logique

Dans la figure ci-contre il suffit qu'un des deux interrupteurs soit fermé pour que le courant électrique puisse passer et que la lampe brille. Il faut que les deux interrupteurs soient ouverts pour que le courant ne passe pas et que la lampe ne brille pas. On peut donc représenter le fonctionnement de ce circuit par une opération logique *OU* : $S = A + B$.

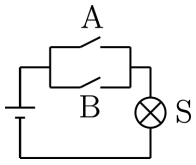


FIGURE 1.2 – Interrupteurs en dérivation

Si on applique le théorème de Morgan, $\overline{S} = \overline{A} \cdot \overline{B}$, on traduit alors cette relation par : « La lampe ne brille pas si l'interrupteur *A* et l'interrupteur *B* ne sont pas fermés. »

Chapitre 2

Types de base, Variables, Répétition

2.1 Le langage *python*

Pendant le premier TP, vous avez déjà pu faire connaissance avec le langage *python*. Vous avez même pu tester le mode script. Dans toute la suite de ce cours, nous utiliseront le mode script.

Dans ce chapitre, nous allons rapidement valider et généraliser les résultats expérimentales que vous avez obtenus au TP1.

2.1.1 Les commentaires

L'interpréteur python ignore tout ce qui se trouve entre un caractère # et un retour à la ligne. Cette partie ignorée est appelée *commentaire*. Les programmeurs utilisent cette possibilité pour rendre leur programme plus lisible. Par exemple :

```
print(5+3)
# cette instruction doit afficher 8
```

2.1.2 Données et expressions

Les données sur lesquelles agit *Python* peuvent être de plusieurs types : *entiers, flottants, booléens, chaînes de caractère, listes, dictionnaires, fonctions, instances de classes* etc. Dans ce chapitre, nous allons nous intéresser aux *entiers*, aux *flottants* et aux *chaînes de caractères*.

Nous nous intéresserons aussi aux *opérateurs* (par exemple les opérateurs d'addition, de multiplication etc) agissant sur ces données.
Tout objet qui possède une valeur est appelé une *expression*. Il peut s'agir d'une donnée d'un des types de base ou une combinaison de données au moyen d'opérateurs. Voici des exemples d'expressions :

```
1
23.5
'coucou'
1 + 3 * 5
23.5 - 5
"numero" + "2"
```

2.2 Les nombres : entiers et flottants

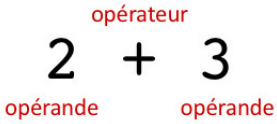
2.2.1 Définitions et exemples

Tous les entiers (en anglais *integer*) au sens mathématique peuvent être représentés en *python*. Les *flottants* (en anglais *float*) représentent les nombres non-entiers. Ces nombres comportent un point séparant la *partie entière* et la *partie fractionnelle*.

Par exemple `1.5` est un flottant. Sa partie entière est 1 et sa partie fractionnelle est le 5. C'est le point qui permet à *python* de reconnaître un flottant. Ainsi `1.0` et `1.` sont des flottants, même si leur partie fractionnelle est nulle.

2.2.2 Opérateurs

Une *opération* est constituée d'un *opérateur* et d'un ou plusieurs *opérandes*.



Les opérateurs suivants opèrent aussi bien sur les opérandes entières que flottantes. Les opérateurs dits *unaires* admettent une seule opérande, alors que les opérateurs dits *binaires* en admettent deux.

Opérateur	**Opération**
**	puissance
+ (unaire)	nombre positif
- (unaire)	changement de signe
*	multiplication
/	division réelle (sans reste)
//	quotient de la division euclidienne
%	*modulo* ou reste de la division euclidienne
+ (binaire)	addition
- (binaire)	soustraction

Les opérateurs de division et de module méritent peut-être quelques explications :

La division dite *réelle* : La division de a par b produit un nombre q (entier ou non) tel que $a = b.q$;

La division dite *euclidienne* : La division de a par b produit un *quotient* q (entier) et un *reste* r tels que $a = b.q + r$ avec $r < b$.

En pratique :

```
>>> 17 / 7
2.4585714
>>> 17 // 7
2
>>> 17 % 7
3
>>> 17.3 // 2.5
6.0
>>> 17.3 % 2.5
2.3
```

La division euclidienne de 17 par 7 donne un quotient de 2 et un reste de 3 car 17 = 7*2 + 3. En mathématique, la division euclidienne concerne les entiers. En *python*, l'opérateur `//` permet aussi de diviser des nombres flottants. Les règles sont les mêmes. Le quotient doit toujours avoir une valeur entière. Le reste peut avoir une valeur entière ou non. `17.3 = 6 * 2.5 + 2.3` avec `2.3 < 2.5` ce qui explique les deux derniers résultats.

2.2.3 L'ordre d'évaluation des opérateurs

Dans une expression il peut y avoir plusieurs opérateurs et, dans certains cas, l'ordre dans lequel on effectue ces opérations peut influer sur le résultat du calcul.

Les parenthèses

En mathématique, comme en informatique, les parenthèses nous permettent de spécifier explicitement dans quel ordre nous souhaitons que les opérations soient effectuées. La règle est de :

— commencer par les parenthèses les plus profondes ;
— évaluer la valeur de l'expression dans ces parenthèses ;
— remplacer les parenthèses et leur contenu par cette valeur.

et de répéter ainsi jusqu'à ce qu'il n'y ait plus de parenthèses. Par exemple la figure 2.1 montre comment le parenthésage détermine l'ordre des opérations dans une expression.

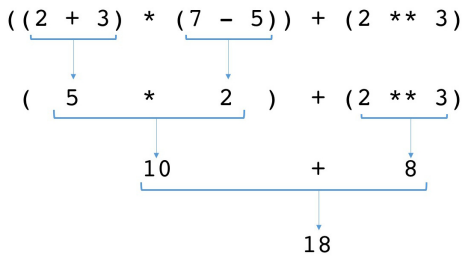


FIGURE 2.1 – Dans l'expressions tout en haut, les parenthèses les plus profondes sont `(2+3)` et `(7-5)`. On les évalue et on les remplace par cette valeur. Puis (deuxième ligne), les parenthèses les plus profondes sont celles qui restent : `(5*2)` et `(2**3)`. Ainsi, cette expression vaut 18.

La notion de *priorité*

Certaines expressions ne sont pas assez parenthésées pour déterminer complètement l'ordre des opérations.

Pour trancher, le langage *python* a défini des règles de priorité. Chaque opérateur est associé à un *niveau* de priorité. Dans le tableau ci-contre, les opérateurs les plus hauts ont les priorités les plus élevées.

`()`
`**`
`+` et `-` (unaires)
`*`, `/`, `//` et `%`
`+` et `-` (binaires)

Voici les règles :

— Commencer par effectuer les opérateurs de plus haut niveau de priorité ;
— Lorsque plusieurs opérateurs ont le même niveau de priorité, effectuer les opérations en commençant par les opérateurs les plus à gauche ...
— sauf lorsqu'il s'agit de l'opérateur puissance (où on commence par les opérateurs les plus à droite).

La figure 2.2 montre comment les priorités permettent de déterminer l'ordre des opérations dans une expression.

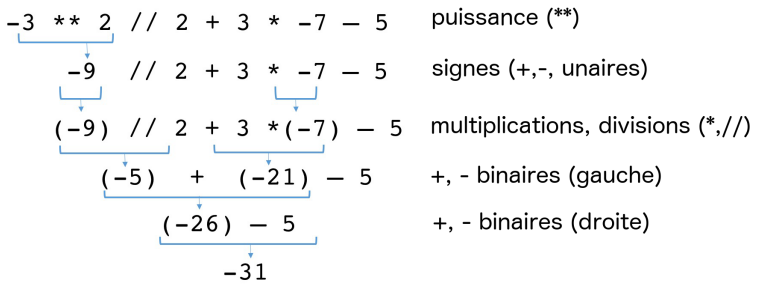


FIGURE 2.2 – Le niveau des priorités détermine l'ordre dans lequel on doit effectuer les opérations.

2.3 Les chaînes de caractères

2.3.1 Définition

les textes

Python permet de manipuler du texte. Les données représentant un texte sont de type *chaîne de caractères* (en anglais *character string* ou simplement *string*). Une chaîne de caractères est une juxtaposition de caractères. Il peut s'agir de caractères entrés au clavier :

— des lettres minuscules ou majuscules ;
— des chiffres ;
— des caractères espace, tabulation, retour à la ligne ;
— d'autres caractères visibles : `@, &, #, $, £, (, {, [` etc.

Mais les caractères peuvent aussi être invisibles (par exemple lorsque la chaîne n'a pas été entrée par l'utilisateur mais provient de la lecture d'un fichier).

les guillemets et apostrophes

Les chaînes de caractères définies de façon explicite (en anglais *string literals*) sont toujours encadrées par des apostrophes (en anglais *quotes*) ou des guillements (en anglais *double quotes*) de façon équivalente. Voici des exemples de chaînes de caractères : `"bonjour"` ou encore `'fzlseher@lh'`.

les opérateurs

Les opérateurs opérant sur des chaînes de caractères sont les opérateurs + et ⋆. Bien entendu, ces opérateurs n'ont pas du tout le même sens qu'avec les nombres.

+ : l'opérateur de *concaténation* ;

***** : l'opérateur de *duplication*.

2.3.2 La *concaténation*

L'opérateur de concaténation + doit avoir deux opérandes de type chaîne de caractères. L'expression résultante est aussi une chaîne de caractères obtenue en juxtaposant les deux opérandes.

Par exemple :

```
>>> "bonjour ! " + "comment allez-vous ?"
"bonjour ! comment allez-vous ?"
```

2.3.3 La *duplication*

L'opérateur de duplication ⋆ doit avoir une opérande de type chaîne de caractères et une opérande de type entier. L'expression résultante est aussi une chaîne de caractères obtenue en répétant la chaîne de caractères. Le nombre de répétitions est donné par l'opérande entière.

Par exemple :

```
>>> 3 * "Ohe "
"Ohe Ohe Ohe "
```

2.3.4 `len` : la longueur d'une chaîne

Si `s` est une chaîne de caractères, alors la valeur de `len(s)` est un entier représentant le nombre de caractères qui constituent `s`.

Par exemple `len('coucou')` vaut 6.

2.3.5 Les séquences d'*échappement*

Comment écrire une chaîne de caractères contenant des guillemets ou des retours à la ligne ? Par exemple dans : `'Ca m'interesse'` il y a trois apostrophes. Dans l'esprit de celui qui écrit ce message, la deuxième apostrophe n'a pas la même signification que les autres. La fonction de la première et de la troisième apostrophe est d'encadrer la chaîne de caractères, alors que la deuxième apostrophe doit être un caractère comme les autres. Il faut trouver un moyen de transmettre cette distinction à *python*.

La solution à ce problème (entre autres) est apportée par les *séquences d'échappement*.

En *Python*, le caractère `'\'` (en anglais *backslash*) a un sens particulier. Lorsqu'un \ est placé devant certains autres caractères, alors *python* n'interprète pas ces caractères de façon habituelle. Il interprète le backslash et le caractère suivant comme une *séquence d'échappement* représentant des caractères particuliers. Voici une liste non-exhaustive de séquences d'échappement :

Séquence	signification
\'	apostrophe texte
\"	guillemets texte
\\	backslash texte

Caractère	signification
\n	retour à la ligne
\t	tabulation
\b	retour en arrière

Voici un exemple d'utilisation de ces séquences. Le programme suivant :

```
>>> print('J\'ai dit non !!!')
J'ai dit non !!!
>>> print("Comment \"non\" ???")
Comment "non" ???
>>> print("\t No !!! \n \t Nein !!! \n \t Niet !!! \n \t M... !!! \n")
        No !!!
        Nein !!!
        Niet !!!
        M... !!!
```

2.4 nombres vs chaînes de caractères

Vous avez fait connaissance avec les nombres et les chaînes de caractères. Avant d'aborder d'autres notions, prenons un peu de recul.

2.4.1 Les opérateurs + et *

Ces deux opérateurs peuvent être utilisés aussi bien pour les nombres que pour les chaînes de caractères. Mais ils n'ont pas le même sens dans les deux cas. Cela implique quelques règles :

- Si l'une des deux opérandes de + est un nombre alors l'autre opérande doit aussi être un nombre. Dans ce cas, + est l'opérateur d'addition ;
- Si l'une des deux opérandes de + est une chaîne de caractères, alors l'autre opérande doit aussi être une chaîne de caractères. Dans ce cas, + est l'opérateur de concaténation ;
- Si l'une des opérandes est un nombre et l'autre une chaîne de caractères, alors le message d'erreur suivant est émis :

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

De même pour * :

— Si l'une des deux opérandes de $\star$ est un flottant alors l'autre opérande doit être un nombre (entier ou flottant). Dans ce cas, $\star$ est l'opérateur de multiplication ;
— Si l'une des deux opérandes de $\star$ est une chaîne de caractères, alors l'autre opérande doit être un ***entier***. Dans ce cas, $\star$ est l'opérateur de duplication ;
— Si l'une des deux opérandes de $\star$ est un entier, alors l'autre opérande peut être soit une chaîne de caractères ($\star$ est l'opérateur de duplication) soit un nombre ($\star$ est l'opérateur de multiplication).
— Si l'une des opérandes est une chaîne de caractères et l'autre une opérande non entier (par exemple flottant ou une autre chaîne de caractères) alors le message d'erreur suivant est émis :

```
TypeError: can't multiply sequence by non-int of type 'str'
```

ou

```
TypeError: can't multiply sequence by non-int of type 'float'
```

2.4.2 Les nombres et les textes-nombres

La différence entre `1` et `"bonjour"` est manifeste. Le premier est un nombre et le second est un texte. Le premier est un entier et le second est une chaîne de caractères. Mais que dire de `"1"` ou de `"3.45"` ?

— La chaîne `"bonjour"` est constituée de caractères qui sont des lettres ;
— La chaîne `"Mondiale 2018"` est constituée de caractères qui sont des lettres, des chiffres et un espace ;
— La chaîne `"2018"` est constituée de caractères qui sont exclusivement des chiffres.

Dans les trois cas, il s'agit de chaînes de caractères comprenant, éventuellement, des caractères particuliers que sont les chiffres.

Une chaîne de caractères constituée exclusivement de chiffres reste une chaîne de caractères. Il ne s'agit pas d'un nombre.

La meilleure façon de montrer la différence entre un nombre et une chaîne de caractères représentant un nombre est sans doute de les employer dans des opérations :

```
>>> 5 + 6
11
>>> "5" + "6"
"56"
>>> "numero" + "9"
"numero9"
>>> "numero" + 9
\texttt{TypeError: unsupported operand type(s) for +: 'int' and 'str'}
```

Les deux formes sont intéressantes :

— un nombre sous la forme d'un entier ou d'un flottant peut être ajouté, multiplié, divisé, alors que ces opérations ne sont pas possibles avec la forme chaîne de caractères.
— un nombre sous la forme d'une chaîne de caractères peut être intégré dans un texte, ce qui est impossible pour un entier ou un flottant.

Il est donc intéressant de pouvoir passer d'une forme à l'autre. C'est un des nombreux intérêts des fonctions de *conversion explicite*.

2.4.3 Conversions explicites

De façon générale, en *python*, chaque type de base est associé à une fonction de conversion qui convertit son argument en un élément de ce type de base. Cette fonction de conversion porte le même nom que le type lui-même (`int`, `float`, `str`). Bien entendu, comme nous le verrons, certaines conversions n'ont pas de sens et sont refusées par ces fonctions.

- Soit `x` un entier ou un flottant (ou autre, on le verra). Alors `str(x)` représente le même nombre que `x` mais sous la forme d'une chaîne de caractères ;
- Soit `s` une chaîne de caractères représentant un nombre entier. Alors `int(s)` représente le même nombre que `s` mais sous la forme d'un entier ;
- Soit `s` une chaîne de caractères représentant un flottant. Alors `float(s)` représente le même nombre que `s` mais sous la forme d'un flottant ;

Par exemple :

```
>>> str(132)
"132"
>>> str(2.5)
"2.5"
>>> int("52")
52
>>> float("5.2")
5.2
```

Par contre, toutes les chaînes de caractères ne peuvent pas être interprétées come des entiers ou des flottants :

```
>>> int("n13")
File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'n13'
>>> float("1,25")
File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: '1,25'
```

car `n13` ne peut pas être interprété comme un entier et `1,25` ne peut pas être interprété comme un flottant. Il aurait fallu utiliser le point `'.'` au lieu de la virgule `','`.

L'argument des fonctions `int` et `float` ne doit pas nécessairement être une chaîne de caractères.

- Soit `n` un entier. Alors `float(n)` représente la même valeur que `n` mais sous la forme d'un flottant ;
- Soit `x` un flottant. Alors `int(n)` représente la partie entière de `x`.

Par exemple :

```
>>> float(132)
132.0
>>> int(2.5)
2
```

2.5 Les variables

2.5.1 Définitions

La plupart des langages de programmation permettent d'associer une valeur donnée à un *nom*. Par exemple en *python*, on peut écrire :

```
taille = 8
```

Cette instruction associe le nom `taille` à la valeur 8. Après cette instruction, lorsque *Python* rencontrera le mot `taille`, il l'interprétera comme l'entier 8. On dit qu'une *variable* a été définie.

- L'instruction qui associe un nom à une valeur est appelée une *définition*.
- Une définition crée une *variable*.
- Une variable est caractérisée par une valeur, un type et un nom (appelé aussi *identificateur*).
- Cet identificateur peut être constitué de lettres (minuscules ou majuscules), de chiffres et de _ (underscore). Le premier caractère ne doit pas être un chiffre.

Les variables peuvent être utilisées en lieu et place de leur valeur. Elles peuvent intervenir dans des opérations ou encore permettre de définir d'autres variables. Par exemple :

```
>>> taille = 8
>>> aire = taille * taille
>>> print(taille)
8
>>> print(aire)
64
```

Réciproquement, l'utilisation d'une variable qui n'a pas été définie auparavant produit un message d'erreur de ce type. Dans l'exemple suivant, la variable `taille` n'a pas été définie.

```
>>> aire = taille * taille
File "<stdin>", line 1, in <module>
NameError: name 'taille' is not defined
```

2.5.2 Dissymétrie

- L'opérateur =, malgré sa ressemblance avec l'égalité mathématique, n'est pas symétrique. Autrement dit `a = b` n'est pas équivalent à `b = a`.
- Le membre de gauche doit être constitué d'une variable.
- Le membre de droite peut être constitué de n'importe quelle expression composée d'une ou plusieurs constantes ou variables.
- Le membre de gauche reçoit la valeur du membre de droite

En particulier :

```
>>> 8 = taille
File "<stdin>", line 1
SyntaxError: can't assign to literal
```

2.5.3 Opérateurs

Dans certains langages (en particulier les langages fonctionnelles), les variables gardent la même valeur pendant tout le programme (on parle, malgré tout de *variables*). Dans d'autres langages (en particulier en *python*), on peut redéfinir une variable, c'est à dire y associer une nouvelle valeur :

```
>>> a = 5
>>> a = 8
>>> print(a)
>>> 8
```

La nouvelle valeur peut être définie en fonction de la valeur précédente de la variable. Par exemple l'instruction suivante :

```
>>> a = a + 2
```

peut paraître mathématiquement surprenante. Mais, encore une fois, l'opérateur = ne représente pas l'égalité mathématique, mais une (re)définition de variable. Cette redéfinition a pour effet d'augmenter de 2 la valeur de `a`. *Python* permet de réaliser cette opération avec un opérateur spécial :

```
>>> a += 2
```

Chaque opérateur binaire est associé à un tel opérateur d'affectation. Supposons que la variable `a` soit égale à 7.2

opérateur	exemple	instruction équivalente	la valeur de `a`
`+=`	`a += 2`	`a = a + 2`	`9.2`
`-=`	`a -= 2`	`a = a - 2`	`5.2`
`*=`	`a *= 2`	`a = a * 2`	`14.4`
`/=`	`a /= 2`	`a = a / 2`	`3.6`
`//=`	`a //= 2`	`a = a // 2`	`3.0`
`%=`	`a %= 2`	`a = a % 2`	`1.2`
`**=`	`a **= 2`	`a = a ** 2`	`51.84`

2.5.4 Variables et chaînes de caractères

Les variables de type chaîne de caractères n'ont pas besoin d'être mis entre guillemets.

```
>>> message = "bonjour"
>>> message + " la compagnie"
'bonjour la compagnie'
>>> "message" + " la compagnie"
'message la compagnie'
```

Réciproquement, *python* essaiera d'interpréter comme une variable toute suite de caractères qui ne se trouve pas entre des apostrophes ou des guillemets. Par exemple :

```
>>> comp = " la compagnie"
>>> "bonjour" + comp
'bonjour la compagnie'
>>> bonjour + comp
File "<stdin>", line 1, in <module>
NameError: name 'bonjour' is not defined
```

2.5.5 Pourquoi utiliser des variables ?

Stocker pour ne pas refaire le calcul

Lorsqu'un même calcul doit être fait à plusieurs endroits d'un programme, il vaut mieux effectuer le calcul une fois et de mettre le résultat dans une variable. À partir de ce point, lorsqu'on aura besoin de cette valeur, on utilisera la variable.

Par exemple supposons que nous voulions résoudre l'équation linéaire du second ordre :

$$2x^2 + 6x - 4 = 0 \tag{2.1}$$

Il faut calculer le discriminant : $6^2 - 4 \times 2 \times 4$. Comme il est positif, il y a deux solutions :

```
print((-6-math.sqrt(6*6-4*2*4))/(2*2),(-6+math.sqrt(6*6-4*2*4))/(2*2))
```

La fonction `math.sqrt` représente la fonction racine carrée. Sans l'usage des variables, on doit calculer le discriminant trois fois. Une fois pour en vérifier le signe et deux autres fois pour chacune des deux solutions. Au lieu de cela, on peut écrire :

```
d = 6 * 6 - 4 * 2 * 4
r = math.sqrt(d)
e = 2*2
print( (-6-r)/e , (-6+r)/e )
```

Les valeurs littérales ne doivent apparaître qu'une seule fois

Une *valeur littérale* est une valeur apparaissant dans un programme sous la forme d'un chiffre. Une chaîne de caractère littérale est une chaîne entourée d'apostrophe ou de guillemets. Les données dans un programme apparaissent soit sous la forme de valeurs littérales soit sous la forme de variables.

Il est déconseillé de représenter une même grandeur sous forme littérale à plusieurs endroits d'un programme. Dans ce cas, s'il devient nécessaire de modifier cette grandeur, il faut modifier à tous les endroits où cette valeur a été disséminée. Par exemple dans le premier encadré ci-dessus, s'il fallait modifier le premier coefficient en 3, quels sont les 2 qu'on devrait modifier ? La solution est de définir une variable pour chaque coefficient et que la valeur explicite n'apparaisse qu'une seule fois :

```
m = 2
n = 6
o = 4
d = n * n - 4 * m * o
r = math.sqrt(d)
e = 2*m
print( (-n-r)/e , (-n+r)/e )
```

De cette façon, pour une nouvelle équation, il suffit de changer les valeurs de m, de n et de o.

Lisibilité

Il est essentiel qu'un programme soit lisible par d'autres personnes que celui qui l'a écrit. Il est aussi essentiel que ce programme reste lisible pour son auteur, ce qui ne va pas toujours de soi. Dans ce contexte, le nom que vous choisissez pour vos variables peut avoir un effet bénéfique ou non. Typiquement, dans le programme de l'encadré ci-dessus, il faut beaucoup réfléchir pour reconnaître la résolution d'une équation du second ordre. En *python*, les identifiants peuvent être aussi longues qu'on le souhaite. Pourquoi ne pas les utiliser cette possibilité pour rendre les programmes plus lisibles ? Voici une proposition :

```
a = 2
b = 6
c = 4
discriminant = b * b - 4 * a * c
racine_disc = math.sqrt(discriminant)
deux_a = 2*a
print( (-b-racine_disc)/deux_a , (-b+racine_disc)/deux_a)
```

2.6 Les répétitions : la boucle `for`

Voici un élément incontournable de beaucoup de langages et qui permet d'automatiser bien des tâches répétitives. La boucle `for` permet de répéter un ensemble d'instructions. Cette boucle prend la forme générale suivante :

2.6.1 Syntaxe

```
for <identificateur> in range(N):
<tabulation><instruction1>
<tabulation><instruction2>
<tabulation><instruction3>
...
<tabulation><instructionN>
<instructionM>
```

L'interpréteur *python* s'attend à ce que :
- l'instruction `for` se termine par le caractère `':'`
- une ou plusieurs des instructions suivantes commencent par un caractère tabulation ;
- une suite de caractères espace n'est pas équivalent à un caractère tabulation ;
- si l'instruction `for` commence déjà par un certain nombre de tabulations, alors les instructions suivantes devront en avoir un de plus ;

Dans ce cas, toutes les instructions suivant `for` et commençant par une tabulation supplémentaire seront exécutées `N` fois. Dans l'exemple ci-dessus, `<instructionM>` ne fait pas partie de la boucle et n'est pas répétée. Par exemple :

```
for i in range(3):
    print("bonjour")
print("Tout va bien ?")
```

affiche :

```
bonjour
bonjour
bonjour
Tout va bien ?
```

Ce programme a répété l'exécution de `print("bonjour")` trois fois. On dit qu'il y a eu 3 *itérations*.

2.6.2 Le *compteur* de la boucle

L'identificateur qui suit `for` n'a pas besoin d'avoir été défini auparavant. *Python* fait de cet identificateur une variable.
- À la première itération, cette variable vaut 0 ;
- à la deuxième itération, cette variable vaut 1 ;
- ...
- à la dernière itération, cette variable vaut `n-1`.

On l'appelle le *compteur* de la boucle.

```
for i in range(3):
    print(i)
print("Apres : ")
print(i)
```

affiche :

```
0
1
2
Apres :
2
```

Il est possible de commencer les itérations avec un compteur non-nul, en donnant un deuxième argument à l'opérateur `range`.

```
for i in range(1,4):
    print(i)
print("Apres : ")
print(i)
```

affiche :

```
1
2
3
Apres :
3
```

2.6.3 Boucles `for` imbriquées

À l'intérieur d'une boucle, on peut faire une deuxième boucle (voire une troisième ou une quatrième si nécessaire). Par exemple :

```
for i in range(2):
      for j in range(3):
            print((i,j))
```

affiche :

```
(0,0)
(0,1)
(0,2)
(1,0)
(1,1)
(1,2)
```

Chapitre 3

Booléens et conditionnelles

3.1 Le type booléen

Dans le chapitre 1 nous avons vu la notion de variable logique permettant d'évaluer la véracité d'une proposition logique. Ce type de variable peut prendre deux valeurs souvent représentées par 0 ou 1.

En python il existe un type de variable (au même titre que l'entier ou le flottant) permettant de représenter ces variables logiques : *le Booléen*. Une variable booléenne peut prendre deux valeurs : *True* (vrai) ou *False* (faux). Attention la majuscule est importante en python.

Dans ce chapitre nous verrons notamment dans quels cas ce type de variable est utilisé.

3.2 Comparaisons d'expressions

3.2.1 Opérateurs de comparaison

Dans le chapitre précédent, nous avons vu plusieurs opérateurs en python permettant d'agir sur des nombres ($+$, $//$, $\%$ ...) ou des chaînes de caractères ($+$, $*$). Il existe en python d'autres opérateurs permettant de comparer deux expressions :

Opérateur	Sens
>	strictement supérieur à
<	strictement inférieur à
>=	supérieur ou égal à
<=	inférieur ou égal à
==	égal à
!=	différent de
in	appartenant à
not in	n'appartenant pas à

Remarque : Attention à ne pas confondre l'opérateur "=" qui permet l'affectation d'une valeur à une variable (voir chapitre 2.5) et l'opérateur "==" qui permet la comparaison entre deux variables.

Par exemple les opérations suivantes sont valides en python :

```
>>> 1.5 > 2.5
```

```
>>> 2 == 1
```

```
>>> "world" in "hello world"
```

Mais quel est le résultat de telles opérations ?

Les opérations utilisant des opérateurs de comparaison retournent un booléen.
Ainsi, les opérations précédentes retournent les valeurs suivantes :

```
>>> 1.5 > 2.5
False
>>> 2 == 1
False
>>> "world" in "hello world"
True
```

On peut ainsi définir une variable booléenne comme n'importe quel autre type de variable :

```
>>> a = True
>>> print(a)
True
```

Et comparer des variables entre elles :

```
>>> a = 2
>>> b = 3
>>> a > b
False
```

De la même manière, on peut créer une variable contenant le résultat d'une opération de comparaison :

```
>>> variable_booleenne = (1 != 1)
>>> print(variable_booleenne)
False
```

3.2.2 Comparaison entre nombres de types différents

En python les opérateurs de comparaison (sauf le "*in*" et "*not in*") permettent de comparer tous les nombres quels que soient leur type (entier, flottant...).

Par exemple :

```
>>> 1.0 <= 1 # Comparaison entre flottant et entier
True
>>> 1e2 == 100 # Comparaison entre notation scientifique et entier
>>> True
```

Les opérateurs "*in*" et "*not in*" permettent de tester l'appartenance ou non de nombres dans des ensembles de nombres mais cela nécessite l'utilisation de structures que nous n'avons pas encore vues dans ce cours. Nous y reviendrons plus tard dans le chapitre 4.

3.2.3 Comparaison entre chaînes de caractères et Unicode

Les opérateurs de comparaison sont également définis pour comparer des chaînes de caractères (string).

Pour pouvoir comparer des chaînes de caractères, il faut définir un ordre permettant de dire pour tout couple d'éléments lequel est supérieur ou inférieur. Cet ordre dépend de la manière dont le texte est encodé.

L'encodage est le processus qui assigne à chaque caractère (par exemple "a", "A", "\", "2"...), un nombre unique appelé *code point*. Python 3 utilise par défaut l'Unicode pour encoder les chaînes de caractères. Pour plus d'informations sur l'encodage et l'Unicode vous pouvez aller voir la page suivante : doc python.

Le code point d'un caractère peut être affiché en utilisant la fonction ord() :

```
>>> ord("a")
97
>>> ord("b")
98
>>> ord("A")
65
>>> ord("/")
47
```

Lorsqu'on compare deux caractères avec un opérateur de comparaison, ce sont les code points des caractères qui sont comparés.

Par exemple :

```
>>> "a" > "A" # car ord("a") > ord("A")
True
>>> "a" > "b" # car ord("a") < ord("b")
False
```

En pratique, il faut retenir que les lettres sont ordonnées par ordre alphabétique et les chiffres sont ordonnés par ordre croissant. De plus, l'ensemble des chiffres sont positionnés avant les lettres majuscules qui sont elles même avant les lettres minuscules :

Caractère	**Code point**
"1", "2", ... "9"	de 49 à 57
"A", "B", ..., "Z"	de 65 à 90
"a", "b", ..., "z"	de 97 à 122

La comparaison de chaînes de caractères se fait caractère par caractère.

Pour les égalités "==", chaque caractère deux à deux doit être égal pour retourner *True* et pour les différences "! =", il suffit d'un caractère différent entre les deux chaînes pour retourner *True*.

```
>>> "bonjour" == "bonjour"
True
>>> "bonjour " == "bonjour"
False
>>> "bonjour" != "bonjour!"
True
```

Pour les inégalités strictes (">", "<"), les premier caractères de chaque chaînes sont comparés, s'ils sont différents, l'expression peut être évaluée (*True* ou *False*). Sinon les deux caractères suivants sont comparés, *etc*.

Si une chaîne est plus longue que l'autre et que la comparaison n'est plus possible car il manque un caractère, la chaîne la plus longue est considérée supérieure à la chaîne plus courte.

Voici quelques exemples détaillés :

```
>>> "bonjour" > "bonJour"
True
>>> "Bonjour" > "bonJour"
False
>>> "bonjour!" > "bonjour"
True
```

Dans le premier cas, la comparaison du "j" et "J" permet de conclure car *ord("j")* > *ord("J")*.

Dans le deuxième cas, la comparaison du "B" et "b" permet de conclure car *ord("B")* < *ord("b")*.

Dans le troisième cas, tous les caractères sont égaux deux à deux, sauf le dernier de la première chaine qui ne peut pas être comparé. La première chaîne étant plus longue que la deuxième, elle est considérée comme supérieure.

L'opérateur "*in*" renvoie *True* si chaque caractère de l'opérande gauche est présent dans le même ordre dans l'opérande droit. Inversement, l'opérateur "*not in*" renvoie *True* si la chaîne de caractère de l'opérande gauche n'est pas présente caractère par caractère dans l'opérande droit.

Voici quelques exemples :

```
>>> "hello" in "hello !"
True
>>> "Hello" in "hello !"
False
>>> "hello !" not in "helo !"
True
```

A retenir : La comparaison de chaînes de caractères n'est pas forcément intuitive. Il faut bien faire attention à la constitution des chaînes de caractères que vous comparez avant de vous en servir dans vos programmes. Si elles ne contiennent pas que des lettres par exemple, le résultat pourrait vous surprendre.

3.3 Structure conditionnelle

3.3.1 Condition *si*

Les structures conditionnelles permettent d'exécuter différentes opérations en fonction de l'évaluation d'une condition spécifique. Par exemple, supposons que nous ayons deux nombres quelconques stockés dans les variables "x" et "y" ; nous voulons un programme qui retourne la chaîne de caractère "x est plus grand que y" seulement si c'est le cas. Dans le cas contraire, rien ne se passe.

Voici comment l'écrire en python :

```
if x > y:
    print("x est plus grand que y")
```

Évidemment il faut au préalable avoir affecté des valeurs à x et y.

Analysons la structure de cette condition :

1. Le mot clef *if* doit être suivi d'une condition. Généralement cela passe par une proposition logique mettant en jeu un opérateur de comparaison (par exemple $a + b > 0$ ou $x! = 1$). En d'autres termes, le mot clef *if* doit être suivi d'un **booléen**.
   En effet, le résultat d'une comparaison renvoie bien soit *True* soit *False* comme nous l'avons vu dans la section précédente. Bien que cela présente peu d'intérêt dans un programme, nous pourrions tout à fait écrire :

   ```
   if True:
       print("c'est vrai")
   if False:
       print("c'est faux")
   ```

   Dans ce cas, le programme afficherait toujours "c'est vrai", mais n'afficherait jamais "c'est faux".
2. la condition après le *if* doit être suivie de " :" comme pour la boucle *for*.
3. l'ensemble des instructions à effectuer si la condition est vraie (ici seulement le *print*), doit être indenté sous le *if*.

3.3.2 Condition *si ... sinon*

Nous venons de voir comment faire pour effectuer une action seulement si une condition est vraie. Mais comment faire pour effectuer une autre action dans le cas où la condition est fausse ? Ceci se fait grâce au mot clef *else* :

```
if x > y:
    print("x est plus grand que y")
else:
    print("x n'est pas plus grand que y")
```

Dans ce cas là, une seule des deux phrases sera affichée en fonction du résultat de $x > y$. Vous noterez que le mot clef *else* doit être au même niveau que le *if* et doit être suivi des " :". Comme toujours, l'ensemble des instructions à effectuer dans le cas *else* doit être indenté sous le *else*.

3.3.3 Condition *si ... sinon si ... sinon*

Enfin, que faire dans le cas d'une condition plus complexe ? Reprenons l'exemple précédent avec la condition initiale $x > y$. Trois cas peuvent se produire :

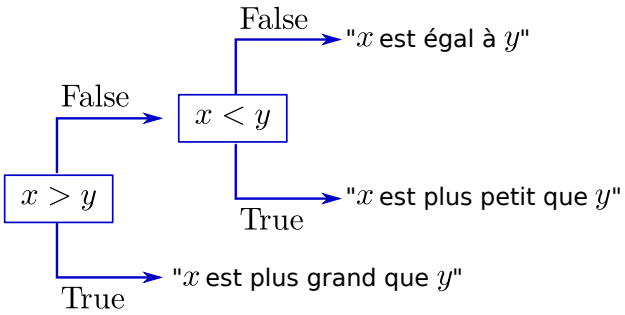
— x est strictement plus grand que y
— x est strictement plus petit que y
— x est égal à y

Si nous voulons que le programme indique dans quel cas nous somme, nous avons besoin d'un nouveau mot clef : *elif* (contraction de *else if*).

```
if x > y:
    print("x est plus grand que y")
elif x < y:
    print("x est plus petit que y")
else:
    print("x est egal Ã  y")
```

Le mot clef *elif* doit lui aussi être suivi d'une condition (c'est à dire d'un booléen). Lors de l'exécution du programme, la condition du *elif* n'est évaluée que si la condition précédente (ici celle du *if*) est fausse.

La figure ci-après est un diagramme des possibilités pour cet exemple :



Dans cet exemple, un seul *elif* est requis, mais autant de *elif* que nécessaire peuvent être ajoutés après un *if*. Chaque *elif* est alors évalué seulement si toutes les conditions précédentes ont été évaluées comme fausses.

Différence entre if ... if et if ... elif

Lorsque vous écrivez un programme, il est important de bien réfléchir au type de structure conditionnelle que vous allez utiliser. Les *if* et *elif* ne sont pas interchangeables.

Prenons les deux exemples suivant :

```
a = 5
b = 6
if a > 0:
    print("a>0")
if b > 0:
    print("b>0")
print("fin")
```

```
a = 5
b = 6
if a > 0:
    print("a>0")
elif b > 0:
    print("b>0")
print("fin")
```

Dans le premier cas le programme va retourner

```
a>0
b>0
fin
```

Alors que dans le deuxième cas, le programme va retourner :

```
a>0
fin
```

Dans le deuxième cas, la condition dans le *elif* est pourtant bien *True*. Seulement, le programme ne l'a jamais évalué. Il a dans un premier temps évalué la condition du *if*. Comme cette dernière est vraie, il a exécuté les lignes de code indentées sous le *if* puis est sorti de la structure conditionnelle.

3.4 Boucle while

La boucle *while* permet de répéter des instructions tant qu'une condition n'est pas remplie. Par exemple :

```
a = 0
while a <= 3:
    print(a)
    a+= 1
print("fin de boucle")
```

Dans ce cas là, le programme va évaluer la condition après le *while* ($a <= 3$).

Si elle est fausse le programme exécute les instructions dans la boucle puis re-teste la condition du *while*.

Si la condition est vraie, le programme sort de la boucle et exécute les instructions suivantes si il y en a.

Dans le cas de notre exemple, la sortie de ce code serait :

```
0
1
2
3
fin de boucle
```

Vous aurez peut être remarqué que la boucle *while* de l'exemple précédent peut être remplacée par la boucle *for* suivante :

```
for i in range(0,4):
    print(i)
print("fin de boucle")
```

Quand il s'agit d'implémenter des compteurs, la boucle for est effectivement plus judicieuse. La boucle while est elle par contre très utile lorsque le nombre d'itérations n'est pas connu à l'avance par le programmeur.

Imaginons que nous voulons écrire un programme qui demande à l'utilisateur de résoudre une multiplication, par exemple 7×4. Le programme doit afficher l'instruction autant de fois que nécessaire, jusqu'à ce que l'utilisateur entre effectivement la solution. On ne peut pas prédire à l'avance combien d'essais l'utilisateur va tenter. Ce code s'écrirait alors ainsi :

```
input_utilisateur = "0"
while input_utilisateur != "28":
    input_utilisateur = input("Combien font 7x4 ? ")
print("Bravo !")
```

La fonction *input* affiche la chaîne de caractères entre parenthèses et attend que l'utilisateur entre quelque chose au clavier. L'entrée de l'utilisateur est alors stockée dans la variable de sortie (ici *input_utilisateur*) sous forme de string.

Dans cet exemple, le programme commence par tester la condition du *while*. Au début comme *input_utilisateur* vaut "0", il exécute l'intérieur de la boucle et demande donc une valeur à l'utilisateur. Le programme test de nouveau la condition du *while*, si l'utilisateur entre 28, la condition est fausse et le programme sort de la boucle et affiche "Bravo", sinon il redemande une nouvelle valeur à l'utilisateur, etc.

Boucle infinie

Lorsque vous codez une boucle *while*, il faut faire bien attention que la condition demandée pourra être atteinte au cours de votre programme. Dans le cas contraire le programme entrera dans une boucle infinie et ne s'arrêtera jamais de tourner.

Par exemple, la boucle suivante est infinie :

```
a = 0
while a < 0:
    print(a)
    a+= 1
print("fin de boucle")
```

En effet, le programme commence par évaluer $a < 0$, ce qui est faux car $a = 0$, il exécute donc l'intérieur de la boucle. Il affiche "0" et a vaut maintenant 1. Puis le programme teste de nouveau la condition qui est toujours fausse etc. Ce programme va donc afficher les nombres entiers de 1 en 1 sans s'arrêter, jusqu'à ce que l'utilisateur tue le processus manuellement.

3.5 Combinaison de propositions par opérateurs logiques

Jusqu'à présent nous avons vu que les structures *if elif if* et les boucles *for* requerraient une proposition souvent sous la forme d'une comparaison. Il est également possible de tester plusieurs propositions en même temps en les combinant dans une "super" proposition.

Par exemple :

```
if a > 0 and a < 10:
    print(a)
```

Ici le programme affichera la valeur de a, seulement si la combinaison des deux conditions est vérifiée, c'est-à-dire si a est supérieur à 0 et inférieur à 10.

La combinaison de conditions peut se faire en utilisant les mots clefs "*and*", "*or*" et "*not*".
and représente le ET logique et *or* le OU logique (voir chapitre 1).

Il est possible de combiner autant de *and* et *or* que nécessaire dans une seule condition. Du point de vue informatique, qu'il y ait une seule condition ou plusieurs ne change rien. En effet, le résultat d'une comparaison ou d'une combinaison de comparaisons est toujours une valeur booléenne :

```
>>> 1 == 0 and 0 == 0
False
>>> 1 == 1 or 0 == 0
True
>>> 2+1==2 or 1<=1 and 1>2
False
```

not permet la négation d'une proposition logique. *not A* est vraie si A est fausse, et fausse si A est vraie.

Par exemple :

```
>>> not(1==1)
False
>>> not(2+1<0)
True
```

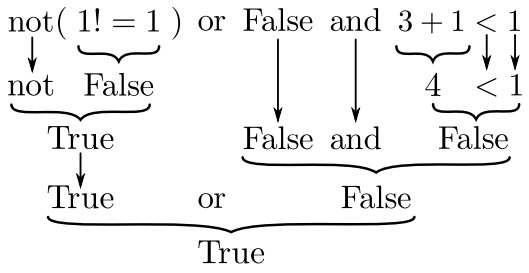
Retour sur les priorités entre opérateurs

Dans le chapitre 2.2.3, nous avons déjà vu qu'il existe des priorités entre opérateurs permettant à l'interpréteur python de savoir quelle opération effectuée en premier lorsqu'il rencontre une combinaison d'opérations.

Les opérateurs de comparaison ainsi que les opérateurs logiques que nous avons vus dans ce chapitre s'intègrent également dans cet ordre de priorité. L'ensemble des opérateurs vus jusqu'à présent sont rangés par **ordre de priorité décroissant** dans le tableau ci-après :

`()`
`**`
+ et − (unaires)
`*`, `/`, `//` et `%`
+ et − (binaires)
`in`, `not in`, <, <=, > , >=, `!=`, `==`
`not`
`and`
`or`

La figure suivante montre un exemple de l'ordre d'exécution des opérations en python :



3.6 Résumé

— Certaines expressions font intervenir des opérateurs de comparaison : $>$, $<$, $>=$, $<=$, $==$, $!=$, *in*, *not in*.

— Ces expressions sont évaluées par le programme qui retourne une valeur booléenne : *True* ou *False*

— Les comparaisons sont très utiles pour créer un programme non séquentiel à l'aide de la structure conditionelle "*if, elif, else*" :

```
if <condition>:
    <une ou plusieurs lignes de code>
elif <condition2>:
    <une ou plusieurs lignes de code>
else:
    <une ou plusieurs lignes de code>
```

— Il existe également une boucle utilisant des conditions : la boucle while qui s'exécute tant que la condition n'est pas vérifiée :

```
while <condition>:
    <une ou plusieurs lignes de code>
```

— Il est possible de combiner des comparaisons avec les mots clefs *and*, *or* et *not* et donc d'utiliser des conditions composées dans les structures *if, elif, else* et les boucles *while*

Chapitre 4

Listes et Tuples

4.1 Listes

4.1.1 Qu'est ce qu'une liste ?

Une *liste* est une structure de données représentant une séquence ordonnée d'éléments. Une liste a la propriété d'être *mutable* (on dit encore modifiable), c'est-à-dire que son contenu peut être modifié après sa création. On peut voir une liste comme un conteneur dont on peut agrandir ou rétrécir la taille, ou changer les éléments qui y sont stockés.

Une liste est représentée syntaxiquement par des crochets `[ ]`. On peut par exemple créer une liste d'entiers :

```
>>> liste_entiers = [1, 2, 3, 4, 5]
>>> print(liste_entiers)
[1, 2, 3, 4, 5]
>>>
```

Une liste peut ne pas contenir d'éléments, c'est une liste vide.

```
>>> ma_liste = []
>>> print(maliste)
[]
>>>
```

Le nombre d'éléments que contient la liste peut être obtenu avec la fonction `len()` :

```
>>> liste_entiers = [1, 2, 3, 4, 5]
>>> len(liste_entiers)
5
```

4.1.1.0.1 Types des éléments d'une liste L'exemple de la liste `liste_entiers` ne contenait que des éléments de même type (des entiers). Notez qu'il est d'usage d'utiliser les listes en n'y stockant que des éléments d'un même type. En effet, il est fréquent de vouloir appliquer une même opération à tous les élements de la liste, et avoir des éléments de type différents peut empêcher ce type d'opération.

Cependant, une liste en python peut contenir des éléments de types différents. Voici un exemple de définition de liste qui stocke une chaine de caractères, un entier, un flottant, et même une liste vide.

```
>>> liste_divers=['hello',1,5.4,[]]
>>> print(liste_divers)
['hello', 1, 5.4, []]
>>>
```

4.1.1.0.2 Liste de listes Nous avons vu qu'une liste peut contenir un élément de n'importe quel type. Une liste peut donc contenir d'autres listes. L'exemple ci-dessous illustre ceci. Deux listes sont créées (lignes 1 et 2). A la ligne 3, la partie droite `[ligne,colonne]` provoque la création d'une liste contenant nos deux listes précédentes, et affecte cette liste de listes à `tableau`.

Quand on demande l'accès au premier élément de la liste `tableau`, c'est donc une liste que l'on reçoit en retour (ligne 6). Pour accéder au premier élément de la première liste, il faut un niveau de crochets supplémentaire : `tableau[0][0]` (ligne 8).

```
1  >>> ligne=['a','b','c','d','e','f']
2  >>> colonne=[0,1,2,3,4,5]
3  >>> tableau=[ligne,colonne]
4  >>> tableau
5  [['a', 'b', 'c', 'd', 'e', 'f'], [0, 1, 2, 3, 4, 5]]
6  >>> tableau[0]
7  ['a', 'b', 'c', 'd', 'e', 'f']
8  >>> tableau[0][0]
9  'a'
10 >>> tableau[0][1]
11 'b'
12 >>> tableau[1][0]
13 0
```

Nous avons décrit ci-dessus les listes comme des structures de données, mais une liste est plus généralement un type de données. L'interpreteur, si on lui demande de quel type est une de nos listes précédentes, nous dirait par exemple :

```
>>> type(liste_divers)
<class 'list'>
```

Un type de données est défini par les valeurs que peut prendre ce type et les opérations définies sur ces valeurs. Pour les listes, nous venons de voir que les valeurs possibles sont toutes celles possibles pour les types python.

Voyons maintenant les opérations définies sur les listes.

4.1.2 Opérations sur les listes

Nous donnons dans cette section un ensemble minimaliste des opérations possibles sur les listes. Python offre davantage d'opérations que nous ne détaillons par pour l'instant, car elles peuvent être ré-écrites à partir du sous-ensemble présenté. L'encadré dans la section 4.4 pointe vers la documentation Python et l'ensemble des opérations offertes.

4.1.2.1 Accès aux éléments d'une liste

Une liste étant une séquence ordonnée, on peut accéder individuellement à un ou plusieurs élements de la liste en utilisant leur numéro d'ordre (ou *indice*). Le premier élément de la liste a le numéro 0.

La notation `[]` permet l'accès aux éléments de la liste.

4.1.2.1.1 Accès à un élément

On accède à un élément particulier :

```
>>> liste_divers=['hello',1,5.4,[]]
>>> liste_divers[2] # le 3e element de la liste a partir du debut
5.4
```

4.1.2.1.2 Accès illégal

Il est illégal de demander l'accès à un élément de liste non défini : si pour une liste `l` on tente d'accéder à `l[i]` avec `i >= len(l)`, la tentative provoquera l'erreur `IndexError: list index out of range`.

4.1.2.1.3 Accès à plusieurs éléments (sous-liste)

On peut également accéder à plusieurs éléments consécutifs [1] qu'on appelle souvent *tranches* (ou *slices*), en utilisant le symbole double-point vertical (`:`). On appelle ce mécanisme *slicing* dans la terminologie python. Il est disponible pour tous les types qui représentent des séquences : vous l'avez découvert sur les **chaines de caractères**, et il sera aussi applicable sur les **tuples**.

```
>>> liste_divers[0:2] # la sous-liste constituee des elements 0 et 1
['hello', 1]
```

Vous noterez qu'à partir du moment où l'on utilise le slicing (c'est-à-dire que `:` figure entre les crochets) la valeur retournée est une **liste** et pas un élément simple.

De manière générale, le slicing d'une liste `maliste` est défini comme suit, en considérant `a` et `b` deux entiers positifs ou nuls, avec `a<b`, et `a<len(maliste)` et $b \leq$ `len(maliste)` :

```
maliste[a:b]   # les elements de l'indice a a l'indice b-1
maliste[a:]    # les elements de l'indice a jusqu'a la fin de la liste
maliste[:b]    # elements a partir du debut jusqu'a b-1
maliste[:]     # tous les elements
maliste[a:a]   # liste vide : []
```

Le rang peut être un entier négatif, auquel cas la signifcation change :

```
maliste[-1]    # dernier element de la liste
maliste[-2:]   # les deux derniers elements
maliste[:-2]   # tous les elements sauf les deux derniers
```

1. Il est possible d'extraire plusieurs éléments non-consécutifs avec un troisième argument qui représente un *pas*. Par exemple, dans `maliste[a:b:p]`, on indique qu'on prend l'intervalle $[a; b-1]$ avec un pas de p. Nous n'utiliserons pas cette possibilité dans ce cours.

4.1.2.2 Modification d'une liste

Nous avons dit que les listes étaient des structures de données mutables (modifiables).

— On peut **modifier** un élément par une affectation :

```
>>> maliste[1]='abc'
>>> maliste
['hello', 'abc', 5.4, []]
>>>
```

— On peut **ajouter** un élément à une liste :
On utilisera pour cela l'opérateur de concaténation + (aussi défini pour concaténer deux chaines). Pour deux listes `l1` et `l2`, `l1 + l2` rend une nouvelle liste qui est constituée des deux listes mises bout-à-bout.

```
1 >>> l1 = [1,2,3]
2 >>> l2 = [4,5]
3 >>> l3 = l1 + l2
4 >>> l3
5 [1, 2, 3, 4, 5]
```

Pour ajouter un élément à une liste `l`, il suffit donc de concaténer `l` avec un élément sous forme de liste, et d'affecter le résultat à `l` (ligne 2 dans l'exemple suivant) :

```
1 >>> l = [1,2,3]
2 >>> l = l + [4]
3 >>> l
4 [1,2,3,4]
```

Bien entendu, on peut ajouter plus d'un élément du moment que l'ajout est une liste.
Notez que l'opérateur += est également défini et permet d'écrire la même chose de manière plus concise :

```
>>> l += [5]
>>> l
[1,2,3,4,5]
```

— On peut **supprimer** un élément de la liste.
On utilisera pour cela la fonction `del` qui supprime un élément à un indice donné.

```
>>> maliste
['a', 'b', 'c', 'd', 'e', 'f']
>>> del maliste[1]
>>> maliste
['a', 'c', 'd', 'e', 'f']
```

4.1.2.3 Affectation de listes

Le fait que listes soient modifiables soulève une question importante concernant la signification de l'opérateur d'affectation = entre listes. Si j'écris :

```
>>> l = ['a','b','c']
>>> m = l
```

`l` et `m` sont elles deux variables indépendantes, ou sont elles liées ? Dans le premier cas, si je modifie un élément de `l`, cela n'aura pas d'incidence sur `m`. Dans le deuxième cas, une modification de l'une des listes sera reflétée dans l'autre. Essayons :

```
>>> l[0]= 'z'
>>> m
['z', 'b', 'c']
```

Ceci met en évidence que l'opérateur = entre listes fait que `m` devient un synonyme (ou un alias) pour `l` : toute modification dans une liste est visible dans l'autre. Ces deux variables pointent en fait vers le même espace mémoire.

Cette proritété est également vraie pour les tuples et les chaines mais pose moins fréquement de difficultés car les variables de ces types sont non modifiables.

Quand on souhaite dire que deux listes sont identiques mais que chacune est indépendante, il faut utiliser l'instruction `list()`, que nous reverrons en section 4.3. Dans notre cas

```
>>> l = ['a','b','c']
>>> m = list(l)
>>> l[0]= 'z'
>>> m
['a', 'b', 'c']
```

fait que `m` est une recopie de `l` dans une nouvelle zone mémoire propre à `m`.

4.2 Tuples

4.2.1 Qu'est-ce qu'un tuple ?

Les *tuples* (on dit aussi *n-uplets* en français) partagent beaucoup de points communs avec les listes. Un tuple représente également une séquence d'éléments, qui peuvent être de type différent. Une différence fondamentale est qu'ils sont *unmutable*, c'est-à-dire non modifiables après leur création [2].

Syntaxiquement, on utilise la virgule pour dénoter un tuple :

```
>>> t=1,2,3
>>> print(t)
(1, 2, 3)
```

Même si ce n'est pas obligatoire, il est usuel d'ajouter des parenthèses autour de l'ensemble d'éléments. L'utilisation de parenthèses est en revanche nécessaire pour imbriquer des tuples les uns dans les autres (voir plus loin).

Comme les listes, les tuples peuvent contenir des éléments de types différents.

```
>>> personne=('Jo','Bar',34,56.4)
>>> print(personne)
('Jo', 'Bar', 34, 56.4)
```

Deux cas particuliers existent : le cas des tuples à 0 et 1 élément. On note le 0-uplet à l'aide de parenthèses ouvrantes et fermantes ne contenant rien, et on note le 1-uplet avec une virugle sans rien derrière :

2. Cependant, un élément du tuple peut contenir des éléments qui sont eux modifiables.

```
>>> vide = ()
>>> singleton = 'hello',
>>> len(vide)
0
>>> len(singleton)
1
```

4.2.2 Opérations sur les tuples

4.2.2.1 Accès aux éléments d'un tuple

On peut distinguer deux modes d'accès aux éléments d'un tuple `t` :

— par l'indice `i` de l'élément voulu dans cette séquence, exactement comme nous l'avons fait pour les listes, avec la notation crochet `t[i]`. Le mécanisme de slicing est également applicable.

— par le mécanisme d'*unpacking* qui consiste à attribuer en une fois l'ensemble des éléments du tuple à des variables différentes. Dans l'exemple suivant, après avoir créé un tuple `personne` (ligne 1), on accède (ligne 2) aux valeurs stockées dans personnes en affectant respectivement aux variables `prenom`, `nom`, `age` et `poids` les éléments `personne[0]`, `personne[1]`, `personne[2]` et `personne[3]`.

```
1  >>> personne='Jo','Bar', 34, 78.5
2  >>> (prenom,nom,age,poids) = personne
3  >>> print(prenom)
4  Jo
```

4.2.2.2 Modification d'un tuple

Nous avons dit qu'on ne pouvait pas modifier un tuple. On peut le vérifier sur cet exemple où l'on essaie de modier le premier élément du triplet `t` après l'avoir défini :

```
>>> t = ('jaune',3,4)
>>> t[0]= 'vert'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

Cependant, si l'on ne peut pas modifier **le contenu** d'un tuple existant, il est possible de le re-définir, de manière similaire à ce que nous avons vu pour ajouter un élément à une liste (section 4.1.2.2). Il est par exemple possible d'écrire :

```
>>> t = ('jaune',3,4)
>>> t += (5,6)
>>> t
('jaune', 3, 4, 5, 6)
```

Le tuple a changé, mais cela s'est fait par la création d'un nouveau tuple obtenu par concaténation de `('jaune',3,4)` et `(5,6)`, qui a ensuite été utilisé pour ré-définir `t`.

4.3 Conversions entre chaines, listes, tuples

Nous avons vu jusqu'ici trois types dits *iterable* dans la terminologie Python : les chaines de caractères, les listes et les tuples. Ceci signifie qu'ils permettent de stocker un ensemble d'éléments que l'on peut parcourir. Vous avez d'ailleurs pu remarquer que ces trois types bénéficient du même mécanisme de *slicing* évoqué précédemment. En raison de cette propriété commune, on peut facilement convertir un ensemble d'éléments d'un type vers un autre, ce qui peut s'avérer bien pratique.

Dans les exemples suivants, on transforme :

— la chaine `s` en une liste `l` (ligne 2)
— la chaine `s` en un tuple `t` (ligne 5)
— la liste `l` en un tuple `t2` (ligne 8)
— le tuple `t` en une liste `l2` (ligne 11)

```
1  >>> s="bonjour"
2  >>> l = list(s)
3  >>> l
4  ['b', 'o', 'n', 'j', 'o', 'u', 'r']
5  >>> t = tuple(s)
6  >>> t
7  ('b', 'o', 'n', 'j', 'o', 'u', 'r')
8  >>> t2 = tuple(l)
9  >>> t2
10 ('b', 'o', 'n', 'j', 'o', 'u', 'r')
11 >>> l2 = list(t)
12 >>> l2
13 ['b', 'o', 'n', 'j', 'o', 'u', 'r']
14 >>>
```

4.4 Pour aller plus loin

Les opérations présentées ci-dessus sont suffisantes pour exprimer les algorithmes que nous voulons écrire en cours ou dans les exercices. Il existe cependant bien d'autres opérations définies sur les listes ou sur les tuples. L'encadré qui suit pointe vers les pages de la documentation officielle de python listant l'ensemble des méthodes sur les listes et tuples. La lecture de cette documentation doit se faire après avoir appris à maitriser les opérations élémentaires définies dans le cours.

> Les **listes** et les **tuples** sont tous deux des types particuliers de *séquence*. À ce titre, les opérations définies sur les séquences sont aussi définies sur les listes et sur les tuples. Il s'agit d'opérations permettant de tester l'appartenance, de concaténer, d'accéder aux éléments, de trouver les valeurs minimales ou maximales, de trouver l'indice de la première occurence d'une valeur, ou de compter le nombre d'éléments d'une occurence. La liste de ces opérations est donnée précisément dans la documentation python 4.6.1. Common Sequence Operations
>
> Les **listes** présentent la particularité d'être *mutable* (modifiable). A ce titre, elles bénéficient en plus des opérations sur le sous-type particulier des *séquences modifiables*. Ces opérations permettent de modifier les éléments de la liste, ou d'effacer,

d'agrandir ou rétrécir la liste, ou de l'inverser. La liste de ces opérations est donnée précisément dans la documentation python 4.6.3. Mutable Sequence Types

4.5 Exemples simples

4.5.1 Parcours de liste

Une des opérations les plus courantes consiste à parcourir une liste. Pour afficher les éléments d'une liste avec la fonction `print()`, nous allons accéder à chaqué élément à travers son rang (on dit aussi son indice) dans cette liste.

— Une première façon de faire est de construire l'ensemble des indices de la liste. Pour cela, nous pouvons utiliser la fonction `range()` vue dans le chapitre 2 : `range(n)` nous donne les entiers successifs $[0; n-1]$. Pour une liste `l`, le premier indice étant 0 et le dernier étant la longueur de la liste moins un, nous obtenons les indices de `l` avec `range(len(l))`. Nous pouvons ensuite utiliser `for` pour itérer, en faisant prendre à une variable `i` toutes les valeurs d'indice et accéder à chacun des éléments à l'aide de la notation `l[i]` :

```
>>> l=['a','b','c','d']
>>> for i in range(len(l)):
...     print(l[i])
a
b
c
d
```

— Cependant, on peut obtenir le même résultat avec une écriture plus simple. En effet, la variable suivant le `for` est ce qu'on appelle un *itérateur*. Dans l'exemple précédent, l'itérateur `i` prenait tour à tour les valeurs des indices. Rien n'empêche que l'itérateur prenne directement, tour à tour, les valeurs des éléments de la liste. Ceci s'écrit (et la notion d'indice disparait complètement).

```
>>> l=['a','b','c','d']
>>> for e in l:
...       print(e)
...
a
b
c
d
```

— Quand on souhaite, à la fois parcourir la liste avec un itérateur, et connaître l'indice courant dans le parcours, on peut utiliser un objet `enumerate`. Si on considère une liste `l`, `enumerate(l)` construit une "liste" constituée de tuples *(indice, élément)* que l'on peut parcourir avec un itérateur :

```
>>> for e in enumerate(l):
...     print(e)
...
(0, 'a')
```

```
(1, 'b')
(2, 'c')
(3, 'd')
```

Prenons l'exemple suivant. Je dispose de deux listes de même taille : `l1`, liste d'entiers, et `l2` une liste de lettres. Pour chaque indice `i`, je veux remplacer l'élément `l2[i]` par un espace quand `l1[i]` est nul. J'ai donc besoin de l'indice `i` pendant le parcours.

```
>>> l1=[1,1,0,0,1]
>>> l2=['h','e','l','l','o']
>>> for (i,elem) in enumerate(l1):
...     if elem==0:
...         l2[i]=' '
...
>>> print(l2)
['h', 'e', ' ', ' ', 'o']
>>>
```

A la ligne 3, le `for` présente à chaque itération un nouveau tuple `(i,elem)` correspond au contenu de la liste `l1` parcourue.

4.5.2 Somme d'une liste de nombres

```
>>> l=[1,2,3,4,5,6,7,8,9]
>>> somme=0
for e in l:
...     somme += e
...
>>> print(somme)
45
```

Est ce que le même programme fonctionnerait avec une liste de chaines de caractères ? Quel en serait le résultat ?

4.5.3 Traitement d'une liste de tuples

Dans cet exemple, nous définissons un ensemble de points que nous allons stocker dans une liste `l`. Chaque point est un triplet $(couleur, x, y)$ où x et y sont les coordonnées du point. On veut construire une nouvelle liste `nl` dans laquelle les points de couleur rouge ont leurs coordonnées divisées par 2.

```
1  p0 = ('bleu',0,0)
2  p1 = ('rouge',1.5,0)
3  p2 = ('vert',4,1)
4  p3 = ('vert',2,-3)
5  p4 = ('bleu', 4,3)
6  p5 = ('rouge',-5,3.5)
7  l = [p0,p1,p2,p3,p4,p5]    # on met ces tuples dans une liste
8
9  nl=[]                      # la nouvelle liste a construire
10 for (couleur,x,y) in l:    # pour chaque element de l (un triplet)
```

```
11      if couleur=='rouge':
12          nl += [(couleur,x/2,y/2)]  # ajouter a la liste avec coords/2
13      else:
14          nl += [(couleur,x,y)]      # ajouter le point inchange a la liste
15
16  print(nl)
```

On remarque qu'à la ligne 10, l'itérateur qui prend chaque élément de la liste, utilise l'*unpacking* : l'élément de liste parcouru, qui est un triplet, est directement décomposé en trois champs `couleur`, `x`, et `y`. On aurait aussi pu écrire

```
for e in l:                    # pour chaque element de l (un triplet)
    couleur,x,y = e
    if couleur=='rouge':
        nl += [(couleur,x/2,y/2)]  # ajouter a la liste avec coords/2
    else:
        nl += [e]                # ajouter le point inchange a la liste
```

Chapitre 5

Structurer les grands programmes

5.1 Un projet en plusieurs morceaux

Dans les projets d'une certaine ampleur, on ne peut pas écrire le programme d'une seule traite. Ils nécessitent un travail amont consistant à discerner, dans ce projet, différentes opérations réalisées par différents programmes ou morceaux de programmes qu'on doit assembler.

5.1.1 Exemple de projet

Soit un projet imaginaire où nous ayons besoin de calculer, plusieurs fois, l'exponentiel de certaines valeurs avec précision [1]. Voici des extraits de ce programme.

```
...
terme = 5
resultat = 0
n=-3
a = 1.0

# Nous avons besoin de calculer l'exponentiel de a

...
b=5

# Nous avons besoin de calculer l'exponentiel de b

print(n, terme, resultat)
```

FIGURE 5.1 –

1. Dans le projet labyrinthe, nous aurons aussi besoin, de nombreuses fois de connaître la liste des voisins d'une cellule ou encore de trouver l'indice d'un élément dans une liste. Les besoins sont différents, mais le propos est le même

5.1.2 Le calcul de e^x

Par ailleurs, en mathématique, on montre que :

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + ... \tag{5.1}$$

Voici un programme qui calcule l'exponentiel d'un nombre entré par l'utilisateur en calculant les 20 premiers termes de la série ci-dessus. Ce programme se base sur le fait que, pour avoir le (n+1)-ième terme en fonction du n-ième terme, il faut multiplier celui-ci par $x/(n+1)$.

```
x = float(input("Entrer un nombre : "))
resultat = 0
terme = 1
nb_termes = 20
for n in range(nb_termes):
    resultat += terme
    terme *= x / (n+1)
print("exp(" + str(x) + ")=" + str(resultat))
```

FIGURE 5.2 – Le calcul de l'exponentiel

En choisissant 1.0 pour la valeur de `x`, on obtient le message :

```
exp(1.0)=2.7182818284590455
```

On suppose que cette approximation est suffisamment précise pour notre application.

5.1.3 Comment assembler les programmes ?

Dans le programme ci-dessus, si on enlève la première et la dernière ligne, on obtient un programme qui calcule l'exponentiel de la variable `x` et met le résultat dans la variable `resultat`. La tentation est grande de faire un copier/coller et de mettre ces lignes de programme, là où on souhaite calculer l'exponentiel (ici à deux endroits différents).

5.2 Problèmes soulevés par l'assemblage brut de programmes

Les problèmes soulevés par un tel assemblage sont :
— le manque de lisibilité ;
— la redondance ;
— l'interférence entre des programmes (la non modularité) ;
— le test.

5.2.1 Lisibilité

Les programmes doivent être lisibles et pas seulement par l'ordinateur. En effet, dans la plupart des cas, les programmes, même s'ils ne comportent pas d'erreur manifeste, doivent pouvoir

```
...
terme = 5
resultat = 0
n=-3
a = 1.0

# Nous avons besoin de calculer l'exponentiel de a

x = a
resultat = 0
terme = 1
nb_termes = 20
for n in range(nb_termes):
    resultat += terme
    terme *= x / (n+1)
...
b=5

# Nous avons besoin de calculer l'exponentiel de b

x = b
resultat = 0
terme = 1
nb_termes = 20
for n in range(nb_termes):
    resultat += terme
    terme *= x / (n+1)

print(n, terme, resultat)
```

FIGURE 5.3 – L'assemblage des deux codes

évoluer et être modifiés, par l'auteur lui-même ou par quelqu'un d'autre. Cela implique que ces programmes soient *lisibles*.

Or qui, en lisant le programme ci-dessus, peut deviner que la boucle `for` qui apparaît deux fois est destinée à calculer une valeur d'exponentiel ? On peut peut-être améliorer les choses en ajoutant des commentaires :

```
...
x = a
#-------- Calcul de l'exponentiel de a : dabut --------
resultat = 0
terme = 1
nb_termes = 20
for n in range(nb_termes):
    resultat += terme
    terme *= x / (n+1)
#-------- Calcul de l'exponentiel de a : fin --------
...
```

5.2.2 La redondance

Un autre problème est celui de la redondance. Le code calculant l'exponentiel se trouve en deux endroits différents. D'autres programmes pourraient avoir besoin de l'utiliser bien plus souvent. Si on imagine que ce code n'aura jamais besoin d'être modifié, alors le fait de le dupliquer ne pose aucun problème. Mais si, pour une raison ou pour une autre, nous avons besoin de le modifier ce code, ne serait-ce que pour en augmenter la précision, en calculant plus de termes, alors il faut parcourir tous les endroits où ce code a été copié et y apporter le changement. Or plus le nombre d'exemplaires est élevé et plus ce changement vous prendra de temps et plus vous prenez le risque de vous tromper.

En l'occurrence, la redondance est à éviter à tout prix.

5.2.3 L'interférence entre les programmes

Comme vous l'avez sans doute remarqué, le programme de la figure 5.1.1 et celui de la figure 5.1.2 comportent, par coïncidence, des variables qui comportent les mêmes noms : `n`, `terme` et `resultat`.

Dans le programme de la figure 5.1.1, la dernière instruction `print(n,terme,resultat)` affichait : `-3 5 0`. Par contre, dans le programme de la figure 5.3, la même instruction affiche `20 4.1103176233121634e-19 2.7182818284590455`. Donc le simple fait d'avoir calculé un exponentiel modifie un comportement qui n'avaient peut-être aucun rapport avec ces exponentiels.

Enfin, dans le programme de la figure 5.1.2, si on omettait la troisième ligne `(terme = 1)`, l'interpréteur nous aurait averti que la variable `terme` n'avait pas été initialisée avant d'être modifiée. Alors que si on omet la même ligne dans le programme de la figure 5.3, l'interpréteur ne se plaint pas, puisque la variable `terme` (celle qui n'a rien à voir avec les exponentiels) a bien été définie mais avec une valeur de 5. Ainsi le programme produit, en toute bonne foi, une valeur erronnée de l'exponentiel.

Ainsi, avant d'assembler deux morceaux de code de cette façon, il faut veiller à ce que les variables des deux morceaux ne portent jamais le même nom, sous peine que les variables d'un morceau n'interfèrent avec celles de l'autre morceau. Au besoin, il faut changer ces noms de variable, avec des noms différents pour chaque exemplaire.

Évidemment, toutes ces manipulations ne sont pas envisageables.

5.2.4 Testabilité

Enfin, le test d'un programme comme celui de la figure 5.1.2 est relativement facile. Ce programme possède une entrée (la valeur `x` entrée par l'utilisateur) et une sortie (la valeur de `resultat` affiché à la fin du programme). Pour tester le programme, il suffit de vérifier si les valeurs affichées correspondent bien à l'exponentiel de `x`.

Par contre, dans un grand programme, le même morceau de code est beaucoup plus difficile à tester puisque son comportement ne dépend plus seulement de deux variables, mais potentiellement d'une multitude d'autres variables, définies au même niveau.

5.3 Solution : isoler/nommer/centraliser les morceaux de programme

Pour pouvoir assembler des morceaux de programme, voici ce dont nous aurions besoin.

5.3.1 Isoler

Il n'est pas viable d'assembler des programmes en changeant tous les noms de variables. Nous aurions besoin de spécifier que les variables dans un bloc de programme donné soient considérées par l'interpréteur comme distinctes des variables n'appartenant pas à ce bloc, même quand elles portent le même nom. Ces blocs ne doivent partager avec le reste du programme que des données d'entrée et le résultat du calcul.

Dans le cas de la figure 5.3, il faudrait que les variables `n`, `terme` et `resultat` permettant de calculer l'exponentiel soient considérées comme distinctes des variables de même nom dans le reste du programme.

5.3.2 Nommer

Le fait de pouvoir explicitement nommer ces blocs isolés augmenterait la lisibilité du programme. Le bloc de la figure 5.1.2 serait simplement appelé `exponentiel`.

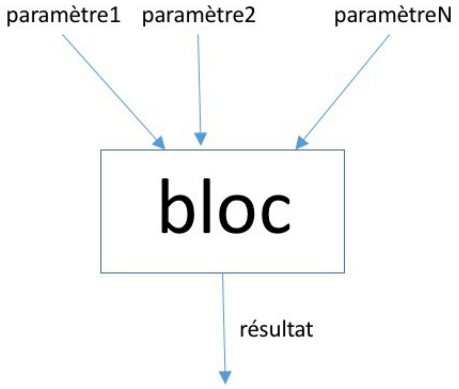
5.3.3 Centraliser

Pour réduire la redondance, il faudrait que les blocs utilisés à différents endroits existe en un seul exemplaire dans le programme et que chaque utilisation de ce morceau de code fasse référence à cet exemplaire unique. De cette façon, lorsque ce bloc est modifié, il l'est partout.

Nous le verrons au chapitre suivant, ces blocs isolés, nommés et centralisés sont simplement les *fonctions*.

5.4 Conséquences

Nous supposons qu'il est possible d'isoler un bloc de programme (une *fonction*) de façon à ce que son comportement ne dépende que d'un nombre limité de paramètres et qu'il produise un nombre limité de résultats. Il en résulte que, quelque soit l'endroit ou le moment où ce bloc de programme est exécuté, pour les mêmes valeurs de paramètres, les résultats seront strictement les mêmes. Le fait que le comportement du bloc ne dépende que de la valeur des paramètres rend ce bloc beaucoup plus facilement testable.



5.4.1 La *modularité*

Cela signifie aussi que ce bloc peut être utilisé à de nombreux endroits du programme sans avoir à changer le noms des variables et en garantissant que le comportement du code sera toujours conforme. Cela rend ce bloc réutilisable à l'infini.

Du fait que le comportement de ces blocs ne dépende que de la valeur des paramètres entraîne que, pour utiliser ces blocs, il n'est pas utile de connaître son fonctionnement interne. Il suffit de connaître la relation entre les entrées et la sortie. Cela rend ces morceaux de programme utilisables par des utilisateurs autres que celui qui a écrit ce programme.

Une fois qu'un programmeur dispose d'un grand nombre de ces blocs avec, pour chacun d'eux, la connaissance de la relation entre les entrées et la sortie, ce programmeur peut réaliser un grand nombre de projets simplement en assemblant ces blocs.

Enfin, même si on ne dispose pas de blocs déjà écrits, ces blocs définissent l'unité de construction des programmes. Lorsqu'on imagine un projet, on peut l'imaginer sous la forme de l'assemblage de blocs, chaque bloc étant défini par la relation entre ses entrées et ses sorties. Chaque grand bloc peut être constitué de blocs plus petits. Ainsi, lorsque le comportement de tous les blocs a été spécifié, il ne reste plus qu'à écrire chaque bloc.

Le fait qu'un programme puisse être considéré comme un assemblage de blocs en fait un programme *modulaire*.

5.5 Exemple : les ensembles de *Mandelbrot*

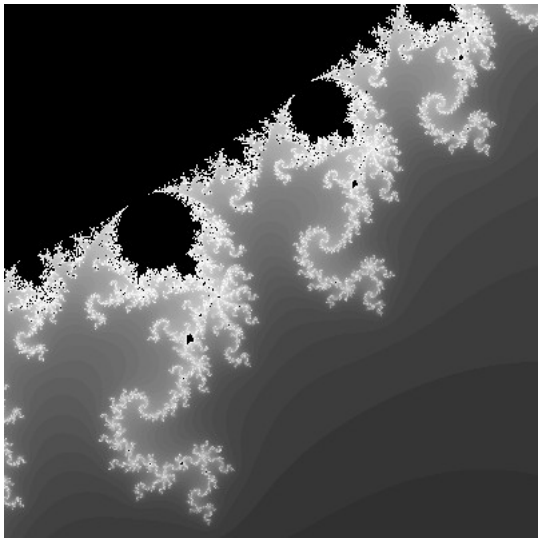


FIGURE 5.4 – Une vue particulière de l'ensemble de *Mandelbrot*.

5.5.1 Définition

Chaque point (x_0, y_0) du plan permet de définir la suite complexe suivante :

$$\begin{cases} z_0 = x_0 + iy_0 \\ z_{n+1} = z_0 + z_n^2 \end{cases} \tag{5.2}$$

Soit ϵ un nombre réel. L'ensemble de *Mandelbrot* associe à chaque point (x_0, y_0) du plan le nombre N tel que $|z_N| < \epsilon$. Si, pour certaines points, $|z_{255}| \geq \epsilon$ alors la valeur associée à ce point est 0 (noir sur la figure 5.4).

5.5.2 Décomposition en blocs

Voici une décomposition du haut vers le bas *top-down*. Autrement dit, on part du résultat recherché et, chaque fonctionnalité est, à son tour décomposée en fonctionnalités plus simples jusqu'à ce que l'opération apparaisse assez simple pour ne pas être décomposée.

1. Tout d'abord la réalisation d'un ensemble de *Mandelbrot* nécessite de savoir créer une image. Imaginons un bloc appelé `PGM_save` qui, à partir d'un nom de fichier, des dimensions de ce fichier et d'une liste de pixels créer un fichier image représentant ces pixels.
2. Ensuite, nous avons besoin d'un bloc `Mand_makeList`, qui parcourt les pixels (x, y), qui calcule, pour chaque pixel la valeur de l'ensemble de *Mandelbrot* et qui remplit une liste de pixels avec cette valeur.

Grâce à ces deux blocs, le projet peut être réalisé. Autant le premier projet paraît simple (vous l'avez presque réalisé en TP), autant le deuxième bloc doit encore être décomposé en bloc plus simples :

3. Créer un bloc `convPixelsComplexe`, qui convertit les coordonnées des pixels (x, y), où le point (0,0) est en haut à gauche, en des complexes (x_c, y_c) où le complexe $(0, 0)$ est au milieu de l'image, les bords étant à -1 et à 1.
4. Créer un bloc `Mand_cplx` qui, à partir d'un nombre complexe $z_0 = x_0 + iy_0$ calcul la valeur de l'ensemble de *Mandelbrot*.

Encore une fois, c'est le deuxième bloc qui est devenu plus simple, mais toujours trop difficile pour être écrit.

5. Le bloc `Mand_cplx`, ci-dessus nécessite de calculer la suite $z_{n+1} = z_0 + z_n^2$ ainsi que le module $|z_n|$. Donc ce bloc nécessite qu'on puisse ajouter, multiplier des complexes ainsi que de calculer le module d'un complexe.

-

Chapitre 6

Fonctions et modules

6.1 Vocabulaire : argument, valeur de retour, appel

6.1.1 Les fonctions en mathématique

La notion de fonction ne vous est sans doute pas étrangère, mais vous l'avez sans doute rencontrée dans un contexte mathématique. Prenons l'exemple de la fonction *cosinus*. En mathématique, lorsqu'on écrit $cos(0)$:

— cos est la fonction ;
— 0 est appelé la *variable* ;
— $cos(0)$ est appelé *l'image de* 0 *par la fonction* cos.

6.1.2 Les fonctions en informatique

En python, vous avez déjà utilisé des fonctions. Par exemple `print`, `input`, `open`, `int`, `random.random` sont des fonctions informatiques. `abs` et `sum` sont aussi des fonctions :

```
>>> abs(-10)
10
>>> int(3.141592)
3
>>> sum([3, 5, 8])
16
```

Autre exemple : la fonction `cos`, qu'on peut importer du module `math` (que nous verrons plus loin), représente la fonction *cosinus*. Dans l'expression `cos(0)` :

— `cos` est la fonction ;
— `0` est appelé *l'argument* ;
— `cos(0)` est appelé *la valeur de retour* de `cos` pour `0`.

En l'occurrence, cette valeur de retour, `cos(0)`, vaut 1.0. Par exemple :

```
y = cos(0)/5        # y prend la valeur de 0.2
print(cos(0))       # affiche 1.0
```

En mathématique, les parenthèses ne sont pas toujours obligatoires. On peut quelquefois écrire $cos\theta$. En *python3*, l'appel à une fonction se fait nécessairement avec des parenthèses qui contiennent les arguments à transmettre à la fonction.

Lorsque l'interpréteur python lit une instruction contenant l'expression `cos(0)` il calcule la valeur de la fonction cosinus pour 0. On dit que l'interpréteur a *appelé* la fonction `cos`.

Ce qu'il faut retenir :

- En informatique, la variable d'une fonction est appelée son *argument*.
- Le(s) argument(s) doivent être entouré(s) de parenthèses.
- La valeur d'une fonction pour un argument donné est aussi quelquefois appelée la *la valeur de retour* de la fonction.
- Lorsque l'interpréteur calcule la valeur d'une fonction pour des valeurs spécifiques d'argument(s), on dit que la fonction a été *appelée*.

6.2 Différentes sortes de fonctions en python

6.2.1 Le nombre d'arguments

Vous avez déjà utilisé la fonction `open` pour ouvrir un fichier.

```
f = open("vermeer.pgm","w")
```

Cette fonction nécessite (au moins) deux variables. Vous avez également utilisé la fonction `random.random` dont la valeur de retour est un nombre aléatoire choisi entre 0 et 1.

```
alea = random.random()
```

Cette fonction n'admet aucun argument. Vous avez aussi utilisé la fonction `input` qui peut être utilisé avec 1 argument :

```
nom = input("Veuillez saisir votre nom:")
```

ou avec 0 arguments :

```
print("Veuillez saisir votre nom:")
nom = input()
```

Autrement dit, pour certaines fonctions, certains arguments peuvent être optionnels. Cela signifie qu'en l'absence de cet argument, la fonction attribue à ces arguments une valeur par défaut spécifiée dans la fonction. En l'occurrence, en l'absence d'argument, `input` considère que l'argument est la chaîne vide : "". Ce qu'il faut retenir est qu'en python :

- Une fonction peut avoir 0, 1 ou plusieurs arguments.
- Le nombre et l'ordre des arguments est important.
- Certaines fonctions admettent des arguments optionnels.

6.2.2 La valeur de retour

Enfin, la valeur de retour peut être de tout type valide en Python, par exemple un nombre, une chaîne de caractère, une liste, un tuple, un booléen ou encore `None`.

Certaines fonctions ont une valeur (par exemple `input()`)

Lorsque Python exécute un programme et rencontre l'instruction `input()`, il s'arrête et attend que l'utilisateur effectue une saisie au clavier. L'exécution du programme reprend une fois que l'utilisateur a appuyé sur ENTRÉE. La fonction `input` retourne alors ce qui a été saisi sous la forme d'une chaîne de caractères, que vous pouvez affecter à une variable pour l'utiliser ensuite.

```
print("Veuillez saisir votre nom:")
nom = input()
print("Veuillez saisir votre age:")
age = input()
```

Les deux variables ci-dessus sont donc de type `str`, ce sont des chaînes de caractères.

D'autres fonctions ne représentent pas une valeur (par exemple `print()`)

Certaines fonctions ne retournent pas de valeur, la fonction `print()` par exemple va afficher en sortie (à l'écran, ou dans un fichier) les arguments contenus dans les parenthèses, mais ne retourne rien. Plus précisément, les fonctions qui ne retournent rien retournent `None` qui est une valeur particluière qui est la seule de son type.

```
x = print("Veuillez saisir votre nom:")
print(x)                                    # affiche None
```

La section 6.B contient des astuces liés à l'utilisation de `input` et de `print`. La compréhension des sections suivantes ne nécessite pas que vous ayez lu cette annexe.

6.3 La définition de fonctions

Toutes les fonctions que nous avons vues jusqu'ici dans ce chapitre sont des fonctions dites *prédéfinies*, dans le sens où elles ont été définies dans le langage *Python3* (cf la liste à l'annexe 6.A) ou, comme `cos` et `random`, définies dans des *modules* (cf section 6.4).

6.3.1 Syntaxe générale

Mais vous pouvez aussi écrire vos propres fonctions. (Vous en écrirez beaucoup). Voici la syntaxe de définition d'une fonction à `N` arguments et retournant une valeur.

```
def <nom  fonction>(<N variables séparées par des virgules>):
          <instruction1>
          <instruction2>
          ...
          <instructionM>
          return <valeur_de_retour>
```

6.3.2 Vocabulaire : *en-tête, corps* et *paramètres*

L'instruction `def` indique que ce qui va suivre définit une fonction. Notez l'importance du `':'` à la fin de la ligne de définition ; cette ligne s'appelle **l'en-tête** de la fonction. Toutes les ins-

tructions qui suivent avec une indentation feront partie de la définition de la fonction, appelé **corps** de la fonction. Les `N` variables correspondant aux `N` arguments sont appelées des *paramètres*.

6.3.3 Quelques exemples

Voici, par exemple, la définition d'une fonction à un argument et retournant le résultat du calcul avec `return`.

```
def f(x):
        return x*x
```

Cette fonction comporte un seul paramètre : `x`. Après cette définition, lorsque l'interpréteur rencontrera l'expression `f(2)`, il appellera la fonction `f`, donnera au paramètre `x` la valeur 2. Enfin la valeur de l'expression `f(2)` sera évaluée comme le carré de `2`, c'est à dire 4.

```
  print(f(5))          # affiche 25
  print(3 + f(f(2)))   # affiche 19 car f(2) vaut 4 donc f(f(2)) vaut 16
```

Voici la définition d'une fonction à deux arguments et retournant le résultat du calcul avec `return`.

```
  def hypotenuse(x, y):
        return ((x ** 2 + y ** 2) ** (1/2))
```

`x` et `y` sont les paramètres de cette fonction. Voici enfin la définition d'une fonction à deux arguments mais qui ne retourne aucune valeur (ne fait qu'afficher le résultat).

```
  def printHypot(x, y):
        print((x ** 2 + y ** 2) ** (1/2))
```

6.3.4 L'ordre et le type des arguments d'une fonction

Lors de l'appel, le descripteur de la fonction doit être respecté, si on appelle la fonction `hypotenuse` avec un seul argument au lieu de deux, comme ceci :

```
def hypotenuse(x, y):
        return ((x ** 2 + y ** 2) ** (1 / 2))

print(hypotenuse(2))
```

On obtient le message d'erreur suivant :

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: hypotenuse() missing 1 required positional argument: 'y'
```

L'ordre des arguments n'a pas d'importance au moment de la définition de la fonction. En revanche, une fois la fonction définie, les appels à la fonction devront respecter l'ordre de ces arguments. Si on définit la fonction suivante :

```
def AfficheIdentite(monNom, monPrenom, monAge):
        print("Je m'appelle", monPrenom, monNom)
        print("et j'ai", monAge,"ans.")
```

Il faudra effectuer l'appel ainsi :

```
AfficheIdentite("Besson", "Luc", "59")
AfficheIdentite("Johansson", "Scarlett", "33")
```

Il en va de même du type des arguments de la fonction. Si on définit une fonction qui accepte en argument des entiers, il faut qu'au moment de l'appel, on lui fournisse des entiers dans le bon ordre. Cependant, en Python, la fonction n'est définie que par son nom, et pas par le type de ses arguments. Par conséquent, si on veut définir dans le même programme une seconde fonction avec le même nom que la première mais des types d'arguments différents, cela ne fonctionnera pas, car la seconde fonction définie écrasera la première.

6.3.5 Au sujet de l'instruction `return`

— On peut retourner le résultat du calcul de la fonction, qui peut être de n'importe quel type Python valide.

— Il est possible d'omettre l'instruction `return`. Dans ce cas, la fonction ne retournera aucune valeur (retournera `None`).

— Le fait de mettre plusieurs instructions `return` et/ou de les mettre ailleurs qu'à la fin du corps de la fonction ne constitue pas nécessairement une erreur. Mais lorsque Python exécute une fonction, il quitte la fonction dès qu'il rencontre le mot `return`. Il n'exécute pas des instructions placées après celle-ci.

— Enfin, l'instruction `return` est prioritaire sur l'opérateur ** (donc sur tous les autres opérateurs arithmétiques). Par exemple :

```
return (x ** 2 + y ** 2) ** (1 / 2)
```

n'est pas interprété comme :

```
return((x ** 2 + y ** 2) ** (1 / 2))
```

mais comme :

```
(return (x ** 2 + y ** 2)) ** (1 / 2)
```

Donc la partie `**(1/2)` de l'expression n'est pas prise en compte et la fonction retourne la valeur `x ** 2 + y ** 2`.

6.3.6 Fonctions et expressions

Pour Python, tout est objet. On peut donc affecter un fonction à une variable, puis utiliser cette variable pour appeler la fonction qu'elle représnte !

```
def f(x):
        return(x * 2)

double = f(5)           # double est un nombre
print(double)           # affiche 10

double = f              # double est une fonction
print(double(5))        # affiche 10
```

De même, une fonction peut être en argument d'une autre fonction ; et une fonction peut retourner une autre fonction.

6.3.7 La portée des variables (locales ou globales)

La *portée* d'une variable désigne la zone de programme dans laquelle une variable est utilisable. En particulier, une variable définie en dehors de toute fonction peut être utilisée à n'importe quel endroit du programme. On l'appelle une variable *globale*. Les variables définies dans une fonction, y compris les paramètres de cette fonction, ne peuvent être accédées qu'à l'intérieur de la fonction et ne peuvent en aucun cas modifier des variables extérieures. On les appelle des variables *locales*.

Les paramètres sont des copies

À l'appel d'une fonction, la valeur des arguments est copiée dans les paramètres de la fonction. Ainsi les paramètres sont des copies. Le fait de modifier ces copies ne modifie en aucun cas l'original.

```
def f(y):
        y = y * 2        # y est une copie de la valeur de l'argument

x=3                      # Programme principal qui appelle la fonction
f(x)                     # La valeur de x est copiee dans le parametre y
                         # et multiple cette copie par 2.
print(x)                 # Affiche 3. Autrement dit, le fait de doubler
                         # la copie (y) ne modifie l'original (x)
```

Le nom des arguments et paramètres

Dans l'exemple précédent, l'argument et le paramètre ne portaient pas le même nom (respectivement `x` et `y`). Il est tout a fait possible d'appeler une variable `x` dans le programme principal et d'appeler une variable `x` dans la fonction. Cela ne modifie en rien le fonctionnement.

```
def f(x):
        x = x * 2

x=3
f(x)
print(x)                 # Affiche toujours 3.
```

La variable `x` définie dans la fonction n'est pas la même variable que celle qui est définie à l'extérieur. Python réservera une place en mémoire pour la variable du programme principal, et une autre place pour la variable de la fonction. La fonction va donc travailler sur une copie de la variable sans écraser celle-ci.

Pour qu'une fonction puisse modifier une variable globale du programme principal, il faut lui dire de ne pas travailler sur une copie de la variable, mais sur l'emplacement en mémoire de la variable globale. Ce qui donnerait :

```
def f():                # la fonction n'a plus besoin de l'argument,
      global x          # x est globale
      x = x * 2         # la fonction travaille sur la variable globale x

x = 3
f()                     # L'appel a f modifie la variable globale x
print(x)                # affiche 6
```

Nous parlons ici du mot-clef `global` pour que vous sachiez qu'il existe. Mais, dans le cadre de ce cours, je vous demande de ne jamais utiliser ce mot-clef autrement qu'à des fins d'expérimentation. Nous verrons pourquoi au chapitre suivant.

Lorsque l'argument est une liste

C'est un peu différent dans le cas des listes, en effet, les listes ne sont pas des variables ordinaires, mais sont des références à des endroits en mémoire où sont stockées les valeurs dans la liste. Ainsi, lorsqu'on modifie une liste, on modifie le contenu de la mémoire où elle est stockée, et non pas la variable elle-même. Regardons l'exemple suivant :

```
def modifiePremier (lst):
        lst[0] = 25

a = [4, 0, 0]
print("Avant : a= ",a)
modifiePremier(a)
print("Apres : a= ", a)
```

Au moment de l'appel de la fonction `modifiePremier`, la valeur de `a` est copiée dans `lst`. Mais `a` est une liste. Elle indique donc l'endroit dans la mémoire où est écrite la liste. Par conséquent, `lst` indique à son tour l'endroit dans la mémoire où est écrite la liste `a`.

La fonction `modifiePremier` fait son travail : elle affecte la valeur 25 au premier élément de `lst`. Quand on sort de la fonction, la variable `a` n'a pas changé. En revanche, le contenu stocké en mémoire, lui, a changé.

```
Avant : a=  [4, 0, 0]
AprÃ¨s : a=  [25, 0, 0]
```

6.3.8 Documentation d'une fonction

Il est important aussi de documenter votre fonction. Pour cela, il est conseillé de rédiger une description pour chaque fonction. Cette description (docstring) doit figurer entre triple quote :

```
def f(x):
        """
                x : un nombre (entier ou flottant)
                valeur de retour : entier ou flottant
                La valeur de retour est le carrÃ© de x.
        """
        return x * x
```

Le docstring doit contenir les informations nécessaires et rapidement lisibles pour le bon usage de la fonction : Le nom et le type de chaque paramètre et ce qu'il représente, le type de la valeur de retour, ainsi que la relation entre la valeur de retour et les paramètres. Ces informations constituent la *spécification* de la fonction.

```
>>> help(f)
Help on function f in module __main__:
f(x)
                x : un nombre (entier ou flottant)
                valeur de retour : entier ou flottant
                La valeur de retour est le carrÃ© de x.
```

Remarquer que ce que renvoie la fonction `help()` est ce qui a été écrit entre les triples quotes.

6.4 Les modules

6.4.1 Qu'est-ce qu'un module ?

Un module est un fichier dans lequel on a placé des fonctions et des variables ayant un rapport entre elles. Un module doit avoir une extension `.py`. Un exemple de module fourni automatiquement avec Python est le module `math` qui contient des fonctions mathématiques.

6.4.2 Utiliser un module

Pour utiliser un module il faut importer le module dans le programme. La manière la plus simple est d'utiliser la commande suivante :

```
import math
```

De cette manière toutes les variables et toutes les fonctions du module `math` sont disponibles dans le programme. On peut ensuite utiliser les fonctions contenues dans le module en les appelant de la manière suivante : par exemple `sqrt` [1].

```
import math
a = math.sqrt(4)    # l'instruction a = sqrt(4) est ici equivalente
print(a)            # car on a importe tout le module math
```

La variable `a` contient le résultat de $\sqrt{4} = 2$, l'instruction `print` affichera `2` à l'écran. On peut donner un alias à un module : on changera le préfixe lors de l'appel à la fonction en conséquence.

```
import math as m
a = m.sqrt(4)
print(a)
```

6.4.3 Utiliser plusieurs modules

On peut importer plusieurs modules. Les fonctions de chaque modules seront alors différenciées par le préfixe adéquate. Dans l'exemple ci-dessous, Python ne confond pas la fonction `sqrt` du module `math` avec celle du module `numpy`.

1. square root

```
import math, numpy
a = math.sqrt(4)
b = numpy.sqrt(9)
print(a, b)
```

6.4.4 N'utiliser que partiellement un module

Il existe une autre façon d'importer un module. La commande suivante importe le module en entier.

```
from math import *
a = sqrt(4)
```

Il faut cependant se méfier de cette manière d'importer les modules, lorsque plusieurs modules contiennent des variables ou des fonctions de même nom, seules celles du dernier module importé sont utilisées dans le programme si on oublie le préfixe. En conséquence il est possible de n'importer qu'une fonction ou qu'une variable d'un module en remplaçant l'étoile par le nom de l'élément qu'on souhaite importer. Par exemple pour n'importer que la fonction `sqrt` du module `math` on utilise la ligne de commande suivante :

```
import numpy
from math import sqrt
nombre = float(input("Valeur du nombre ?"))
print("la racine carree de ", nombre, "est", sqrt(nombre))
```

Python utilise ici la fonction `sqrt` du module `math` qui a remplacé celle du module `numpy`, si aucun préfixe n'est utilisé.

6.4.5 Contenu d'un module

Pour connaître les fonctions, les variables et obtenir des explications sur le contenu d'un module il faut utiliser la fonction `help()` , ci-dessous le résultat de la commande `help(math)`

```
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

    acosh(...)
        acosh(x)
```

```
        Return the inverse hyperbolic cosine of x.

        [...]
```

Il est également possible d'avoir les informations sur un seul des éléments du module :

```
>>> help(math.sqrt)

Help on built-in function sqrt in module math:

sqrt(...)
    sqrt(x)

    Return the square root of x.
```

6.4.6 Créer un module

Un module est ni plus ni moins qu'un fichier qui a pour extension `.py` comme tous les fichiers Python. Il contient des définitions de fonctions et de variables. L'encadré ci-dessous montre le contenu d'un module `myModule.py` dans lequel sont définies deux fonctions :

— une fonction qui détermine si une année est bissextile ;
— une deuxième fonction qui calcule l'aire d'un rectangle.

```
"""
     myModule contient deux fonctions:
                     anneeBissextile et aireRectangle
"""
def anneeBissextile(annee):
     """ Fonction qui retourne True si l'annee est bissextile,
             et False sinon """
     if annee % 400 == 0 or (annee % 4 == 0 and annee % 100 != 0):
             return True
     else:
             return False

def aireRectangle(largr, longr):
     """ Fonction qui retourne l'aire d'un rectangle """
     return largr * longr
```

6.4.7 Docstrings

La chaîne de caractère comprise entre les triples quotes est un `docstring`, une chaîne d'aide ou de documentation. Pour des raisons de lisibilité tout le texte d'aide est indenté de la même manière que le reste du code. Le contenu du `docstring` apparaît lorsqu'on utilise la fonction `help` sur le module ou sur l'élément concerné par le `docstring`. L'encadré ci-dessous donne le résultat de la fonction `help()` appliqué au module `myModule`.

```
Help on module myModule:

NAME
    myModule

DESCRIPTION
    myModule contient deux fonctions:
                    anneeBissextile et aireRectangle

FUNCTIONS
    aireRectangle(largr, longr)
        Fonction qui retourne l'aire d'un rectangle

    anneeBissextile(annee)
        Fonction qui retourne True si l'annee est bissextile,
        et False sinon
```

Noter que le résultat est formaté quand il s'agit d'un module, et qu'on y retrouve le texte qui avait été écrit dans `myModule` inséré dans le format.

6.4.8 Tester son module directement dans... le module

Il est possible de tester le module en l'exécutant directement, sans avoir à passer par un deuxième fichier de test. Pour que le test s'exécute lorsque le fichier est lancé mais pas lors de l'importation du module on peut utiliser la variable `__name__` . C'est une variable qui existe dès le lancement de l'interpréteur. Si elle vaut `__main__` c'est que le fichier appelé est le fichier exécuté. Il suffit donc de soumettre la commande de test à la condition `__name__==__main__`, ainsi elle ne s'exécutera que si le fichier appelé est le fichier exécuté.

```
"""
        myModule contient deux fonctions:
                        anneeBissextile et aireRectangle
"""
def anneeBissextile(annee):
        """ Fonction qui retourne True si l'annee est bissextile,
                et False sinon """
        if annee % 400 == 0 or (annee % 4 == 0 and annee % 100 != 0):
                return True
        else:
                return False

def aireRectangle(largr, longr):
        """ Fonction qui retourne l'aire d'un rectangle """
        return largr * longr

# Auto test du module (fonction anneeBissextile())
if __name__ == "__main__":
        print("procedure de test du module")
        print("test pour une annee bissextile (2000) et l'autre non (2100)")
        print("2000: ", anneeBissextile(2000))
```

```
        print("2100: ", anneeBissextile(2100))
```

Cela donne :

```
\$ python3 myModule.py

procédure de test du module
test pour une année bissextile (2000) et l'autre non (2100)
2000:  True
2100:  False
```

6.5 Les Packages : qu'est-ce qu'un package

Un package[2] est un répertoire qui contient des modules ou des sous-répertoires de modules. Cela permet d'organiser les fichiers nécessaires pour le programme. Nous allons utiliser l'exemple d'un package `myPackage` qui contient deux modules `anneeBissextile.py` et `aireRectangle.py`.

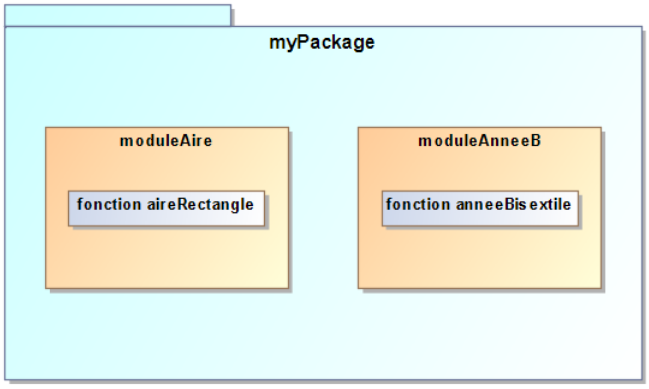


FIGURE 6.1 – Constitution de `myPackage`

Le dossier `myPackage` doit se trouver dans le même dossier que le fichier python du programme principal. Les commandes ci-dessous montrent comment importer dans le fichier principal les modules du package. Dans le programme principal, il faut mettre :

```
from myPackage import *
```

Dans le répertoire `myPackage` il faut créer un fichier `__init__.py` qui contient les informations suivantes :

```
from .moduleAire.aireRectangle import *
from .moduleAnneeB.anneeBissextile import *
```

Notez que le dernier suffixe pointe sur le fichier `.py` qui contient la définition de la fonction. La hiérarchie des modules et répertoires est illustrée à la figure 6.2.

2. Paquetage en français

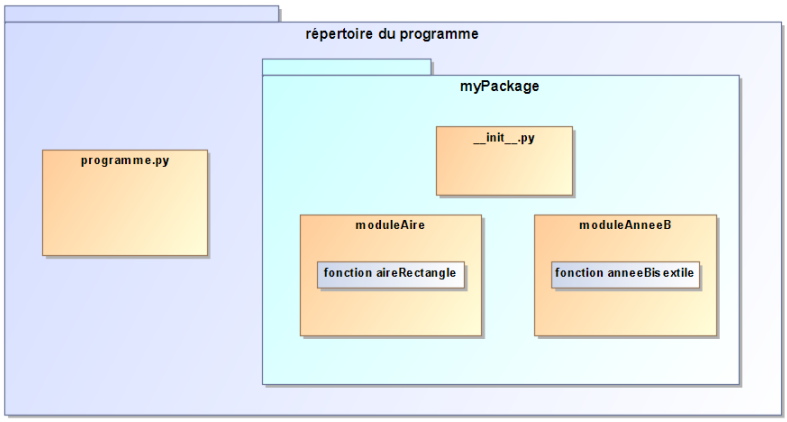


FIGURE 6.2 – Fichier exécutable et package appelé

Pour exécuter du code avant le programme principal, on peut aussi mettre les instructions dans un fichier `__init__.py`.

6.6 Pour aller plus loin

Dans le fichier principal et dans les modules importés on peut faire figurer :

— `#!/bin/env python3` : s'il est nécessaire d'indiquer à l'interpréteur quelle version de python utiliser ;

— `import os` : permet d'importer le module `os` qui fournit les variables et les fonctions nécessaires pour que le programme dialogue avec le système d'exploitation. On y trouvera, entre autres, les fonctions `open, close, write, read` pour manipuler des fichiers.
Pour en savoir plus :

```
\$python3
>>> import os
>>> help(os)
>>> help(os.open)
>>> help(os.write)
>>> help(os.read)
>>> help(os.close)
```

— On pourra importer une bibliothèque graphique (tkinter, pygame, etc)

— La liste des modules disponibles pour Python est consultable ici :
`https://docs.python.org/3/py-modindex.html`

Médiagraphie

— https ://vbaforexcel.wordpress.com/2015/05/19/affichage-fonction-print/

— Le petit Python, Richard Gomez, Ellipse.

— Informatique et Science du Numérique, Edition Python, Gilles Dowek et al. Eyrolles.

Appendix

6.A Les 72 fonctions prédéfinies dans *Python3*

abs all any ascii	bin bool bytearray bytes	callable chr classmethod compile complex copyright credits	delattr dict dir divmod	enumerate eval exec exit	filter float format frozenset	getattr globals
hasattr hash help hex	id input int isinstance issubclass iter	len licence list locals	map max memoryview min	next	object oct open ord	pow print property
quit	range repr reversed round	set setattr slice sorted staticmethod str sum super	tuple type	vars	zip	

6.B Pour aller plus loin : astuces en lien avec `input` et `print`

6.B.1 Saisie d'informations numériques

Si, lors de l'utilisation dans le programme, on a besoin d'autres types de données, il faudra effectuer une conversion :

```
print("Veuillez saisir votre nom:")
nom = input()
print("Veuillez saisir votre age:")
```

```
    age = int(input())
    print("Veuillez saisir la valeur de pi avec 8 dÃ©cimales:")
    pi = float(input())
```

ainsi, `nom` sera de type `str`, tandis que `age` sera de type `int` et `pi` sera de type `float`.

6.B.2 Affichage d'informations numériques

On peut effectuer des manipulations de concaténation de chaînes de caractère avec des variables, et appliquer des formats à la fonction `print()` :

6.B.2.1 En utilisant des arguments multiples pour `print`

La fonction `print` peut admettre plusieurs arguments. Regardons le code suivant :

```
    monNom="Pierre"

    print("Je m'appelle",monNom)
    print(monNom,"est parti.")
    print("Je m'appelle",monNom,"et j'habite a Paris.")

    maSlt="Bonsoir,"
    maVille="Lyon"
    monAge=25

    print(maSlt,"je m'appelle",monNom,", j'habite",maVille,"et j'ai"\
    ,monAge,"ans.")
```

qui donne comme résultat :

```
Je m'appelle Pierre
Pierre est parti.
Je m'appelle Pierre et j'habite a Paris.
Bonsoir, je m'appelle Pierre , j'habite Lyon et j'ai 25 ans.
```

6.B.2.2 Par concaténation avec l'opérateur +

Deux arguments successifs (séparés par une virgule dans la liste des arguments) sont automatiquement séparés par un espace lors de l'affichage.
Remarque : un fichier Python doit contenir 80 colonnes, quand une instruction dépasse les 80 colonnes on doit utiliser le caractère \ qui signifie que la ligne suivante est la suite de la ligne en cours.
Comparons avec celui-ci :

```
  monNom="Pierre"

  print("Je m'appelle"+monNom)
  print(monNom+" est parti.")
  print("Je m'appelle"+monNom+"et j'habite Ã  Paris.")

  maSlt="Bonsoir,"
```

```
maVille="Lyon"
monAge=25

print(maSlt+" je m'appelle "+monNom+", j'habite "+maVille+" et j'ai "\
+str(monAge)+" ans.")
```

qui donne comme résultat :

```
Je m'appellePierre
Pierre est parti.
Je m'appellePierreet j'habite a Paris.
Bonsoir, je m'appelle Pierre, j'habite Lyon et j'ai 25 ans.
```

- Le signe plus ne crée pas automatiquement un espace entre le texte et la valeur de la variable, comme nous le montrent les exemples ci-dessus ; il faut, si nécessaire, ajouter l'espace dans le texte qui précède ou qui suit la variable.
- La variable `monAge`, dans le dernier exemple, est incluse dans la fonction `str()` : il existe plusieurs types de variables (string pour les chaînes de caractère, integer pour les nombres entiers, float pour les nombres décimaux etc...). Ici, il faut considérer que le signe plus « additionne » des données, or Python n'additionne que des données du même type. `str()` convertit le nombre entier 25 en une chaîne alphanumérique de type texte, autrement dit 25 devient "25" et l'addition de chaîne de caractère devient ainsi possible.
- Il est possible d'utiliser les 2 types de signes dans la même fonction `print()` pour concaténer (réunir) textes et variables, comme le montre l'exemple ci-dessous :

```
monObjet ="voiture"
print("La",monObjet+" est rouge.")
La voiture est rouge.
```

6.B.2.3 fonction `format()`

Cette fonction bien pratique s'affranchit des contraintes des guillemets et des signes de concaténation. La fonction `format()` est introduite par un point juste après le guillemet de fin et encadre l'ensemble des variables utilisées dans le texte. Chaque variable est intégrée au texte par des accolades et le numéro de l'accolade correspond à l'ordre dans lequel la variable apparaît dans la fonction `format()`.

```
print("{0} a eu {1} ans. On a fÃªtÃ© les {1} ans \
       de {0}." .format("Pierre", 25))
Pierre a eu 25 ans. On a fÃªtÃ© les 25 ans de Pierre.
```

Remarque : Le numéro de la première accolade est toujours 0 !

6.B.2.4 concaténation avec l'opérateur %

Proche de la fonction `format()`, cette méthode introduit chaque variable dans le texte par le signe % suivi d'une lettre indiquant le type de la variable :

- %s : chaîne de caractère,
- %d : nombre entier,
- %f : nombre décimal,

— %.Xf : X désignant un nombre entier indiquant le nombre de décimales après la virgule.

Le signe % derrière le guillemet de fermeture introduit la valeur de chaque variable dans l'ordre d'apparition respectif dans le texte.

```
monA = 12
monB = 35.6
monC = "abcd"
monD = 7.8251896
print("A=%d, B=%f, D=%.3f, C=%s et Z=%f // A=%d " \
        %(monA,monB,monD,monC,483,monA))
A=12, B=35.600000, D=7.825, C=abcd et Z=483.000000 // A=12
```

Affichage d'une liste :

```
>>> maListe=[1, 3, 5, 7, 9]
>>> print("ma Liste : %s " %(maListe))
ma Liste : [1, 3, 5, 7, 9]
```

Chapitre 7

Méthodologie de test et de debug

En construction.

Chapitre 8

Récursivité

8.1 Principe de la récursivité

De manière générale, un algorithme récursif qui résout un problème, utilise pour le résoudre la solution du même problème *plus petit*, jusqu'à arriver à un problème de taille suffisamment petite pour présenter une solution évidente.

8.1.1 Exemple (récursif)

Imaginons que notre but est de trier un paquet de cartes.

— trier un paquet contenant une seule carte est évident : il est déjà trié (!) ;
— pour trier n cartes : il suffit de trier $(n-1)$ cartes d'abord, puis d'insérer la $n^{\text{ième}}$ en bonne position ; c'est facile, il suffit de comparer sa valeur aux cartes déjà triées dans le paquet pour l'insérer en bonne position.

Donc, (1) je sais trier une carte, et (2) si je sais trier $(n-1)$ cartes, alors je sais en trier n. J'en conclus que je sais trier un nombre quelconque de cartes. De plus, j'ai une méthode pour le faire, décrite par l'*algorithme* décrit ci-dessus.

8.1.2 Exemple (itératif)

Il existe très souvent un algorithme *itératif* équivalent à l'algorithme récursif, mais qui fonctionne avec un raisonnement inverse. Dans cet exemple, pour trier n cartes :

— je commence par en prendre une,
— puis j'insère la deuxième avant ou après selon que sa valeur est inférieure ou supérieure à la première ;
— puis j'insère la troisième en bonne position, selon qu'elle est supérieure ou inférieure à la première et à la deuxième ;
— puis j'insère la quatrième...
— ...
— jusqu'à arriver à la $n^{\text{ième}}$.

Finalement, le jeu de cartes est trié.

8.1.3 Récursif ou itératif ?

L'intérêt des algorithmes récursifs en informatique (et des preuves par récurrence en mathématiques, qui fonctionnent sur le même principe) est la possibilité d'écrire facilement de manière complète, ce que j'ai décrit dans l'exemple ci-dessus par « ... ». Dans certains cas, il est difficile de décrire complètement ce que j'ai caché dans l'algorithme itératif par cet artifice. C'est dans ces cas que l'algorithme récursif sera préféré à l'algorithme itératif, plus complexe à écrire.

Dans d'autres cas, les versions itérative – avec des boucles `for` ou `while` – et récursive de l'algorithme sont équivalentes. C'est alors au programmeur de choisir quelle version il préfère, ce que le programmeur expérimenté fera souvent pour des raisons de meilleure lisibilité de son code ou de meilleures performances. Il existe des cas où le code récursif est plus lisible mais moins performant que le code itératif – mais ce n'est pas systématique.

8.1.4 Fonctionnement général

Tout algorithme récursif utilise les deux mécanismes suivants :

— la *condition d'arrêt* est la résolution du problème de petite taille, souvent très facile à résoudre ;

— le *cas général* est calculé en utilisant la solution du *même* problème de taille plus petite. On peut trouver cette solution grâce à cet algorithme lui-même !

8.2 Fonctions récursives en Python

Pour implémenter un algorithme récursif, nous utiliserons une fonction, dont le code va appeler cette fonction elle-même pour résoudre le même problème plus petit.

Pour que l'enchaînement d'appels de fonction s'arrête, il ne faut plus appeler la fonction dès lors qu'on sait résoudre le problème de petite taille, sinon le programme ne s'arrête jamais (de lui-même) ! Et bien évidemment, il faut faire cela en premier, avant d'exécuter l'appel récursif à la fonction.

8.2.1 Structure

La structure générale que prend une fonction récursive est habituellement :

```
def fonction_rec(...):
  if condition_d_arret:
    ...
    return ...
  else:
    ... fonction_rec(...) ...
    return ...
```

Attention, si votre fonction retourne une valeur, elle doit retourner une valeur *dans tous les cas* (les deux branches du `if` ici) : si la fonction retourne une valeur dans un cas et pas dans l'autre, elle est mal écrite ! Bien évidemment, vous pouvez décider de retourner une valeur après l'exécution des deux branches du `if`, en plaçant le `return` tout à la fin.

Le nombre d'appels récursifs à une fonction est limité à 1000 par défaut en python (voir `sys.getrecursionlimit()` du module `sys`). Si ce nombre est atteint, le programme python s'arrête en affichant le message d'erreur "`RecursionError: maximum recursion depth exceeded`". Ce paramètre peut être modifié, avec prudence : si vous écrivez une fonction récursive dont la condition d'arrêt est incorrecte, votre programme risque de ne pas s'arrêter (avant un temps très long) !

Voici un exemple simple de fonction récursive. La fonction suivante calcule et renvoie la somme des n premiers nombres entiers ($n \geq 0$) : $somme(n) = 1+2+...+n$. Ainsi, $somme(36)$ est égal à 666. On peut aussi en donner la définition récurrente suivante :

$$somme(n) = \begin{cases} 0 & \text{si } n = 0 \\ n + somme(n-1) & \text{sinon} \end{cases}$$

Cette définition se traduit directement en Python par :

```
# renvoie la somme des n premiers entiers
def somme(n):
  if n == 0:   # condition d'arrêt
    return 0
  else:        # cas général, appel récursif
    return n + somme(n-1)
```

On peut utiliser cette fonction comme toute fonction bien définie en Python (qu'elle soit récursive ou non), par exemple :

```
>>> print(somme(36))
666
>>> for i in range(10):
...   print(somme(i))
...
0
1
3
6
10
15
21
28
36
45
```

8.2.2 Exemple (suite)

Voici ci-dessous un exemple de code qui implémente un algorithme de tri de cartes récursif comme vu en introduction de ce chapitre. Nous supposerons que les cartes sont stockées dans des listes, et que nous disposons d'une fonction d'insertion dans une liste triée de cartes appelée `inserer(carte, liste)`. Cette fonction renvoie une liste dans laquelle la carte est insérée

en bonne position. Nous ne nous occuperons pas d'écrire cette fonction ici, vous l'écrirez plus tard, lorsque nous aborderons le problème des tris (Chapitre **??**).

```
def trier (cartes):
  if len(cartes) <= 1:  # condition d'arrêt
    # retourne la solution : la liste de cartes
    return cartes
  else:                 # cas général
    # trier la fin du paquet
    resultat_intermediaire = trier(cartes[1:])
    # insérer la première carte du paquet dans la liste triée
    solution = inserer(cartes[0],resultat_intermediaire)
    # retourne la solution qui vient d'être calculée
    return solution
```

On peut même écrire, de manière plus compacte :

```
def trier (cartes):
  if len(cartes) <= 1:
    return cartes
  else:
    return inserer(cartes[0], trier(cartes[1:]))
```

Voici un exemple d'utilisation de cette fonction où la liste `l` est initialisée puis affichée, la liste triée `l_tri` est calculée à l'aide de cette fonction de tri, puis affichée :

```
>>> l=[68, 97, 23, 14, 53]
>>> print(l)
[68, 97, 23, 14, 53]
>>> l_tri = trier(l)
>>> print(l_tri)
[14, 23, 53, 68, 97]
```

8.2.3 Déroulement

Nous allons voir de quelle manière s'exécute la fonction récursive lorsqu'elle est appelée.

Le premier appel à la fonction de tri se fait avec la liste initiale : `trier([68, 97, 23, 14, 53])`. Analysons l'exécution de ce premier appel à la fonction.

- Le contenu de la variable `cartes` est `[68, 97, 23, 14, 53]`.
- La longueur de la liste est 5. Donc, le premier test renvoie faux, et l'exécution passe au `else:`.
- Puis, la fonction trier() est appelée, avec comme argument la liste `cartes[1:]`, c'est à dire `[97, 23, 14, 53]`. Cet appel va trier cette liste, et donc calculer `[14, 23, 53, 97]`.
- Ensuite, l'entier qui se trouve dans cartes[0], c'est à dire 68, est inséré dans cette liste ordonnée, ce qui va nous donner la liste `[14, 23, 53,` **`68`**`, 97]`, qui est le résultat !
- finalement, la fonction renvoie ce résultat, le tableau est trié.

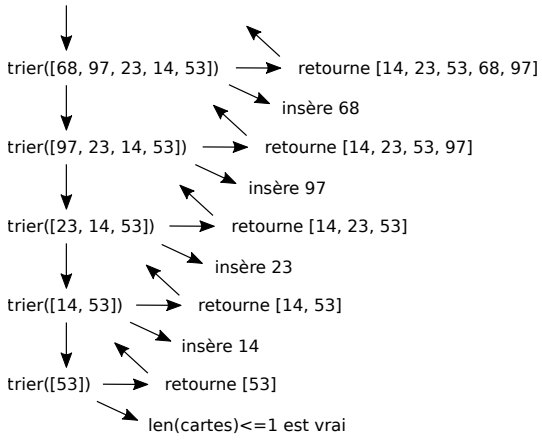


FIGURE 8.1 – Déroulement de l'appel récursif `trier([68, 97, 23, 14, 53])`.

Cet algorithme récursif est entièrement déroulé Figure 8.1, en représentant tous les appels à la fonction `trier()` et leurs arguments. Chaque fonction s'exécute en commençant par la flèche du bas, puis en remontant vers la droite. Vous remarquerez qu'il y a cinq appels à la fonction récursive, autant qu'il y a d'éléments dans le tableau. Si nous avions trié un tableau contenant 1000 éléments, la fonction récursive aurait été appelée 1000 fois.

8.3 Objet de la récursion

La récursivité peut porter sur différents types de variables, nous en explicitons quelques-uns ci-dessous.

8.3.1 Récursivité portant sur des nombres entiers

Une fonction récursive sur des nombres entiers est équivalente à l'exécution d'une boucle parcourant ces nombres. En général, la fonction récursive prendra comme argument le nombre d'itérations restant à effectuer. Ainsi, on exprimera `f(n)` en fonction de `f(n-1)`, et la condition d'arrêt sera habituellement le test `n==0` (ou `n==1`).

Si on voulait donner au compteur `n` la valeur de l'itération dans l'ordre naturel, il faudrait passer un argument supplémentaire indiquant la valeur maximale à atteindre, pour écrire une condition d'arrêt de la forme `n>max`. La fonction se trouve complexifiée avec ce paramètre supplémentaire : elle prendrait la forme `f(n,max)`.

Selon que la fonction récursive est appelée *avant* ou *après* une autre partie de code, cette partie de code est exécuté dans l'ordre décroissant des itérations ou dans l'ordre naturel en revenant des appels récursifs. C'est ce qui est illustré dans l'exemple suivant :

```
def f(n):
  if n==0:
    return
  else:
    print("->", n)         # avant l'appel récursif
```

```
    f(n-1)
    print("  ", n, "<-")  # après l'appel récursif
    return
```

L'exécution de `f(5)` produira la sortie suivante, où les itérations sont parcourues dans l'ordre décroissant *avant* l'appel récursif, et dans l'ordre naturel *après* l'appel récursif :

```
>>> f(5)
-> 5
-> 4
-> 3
-> 2
-> 1
   1 <-
   2 <-
   3 <-
   4 <-
   5 <-
```

Remarquez que cette fonction ne fait rien dans le test de la condition d'arrêt. On peut écrire une telle fonction récursive avec une condition d'arrêt implicite, en testant seulement l'inverse de la condition d'arrêt pour traiter le cas général. Le code résultat est plus court et sans être moins lisible, et cette fonction récursive est parfaitement équivalente à la précédente :

```
def f(n):
  if n!=0:
    print("->", n)         # avant l'appel récursif
    f(n-1)
    print("  ", n, "<-")  # après l'appel récursif
```

8.3.2 Exemple : la factorielle

Le calcul de la factorielle est donné couramment pour illustrer la récursivité, bien que ce soit un exemple purement didactique : cet algorithme pourrait être écrit très facilement de manière itérative (avec une boucle).

La factorielle d'un nombre $n \geq 0$ est donnée par la formule :

$$n! = 1 * 2 * 3 * ... * (n-2) * (n-1) * n$$

La définition récurrente équivalente [1] est :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n * (n-1)! & \text{sinon} \end{cases}$$

D'où nous pouvons déduire le code python :

1. Vous remarquerez peut-être que cette notation est plus complète : elle définit bien la factorielle du nombre 0, qui a la valeur 1 selon sa définition mathématique ; la définition précédente est ambiguë sur la valeur de 0!.

```
def fact(n):
  if n==0:
    return 1
  else:
    return n * fact(n-1)
```

8.3.3 Récursivité portant sur des listes

Il est très courant de parcourir des listes d'éléments (de type quelconque) pour appliquer un traitement sur chacun des éléments. Cela s'effectue de manière très naturelle à l'aide d'un algorithme récursif, qui traite le premier élément (la *tête*) de la liste, puis qui appelle la même fonction pour traiter les éléments suivants (la *queue*), jusqu'à épuisement de la liste.

En python, voici le squelette de fonction récursive typique traitant tous les éléments d'une liste :

```
def traite_liste(l):
  if l==[]:
    return ... # résultat pour le traitement de la liste vide
  else:
    # utilise la tête l[0] pour calculer quelque chose
    r = traite_liste(l[1:])  # traite toute la queue de la liste
    # post-traite le résultat, en utilisant éventuellement l[0]
    return ... # retourne le résultat
```

L'exemple du tri donné plus haut illustre ce type de traitement récursif d'une liste. Dans cet exemple cependant, la récursion s'arrêtait une étape plus tôt, lorsque la liste n'avait plus qu'un seul élément.

Il est souvent plus aisé de raisonner de cette manière que de manière itérative : on écrit une fonction qui traite un élément après l'autre sans avoir à considérer la succession dans une boucle des traitements de tous les éléments et le stockage intermédiaire du résultat qu'on veut calculer.

8.3.4 Définition mathématique et fonction récursive

Vous avez probablement déjà vu des définitions mathématiques récurrentes (la définition de la factorielle donnée précédemment en était une), et peut-être même fait des preuves par récurrence.

La transformation d'une telle récurrence en une fonction récursive implémentant l'algorithme en python est quasi immédiate : il suffit de choisir le nom qu'on donne à la fonction, et de traduire ce qui se trouve à droite de l'accolade en python !

Par exemple, la fonction d'Ackermann-Péter est définie mathématiquement par :

$$A(m,n) = \begin{cases} n+1 & \text{si } m = 0 \\ A(m-1,1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m-1, A(m,n-1)) & \text{si } m > 0 \text{ et } n > 0 \end{cases}$$

Quelle que soit la complexité de cette fonction, il est facile de la traduire en python en remplaçant les "si" par des tests, et les valeurs par des `return` dans la fonction :

```
def ackermann(m,n):
  if m==0:
    return n+1
  elif m>0 and n==0:
    return ackermann(m-1,1)
  elif m>0 and n>0:
    return ackermann(m-1,ackermann(m,n-1))
```

A titre de remarque, cette fonction est assez particulière : bien qu'elle ne contienne qu'une addition avec 1, elle croît très rapidement. Par exemple, ackermann(4,2) est un nombre contenant 19 729 chiffres [wikipedia], bien plus que le nombre d'atomes dans l'univers !

8.4 Quelques exemples plus complexes

8.4.1 Récursivité multiple

Il est parfois utile d'appeler une fonction récursive plusieurs fois à l'intérieur de son code. C'est ce que nous avons fait dans l'exemple mathématique précédent.

Un exemple typique en informatique est l'implémentation des algorithmes de type "*diviser pour régner*", qui consistent à couper un problème en deux parties (ou plus), puis à résoudre chaque partie indépendamment, avant de trouver la solution globale du problème en rassemblant les deux solutions partielles. Les algorithmes de recherche par dichotomie (voir exercices) fonctionnent selon le même principe général : couper un problème en deux pour diminuer la complexité de l'algorithme.

8.4.2 Exemple

Il existe des algorithmes de tri beaucoup plus efficaces que celui que nous avons vu précédemment dans la Section 8.2. Pour trier une liste, plutôt que d'insérer un élément après l'autre dans une liste triée, nous pourrions couper la liste en deux, trier chacune des deux parties, puis les rassembler en effectuant ce qui s'appelle un *interclassement*.

L'interclassement consiste à parcourir les deux parties triées, et à choisir le plus petit élément des deux pour l'insérer à la fin d'une liste `lt` (vide initialement), et ainsi de suite jusqu'à épuisement des deux listes. La liste `lt` construite de cette manière est triée.

Voici le code Python correspondant à cet algorithme récursif, en supposant que nous disposons d'une fonction d'interclassement de deux listes qui renvoie une liste interclassée (nous verrons cette fonction dans le chapitre **??**) :

```
def trier2 (cartes):
  n = len(cartes)
  if n <= 1:
    return cartes
  else:
    l1 = trier(cartes[:n//2])  # trier la première moitié
    l2 = trier(cartes[n//2:])  # trier la deuxième moitié
    return interclassement(l1,l2)
```

8.4.3 Fonction *wrapper*

Une fonction *wrapper* est une fonction qui est appelée directement, et qui ne fait pas la récursion elle-même, mais qui appelle une autre fonction récursive. Il est utile de déclarer une telle fonction si la fonction récursive prend des paramètres supplémentaires qu'on ne veut pas exposer à l'utilisateur de la fonction principale, ou si des tests de validité des paramètres doivent être faits (une seule fois, et non pas à chaque appel récursif !), ou encore si des initialisations particulières sont nécessaires avant les appels récursifs.

Un exemple typique est le calcul de la suite de Fibonacci, donc la formule générale est :

$$F(n) = \begin{cases} 1 & \text{si } n = 0 \text{ ou } n = 1 \\ F(n-1) + F(n-2) & \text{sinon} \end{cases}$$

Une implémentation naïve de cette définition serait d'appeler la fonction récursive deux fois à chaque exécution. Mais cela revient à calculer plus de fois que nécessaire chaque terme, puisqu'on pourrait réutiliser les deux derniers termes calculés pour trouver le nouveau !

Il existe plusieurs variantes d'implémentation de cet algorithme. La fonction récursive peut renvoyer un couple d'entiers qui sont les deux dernières valeurs de la suite. On peut aussi effectuer le calcul dans le sens inverse, en passant en argument de la fonction récursive les deux dernières valeurs calculées, et les valeurs 0 et 1 au départ.

Dans la première variante, on définit une fonction qui renvoie le couple des nombres $(F(n-1), F(n))$, et un *wrapper* qui retourne le deuxième :

```
# fonction récursive :
def fibo_couple(n):
  ...
  return (a,b)
# wrapper :
def fibo1(n):
  a,b = fibo_couple(n)
  return b
```

Ainsi, l'utilisateur appelle la fonction `fibo1()`, et n'a pas à se préoccuper du fait que la fonction récursive renvoie un couple (et non pas un entier).

Dans la deuxième variante, on définit une fonction récursive qui prend deux arguments supplémentaires : les valeurs des deux termes précédemment calculés de la suite.

```
# fonction récursive :
def fibo_rec(n,a,b):
  if n==0:
    return b
  ...
  return ...
# wrapper :
def fibo2(n):
  return fibo_rec(n,0,1)
```

8.4.4 Valeurs par défaut

On aurait aussi pu utiliser les valeurs par défaut des arguments des fonctions, de la manière suivante :

```
def fibo(n, a=0, b=1):
  # même code que fibo_rec
```

Ainsi, lorsque la fonction est appelée avec un seul argument, les variables `a` et `b` prennent leurs valeurs par défaut (0 et 1 ici). Lorsque j'appelle la fonction `fibo(5)`, la fonction est exécutée en tant que `fibo(5, 0, 1)`, qui appelle `fibo(4, 1, 1)`, puis `fibo(3, 1, 2)`, `fibo(2, 2, 3)`, `fibo(1, 3, 5)`, et finalement `fibo(0, 5, 8)`. Le résultat est bien $F(5) = 8$.

8.4.5 Récursivité croisée

Certains algorithmes nécessitent de définir deux fonctions récursives imbriquées comme dans l'exemple suivant :

```
def a():
  # utilise b()
  ... b() ...

def b():
  # utilise a()
  ... a() ...
```

Ceci est parfaitement valide et peut être tout à fait justifié.

À titre d'exemple plus complexe, voici la définition de deux suites mathématiques dépendant l'une de l'autre, qui peuvent être calculées en utilisant des fonctions récursives multiples et croisées [2] :

$$F(n) = \begin{cases} 1 & \text{si } n = 0 \\ n - M(F(n-1)) & \text{sinon} \end{cases}$$

$$M(n) = \begin{cases} 0 & \text{si } n = 0 \\ n - F(M(n-1)) & \text{sinon} \end{cases}$$

Lorsqu'on écrit ce type de code, il est parfois difficile de vérifier que la récursivité se termine bien.

En général, si, pour exécuter la fonction récursive, on fait appel à la fonction récursive avec des paramètres *plus petits* que ceux reçus, et qu'il existe un plus petit paramètre (la condition d'arrêt), alors on est certain de l'atteindre et de s'arrêter lorsqu'on l'atteint. Si ce n'est pas le cas, on peut entrer dans une récursion infinie, et le programme ne s'arrête jamais !

La difficulté lorsqu'on définit deux fonctions récursives imbriquées, est de définir un *ordre bien fondé* portant sur les deux fonctions simultanément. C'est surtout difficile si, comme dans l'exemple ci-dessus, une fonction est appelée avec des arguments dépendant de la récursion elle-même.

2. C'est la suite femelle/mâle de Hofstadter, $F(x) \neq M(x)$ si et seulement si $x + 1$ fait partie de la suite de Fibonacci :)

Heureusement, en python, dès que 1000 (par défaut) appels récursifs sont atteints, le programme s'arrête et l'interpréteur python vous affiche un message d'erreur. Vous pouvez écrire des programmes faux sans risque de faire planter votre machine !

8.5 Exercices

1. Dessinez l'arbre d'appel de la fonction récursive `trier2()` donnée Section 8.4, pour trier une liste contenant 16 éléments. Combien y a-t-il d'appels à la fonction récursive ? Exprimez ce nombre en fonction de la taille de la liste d'entrée. Comparez à la version précédente de l'algorithme de tri (Section 8.2, page 84).

2. [à mettre dans le chapitre sur les tris] Écrivez la fonction `inserer()`, puis vérifiez que votre fonction `trier()` (page 84) fonctionne correctement. Ajoutez des appels à `print()` dans la fonction de tri, qui affiche la liste reçue en entrée et la liste triée produite en sortie par la fonction. Appelez cette fonction pour trier 10 entiers, observez la succession d'appels récursifs, et vérifiez que le résultat final est correct.

   Puis supprimez l'affichage, et mesurez la vitesse de tri de cette implémentation sur un tableau de 800 entiers. Pour mesurer le temps écoulé entre deux instants, vous ferez la différence entre deux valeurs de retour de la fonction `time()` du module `time` :

```
import time
debut = time.time()
#  ...
fin = time.time()
print("temps d'exécution", fin-debut, "secondes")
```

3. [à mettre dans le chapitre sur les tris] Écrivez la fonction `interclasser()`, puis faites la même chose que dans l'exercice précédent pour la fonction `trier2()` (page 89).

   Comparez les temps d'exécution des deux fonctions. Vous devriez observer que la deuxième est beaucoup plus rapide.

4. Dessinez l'arbre d'appel de la fonction `fact(10)` donnée Section 8.3.

5. Écrivez une fonction récursive `puissance(x,n)`, qui calcule $x^n = x * x * ... * x$, en n'utilisant que des multiplications (sans utiliser le symbole $**$).

6. Idem que l'exercice précédent, pour calculer un produit en n'utilisant que des additions : $a * b = b + b + ... + b$.

7. Écrivez une fonction récursive qui calcule le pgcd (Plus Grand Commun Diviseur) de deux nombres positifs a et b, en appliquant la formule :

$$pgdc(a,b) = \begin{cases} a & \text{si } b = 0 \\ pgcd(b, a \bmod b) & \text{sinon} \end{cases}$$

   Déroulez l'exécution de cette fonction pour $a = 60$ et $b = 100$.

8. Écrivez une fonction récursive qui calcule C_n^p pour tout p, n entiers positifs avec $p \leq n$.

   Rappel : C_n^p donne le nombre de combinaisons différentes de p éléments parmi n. Ce sont les nombres qui se trouvent dans le triangle de Pascal. La définition mathématique

récursive (qui utilise la propriété du triangle de Pascal : chaque entier est la somme des deux entiers adjacents de la ligne du dessus) est la suivante :

$$C_n^p = \begin{cases} 1 & \text{si } p = 0 \text{ ou } p = n \\ C_{n-1}^p + C_{n-1}^{p-1} & \text{sinon} \end{cases}$$

9. Cet exercice consiste à écrire une fonction récursive qui effectue une recherche dichotomique dans une liste triée.

   Le principe de la recherche dichotomique est de couper la liste triée en deux, comparer l'élément recherché avec l'élément central de la liste, puis de le rechercher à gauche ou à droite de cet élément central, selon qu'il est supérieur ou inférieur. En effet, si l'élément du milieu de la liste est plus grand que l'élément recherché, on sait que l'élément recherché se trouve dans la première moitié de la liste, à gauche de l'élément central (puisque la liste est triée). S'il est égal, on l'a trouvé, et on peut retourner son indice. Sinon, il est à droite, dans la deuxième partie de la liste.

   Votre fonction de recherche prendra en argument une valeur recherchée et un tableau trié, et renverra l'indice de l'élément s'il est trouvé dans la liste, ou la valeur spéciale `None` sinon.

10. Écrivez une fonction qui compte le nombre d'occurrences d'une valeur dans une liste triée, en effectuant une recherche dichotomique de la première et de la dernière occurrence de la valeur dans la liste. La fonction renverra 0 si la valeur n'apparaît pas du tout dans la liste.

11. Écrivez les trois versions du calcul d'un nombre de la suite de Fibonacci, comme présenté Section 8.4 : la version naïve, la version qui renvoie un couple de valeurs, et la version inversée qui prend deux arguments supplémentaires.

    Dessinez l'arbre d'appel de la fonction récursive avec ses paramètres dans chacun des trois cas, pour $n = 6$.

12. Écrivez une fonction récursive `Rom2Dec()` qui fait la conversion d'un nombre romain vers un nombre décimal : elle prend en entrée une chaîne de caractères (par exemple "MCMXCIX") et renvoie un nombre (1999 ici), en traitant un caractère après l'autre.

    Attention, si un chiffre romain est suivi d'un chiffre romain plus grand, le premier doit être soustrait au résultat ! Sinon, il doit être additionné au résultat.

    Utilisez la fonction donnée ci-dessous pour faire la conversion d'un chiffre romain unique :

```
# tableau de couples qui sert à faire les conversions
conversion = [ (1000,"M"), (500,"D"), (100,"C"), (50,"L"),
               (10,"X"), (5,"V"), (1,"I") ]
# conversion chiffre romain -> décimal
def r2d_c(c):
    for (val,char) in conversion:
        if char == c:
            return val
    return 0
```

13. Tour de Hanoï...

Chapitre 9

Algorithmes et complexité

9.1 Qu'est-ce qu'un algorithme ?

9.1.1 Faire abstraction des langages de programmation et du matériel

Lorsqu'un programmeur souhaite demander à l'ordinateur de réaliser une tâche, il a plusieurs façons d'exprimer cette demande dans un programme. Il a à sa disposition plusieurs *langages* de programmation. Chaque langage possède des règles de syntaxe propores. Certains langages sont assez proches du langage naturel, alors que d'autres langages peuvent nécessiter une connaissance assez poussée du matériel électronique. Par exemple voici deux petits programmes. Le premier est écrit en langage C et le second en langage *csh* (C-shell).

```
void main()
{
  int N=5;
  int somme=0;
  int compteur;
  for(compteur=1;
      compteur<=N;
      compteur++)
    somme += compteur;
  printf("%d",somme);
}
```

```
#!/bin/csh

set N=5
set somme=0
set compteur=1
while ( $compteur <= $N )
  @ somme = $somme + $compteur
  @ compteur  = $compteur + 1
end
echo $somme
```

Le programmeur doit avoir à l'esprit deux choses :

1. quelles sont les opérations qu'il souhaite effectuer, independamment du langage de programmation ?
2. comment réaliser ces opérations au moyen d'un programme syntaxiquement correct ? Faut-il des parenthèses ? un point-virgule ? des accolades ? une boucle *for* ou une boucle *while* ? etc.

Le premier point a trait à l'*algorithme* et le second point à l'*implémentation* de cet algorithme dans un langage donné. Il est clair que les deux programmes ci-dessus ne suivent pas du tout les

mêmes règles sytaxiques. Par contre, malgré des syntaxes visiblement différentes, on peut reconnaître des similitudes. Si on connaît les deux langages, on peut dire que les deux programmes réalisent exactement la même fonction (à savoir calculer la sommes des N premiers entiers). Et on peut même dire qu'ils le font exactement de la même manière. Voici la suite des opérations dans les deux programmes :

> Soit un nombre *somme* valant 0. Ajouter successivement à *somme* les entiers 1 puis 2 puis 3 jusqu'à N. Lorsque le processus est fini, *somme* est égal à la somme des N premiers entiers.

Ce qui précède peut être appelé l'*algorithme* des deux programmes. Les algorithmes peuvent revêtir la forme d'une description en langage naturel comme ci-dessus, mais il y a de nombreuses autres façons de l'exprimer. Par contre il est nécessairement indépendant de toute question de langage informatique ou de matériel. Avant d'écrire la première ligne de programme, tout programmeur a d'abord du élaborer cet algorithme plus ou moins explicitement.

Les programmeurs débutants ont souvent tendance, pour des programmes simples, à esquiver l'explicitation de l'algorithme et de passer directement à l'implémentation. Mais lorsqu'il faut chercher l'erreur dans un programme plus important, la distinction entre syntaxe et sens devient un passage obligé. De façon générale, comme nous allons le voir, dès l'étape de l'algorithme et sans se soucier de l'implémentation, on peut voir si le futur programme :

— pourra parvenir à résoudre le problème ;
— sera général ou s'appliquera à quelques cas particuliers ;
— sera toujours très rapide ou, au contraire, s'avérera très lent pour des problèmes plus importants.

9.1.2 Définition

Nous nous intéressons aux algorithmes dans le contexte informatique, mais cette notion est bien plus générale. En voici une définition.

> Un algorithme est une suite finie d'opérations élémentaires constituant un schéma de calcul ou de résolution d'un problème.

Ainsi, si le problème à résoudre est de cuisiner un plat, la recette de cuisine associée à ce plat en sera l'algorithme.

9.1.3 Les *tours de Hanoï*

Prenons par exemple un problème devenu célèbre : le problème des *tours de Hanoï*. Il s'agit d'un jeu de réflexion consistant à déplacer des disques de diamètres différents d'une tour à une autre tour en passant par une troisième tour, et ceci en un minimum de coups, tout en respectant les règles suivantes :

— on ne peut déplacer plus d'un disque à la fois,
— on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.

On suppose que cette dernière règle est également respectée dans la configuration de départ.

Que proposez-vous comme algorithme pour trois disques ?

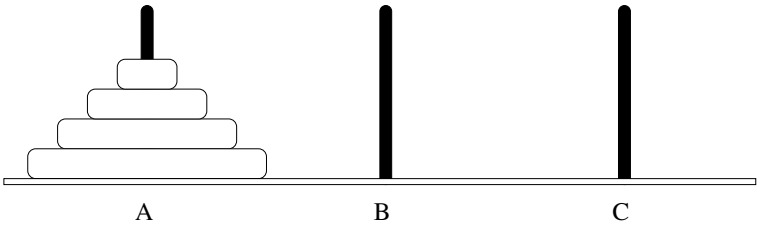


FIGURE 9.1 – Le problème des *tours de Hanoï* : on dispose de N disques qu'il faut déplacer de la tour A à la tour B en respectant certaines règles. On peut également temporairement poser des disques sur la tour C.

9.1.4 L'équation du second ordre

Pour revenir à un contexte de calcul, vous savez tous comment résoudre une équation du second ordre :

$$ax^2 + bx + c = 0$$

- Calcul du discriminant $\Delta = b^2 - 4ac$.
- Si $\Delta < 0$ alors il n'y a pas de solution ;
- Si $\Delta = 0$ alors il y a une seule solution égale à $x = -\frac{b}{2a}$;
- Si $\Delta > 0$ alors il y a deux solutions respectivement égales à $x_1 = \frac{-b-\sqrt{\Delta}}{2a}$ et $x_2 = \frac{-b+\sqrt{\Delta}}{2a}$;

Les algorithmes comme celui de la résolution de l'équation du second ordre peuvent être qualifiés de *directs* dans la mesure où ils donnent directement un résultat exact. Ces algorithmes sont très intéressants pour leur rapidité et pour leur exactitude. Les concepteurs de modèles en tout genre sont très satisfaits lorsqu'ils peuvent utiliser ce type de résultat. Mais très peu de problèmes peuvent trouver des solutions aussi satisfaisantes. Par ailleurs, ces calculs ne représentent aucune difficulté pour les ordinateurs. C'est la raison pour laquelle les informaticiens ne s'y intéressent pas beaucoup.

Pour la grande majorité des modèles, la solution est trouvée après une suite de calculs plus ou moins longue. Ce sont ces problèmes qui intéressent au plus haut point les informaticiens car ils mettent les machines à l'épreuve et permettent de distinguer les machines puissantes et les machines moins puissantes. Ils permettent aussi de distinguer les algorithmes efficaces et les algorithmes moins efficaces.

9.2 Algorithmes récursifs et itératifs

9.2.1 Définition

- Un algorithme *itératif* est un algorithme qui génère une séquences de solutions qui approchent de plus en plus de la solution définitive, cette dernière pouvant être approchée ou vraiment atteinte.
- Un algorithme *récursif* est un algorithme fondé sur la récurrence. Il s'agit de
  - fournir la solution du problème dans un cas très simple ;

— de trouver la solution du problème en fonction de celle d'un problème légèrement plus simple (i.e. plus proche du problème ci-dessus).

Le mécanisme de la récursivité fait le reste. La résolution du problème complexe est soumis à celle d'un problème plus simple. Celui du problème plus simple est soumis à celle d'un problème encore plus simple et ainsi de suite jusqu'à ce que le problème soit celui du cas trivial fourni. Le problème trivial a une réponse, ce qui fournit donc la réponse du cas un peu plus complexe dont il dépendait et ainsi de suite : tous les problèmes successifs trouvent leur solution.

Dans un algorithme itératif, on part d'un problème que l'on sait résoudre pour approcher la solution définitive. Dans un algorithme récursif, on part d'un problème qu'on ne sait pas résoudre pour parvenir finalement à un problème dont on connaît la solution.

9.2.2 Exemples

9.2.2.1 Calcul d'une suite géométrique

Soit une suite géométrique de raison r et de premier terme u_0

$$\forall n \in \mathbb{N}, u_n = u_0 \times r^n$$

Solution itérative On connaît le premier terme u_0. La solution itérative consiste à initialiser un nombre x à u_0 et, ensuite, par n fois, de mutiplier x par r. La valeur de x sera celle du résultat u_n.

Solution récursive La solution récursive est fondée sur la valeur du premier terme u_0. Pour trouver u_n, il faut chercher u_{n-1} et le multiplier par r. L'algorithme récursif tient dans les deux phrases précédentes.

Voici comment le mécanisme sous-jacent de la récursivité trouve la solution avec ces deux phrases : La valeur de u_n dépend de la valeur de u_{n-1} qui, elle-même dépend de celle de u_{n-2}. Et ainsi de suite : la valeur de u_1 dépend de celle de u_0 qui est connue. Cette valeur fournit la valeur de u_1 (grâce à la relation de récurrence), qui, elle-même fournit la valeur de u_2 et ainsi de suite jusqu'à donner la valeur de u_n.

9.2.2.2 Calcul de la suite factorielle $n!$

Solution itérative On sait que pour $0! = 1$. La solution itérative consiste à initialiser un nombre x à $0! = 1$ et, ensuite, de mutiplier x par 1, puis par 2, puis par 3 et ainsi de suite jusqu'à n. À la fin de ce processus, la valeur de x sera celle du factoriel de n.

Solution récursif La solution récursive est fondée sur la valeur du premier terme $0! = 1$. Pour trouver $n!$ il faut connaître $(n-1)!$ et multiplier sa valeur par n. Le mécanisme sous-jacent de la récursivité est le même que ci-dessus.

9.2.2.3 Application

Il est quelquefois difficile de voir l'intérêt des solutions récursives. Pour s'en convaincre, essayez de généraliser l'algorithme des *Tours de Hanoï* au cas de n disques. Énoncez l'algorithme

récursif puis, dans un deuxième temps, suivez le mécanisme sous-jacent de la récursion pour en vérifier le fonctionnement.

9.3 Généralité, Convergence

9.3.1 Trouver la solution de l'équation $f(x) = 0$

Voici un autre exemple illustrant la notion de généralité et de convergence. On souhaite résoudre l'équation $f(x) = 0$ et trouver un résultat à ϵ près. On sait seulement que la fonction f est une fonction continue de $\mathbb{R}$ dans $\mathbb{R}$. On cherche la solution x_0 telle que $f(x_0) = 0$ dans un intervalle $[a, b]$.

Quelles solution proposez-vous ?

9.3.1.1 Solution brute

On parcourt l'intervalle $[a, b]$ par pas de ϵ et on teste les différentes valeurs de f : $f(a)$, $f(a+\epsilon)$, $f(a+2\epsilon)$, $f(a+3\epsilon)$ etc. jusqu'à trouver deux valeurs successives pour lesquelles f n'a pas le même signe. Plus explicitement, si $f(a+n\epsilon) < 0$ et $f(a+(n+1)\epsilon) > 0$ ou $f(a+n\epsilon) > 0$ et $f(a+(n+1)\epsilon) < 0$ alors en choisissant comme solution $x_0 = a + n\epsilon$ on sera certain d'avoir la solution à ϵ près.

9.3.1.2 Dichotomie

Supposons, à présent, que la fonction f soit monotone sur l'intervalle $[a, b]$. Par exemple supposons qu'elle soit croissante. On suppose aussi que $f(a) < 0$ et $f(b) > 0$ sinon il est clair qu'aucune solution n'existe. Soit $m = (a + b)/2$ le milieu de l'intervalle $[a, b]$.

— Si $f(m) < 0$ alors la solution ne peut pas se trouver dans l'intervalle $[a, m]$ mais nécessairement dans l'intervalle $[m, b]$;

— Si $f(m) > 0$ alors la solution est nécessairement dans l'intervalle $[a, m]$.

À présent, on applique au nouvel intervalle ($[a, m]$ ou $[m, b]$) le même traitement qu'à l'intervalle $[a, b]$.

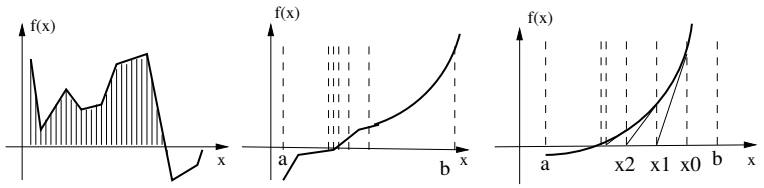
On voit ainsi qu'à chaque pas de calcul, on élimine la moitié des valeurs à tester. Cet algorithme donne un résultat bien plus rapidement que le précédent.

9.3.1.3 Méthode de Newton

Supposons, à présent que la fonction f est dérivable que sa dérivée ne s'annule jamais sur l'intervalle $[a, b]$. Soit Δ une droite tangente à la courbe de f au point $x_0 = (a + b)/2$. Cette droite coupe l'axe x en un point x_1. On recommence l'opération précédente mais en prenant la tangente de la fonction f au point x_1 jusqu'à ce que $f(x_n) < \epsilon$. Pour une même précision ϵ souhaitée, l'algorithme de Newton trouve un résultat bien plus rapidement qu'avec l'algorithme de la dichotomie.

9.3.1.4 Généralité vs vitesse de convergence

En conclusion, nous avons vu trois algorithmes, le premier étant le moins efficace et le troisième étant le plus efficace. Cependant ces accélérations n'ont été possibles qu'au prix d'une perte de généralité. Autrement dit, le premier algorithme, malgré sa faible efficacité est le plus



(a) (b) (c)

FIGURE 9.1 – Trois méthodes pour trouver une solution à l'équation $f(x) = 0$. La solution brute (a) convient à toute fonction continue. La méthode de la dichotomie (b) convient pour les fonctions continues et monotones, mais converge plus vite. La méthode le plus rapide est celle de Newton (c) mais ne marche que pour des fonctions dérivables dont la dérivée ne s'annule pas dans l'intervalle de recherche.

général et le troisième algorithme est le moins général car il suppose une fonction dérivable à dérivé croissante, alors que le premier algorithme suppose seulement que f soit continue.

9.4 Complexité

9.4.1 définition

Nous avons vu, dans les sections précédentes, qu'un même problème peut être résolu par plusieurs algorithmes, chaque algorithme ayant des spécificités en termes de généralité, de vitesse de convergence : de *ressources*. Dans le contexte informatique, *ressources* peut signifier :

— temps de calcul ;
— espace mémoire ;
— processeurs (dans le cas d'algorithmes sur plusieurs processeurs).

L'utilisation des ressources par un même algorithme dépend souvent de la *taille* du problème à résoudre. En toute rigueur, la taille du problème est le nombre de bits d'information nécessaires pour décrire ce problème. En pratique, la *taille* désigne un entier qui semble décrire de façon pertinente cette information. Par exemple :

— si nous avons une suite de N nombres et que le problème consiste à les arranger dans l'ordre croissant, la *taille* du problème est le nombre N de nombres à trier.
— si le problème est celui du paragraphe 9.3.1 on peut dire que la *taille* du problème est la longueur de l'intervalle divisé par la précision souhaitée : $(b - a)/\epsilon$.

La *complexité* d'un algorithme est la façon dont évolue le besoin en ressources de cet algorithme en fonction de la *taille* du problème à résoudre.

9.4.2 Classes de complexité

9.4.2.1 Évaluation expérimentale vs calcul analytique

Comment peut-on obtenir la complexité d'un algorithme. Une solution simple serait d'implémenter l'algorithme et de mesurer le temps et l'espace mémoire qu'il utiliserait pour différentes valeurs de taille. Il y existe des situations où on ne peut pas faire autrement que de mesurer les per-

formances d'un algorithme de façon expérimentale. L'inconvénient de la méthode expérimentale est qu'elle ne peut pas être exhaustive.

Lorsqu'on compare deux algorithmes A_1 et A_2 de façon expérimentale, on peut trouver que A_1 est plus performant que A_2 pour toutes les tailles testées. Rien ne prouve que pour des tailles 100 fois plus élevées ou 1000000 fois plus élevées la comparaison ne donnera pas un résultat différent voire opposé. (On en verra une application concrète au paragraphe 9.4.2.3.) Par exemple un nouvel opérateur de téléphonie mobile peut tester le fonctionnement de son équipement sur 100 utilisateurs, pourquoi pas 1000 ? Il ne pourra jamais tester le foncionnement sur 10^7 utilisateurs répartis dans le monde (10^7 est le nombre d'abonnés de *free*). Il doit savoir, à l'avance, comment réagira ses équipement pour de très grands nombres d'utilisateurs.

Pour pouvoir évaluer les performances d'un algorithme dans tous les cas et, en particulier, dans les cas des problèmes de très grande taille, on cherche, lorsque cela est possible, de *calculer* leur complexité de façon analytique. Autrement dit, on cherche à établir une relation explicite entre les performances d'un algorithme et sa taille N.

9.4.2.2 Le comportement asymptotique

Les fonctions polynômiales Le fait de s'intéresser aux performances d'un algorithme pour les grandes tailles simplifie les comparaisons. Que pensez-vous de la comparaison entre les deux fonctions suivantes ?

— $f_1(x) = 0.0000001x^3$
— $f_2(x) = 100000000x^2 + 1000000000000x + 1000000000000000000$

Évidemment, pour de faibles valeurs de x, f_1 a des valeurs bien plus faibles que f_2. Par contre il suffit de visualiser les deux courbes grâce à gnuplot. On peut voir qu'au delà d'un certain seuil (en l'occurrence $x = 2.10^{15}$) c'est f_1 qui deviendra plus grand. De fait le terme d'ordre le plus élevé dans f_1 est le terme d'ordre 3 (x^3) alors que dans f_2 le terme d'ordre le plsu élevé est le terme d'ordre 2. De façon générale, pour comparer deux fonctions polynômiales pour de grandes valeurs de x, il suffit de comparer le terme d'ordre le plus élevé. Par exemple dans le cas suivant :

— $f_a(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_1 x + a_0$
— $f_b(x) = b_m x^m + b_{m-1}x^{m-1} + ... + b_1 x + b_0$

Si $n > m$ alors on montre que, quelques soient tous les coefficients des deux polynômes, (pour peu que $a_n > 0$ et $b_m > 0$) il existe un seuil à partir duquel f_a aura toujours des valeurs plus élevées que f_b. Dans toute la suite, on convient que, pour comparer le comportement de deux fonctions polynômiales pour de grandes valeurs, on peut se contenter de comparer le comportement du terme d'ordre le plus élevé.

Autres fonctions On peut aussi comparer le comportement des fonctions polynômiales, des fonctions logarithmiques et exponentielles.

— $f_a(x) = a_n log(x)$
— $f_b(x) = b_m x^m + b_{m-1}x^{m-1} + ... + b_1 x + b_0$
— $f_c(x) = c_p e^x$

On montre ainsi que, quelques soient les coefficients de toutes ces fonctions, (pour peu que $a_n > 0$, $b_m > 0$ et $c_p > 0$) et quelque soit le degré de la fonction polynômiale (! ! !) :

$$\exists X_0 \in \mathbb{R} \mid \forall x > X_0 \quad b_m x^m + b_{m-1}x^{m-1} + ... + b_1 x + b_0 > a_n log(x)$$
$$\exists X_1 \in \mathbb{R} \mid \forall x > X_1 \quad b_m x^m + b_{m-1}x^{m-1} + ... + b_1 x + b_0 < c_p e^x$$

Autrement dit, pour des valeurs assez grandes, une fonction exponentielle est toujours plus grande qu'une fonction polynômiale et une fonction polynômiale est toujours plus grande qu'une fonction logarithmique.

Voici quelques ordres de grandeurs pour situer les idées :

$log_2(n)$	n	$nlog_2(n)$	$n2$	$n3$	2^n
3	10	33	100	1 000	1 000
7	100	660	10 000	10^{06}	[d]$1,3.10^{30}$
10	1 000	10 000	10^{06}	10^{09}	∞
13	10 000	130 000	10^{08}	[b]10^{12}	∞
17	100 000	$1,7.10^6$	10^{10}	10^{15}	∞
20	1 000 000	[a]20.10^6	[b]10^{12}	[c]10^{18}	∞

Temps d'exécution à 1 giga-flops :

[a] 20 ms.

[b] 17 minutes.

[c] 32 années.

[d] 40 milliards d'années.

Ordres de grandeur pour n :

— Téléphonie mobile (France, 2008) : plusieurs 104 antennes émettrices, plusieurs 107 abonnés.

— Quelques 105 polygones à visualiser en informatique graphique.

— Projet du génome humain : identifier 100.000 gènes, trouver les séquences parmi les 3.109 paires de bases chimiques de l'ADN.

— Google : plusieurs 105 serveurs, 10^{12} pages référencées en 2008.

9.4.2.3 Exemple

On cherche à simuler physiquement le mouvement de N balles en mousse de même taille dans une boîte fermée. La simulation consiste, à chaque pas, à calculer successivement la somme des forces appliquées à chaque balle par (potentiellement) toutes les autres balles puis, à partir de cette force, de calculer le déplacement de chaque balle. Au pas suivant, on calculera de nouveau la somme des forces résultant des nouvelles positions calculées au pas précédent et ainsi de suite.

On notera que, pour le calcul des forces, la partie la plus coûteuse est le calcul de la distance entre les deux balles.

On propose deux variantes pour ce calcul :

— (Algorithme A_1) : pour le calcul des forces, on calcule la distance entre chaque balle et toutes les autres balles. Et en fonction de cette distance, on décidera s'il faut ou non calculer une force.

— (Algorithme A_2) : on subdivise l'espace de la simulation en cases cubiques de la taille d'une balle. On associe à chaque case la liste des balles qu'elle contient. Cette liste est remise à jour à chaque pas de simulation. Ces cases permettent de ne pas calculer la distance entre une balle et toutes les autres, mais seulement les balles qui se trouvent dans les cases voisines.

La taille de ce problème est N le nombre de balles. D'un point de vue espace mémoire, A_2 est clairement moins performant que A_1. Même le temps de calcul de A_1 peut être plus faible

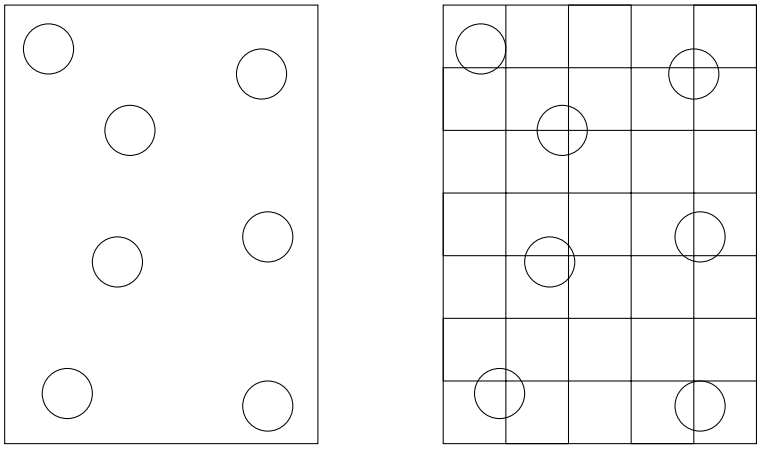


FIGURE 9.1 – Il s'agit de calculer, pour chaque balle, la somme des forces exercées par les autres balles et les parois. À gauche, l'algorithme A_1 et, à droite, l'algorithme A_2.

que celui de A_2 pour de faibles valeurs de A_2. Par contre, dès que N devient plus important, les temps de calcul augmentent très rapidement pour A_1.

Par ailleurs, on remarque aussi que le temps de calcul de A_2 varie beaucoup en fonction des simulations. Notamment les temps de calcul augmentent beaucoup lorsque se forment des amas denses de balles dans certaines régions. Il est difficile de conclure lorsque les résultats sont si variables.

Dans l'exemple ci-dessus, (algorithme A_1) il faut calculer la distance de chaque balle à toutes les autres balles. Quel est le nombre de calculs de distances ? (Réponse : $N \times (N-1)/2$). Comme on s'intéresse seulement aux termes de plus haut degré, on dit que la complexité de l'algorithme A_1 croît en $O(n^2)$.

Pour l'algorithme A_2, le calcul des forces nécessite le calcul de la distance de chaque balle avec les balles contenues dans les 8 voisins de sa case. Donc si on suppose que chaque case contient une seule balle (à cause de la taille des cases) le calcul des forces nécessite en tout $N \times 8$ calculs de distance. Dans ce cas, on dit que la complexité de l'algorithme augmente en $O(n)$.

9.4.2.4 Le pire des cas

Si les balles étaient des billes indéformables, cette conclusion aurait été absolument correcte. Mais précisément, il ne s'agit pas de billes mais de balles en mousse déformables. Et nous avions observé des changements importants de performances lorsque la simulation présentait des amas denses de balles à certains endroits. Pour résoudre ce type d'incertitude dépendant de conditions aléatoires, on décide toujours de calculer la complexité de l'algorithme dans le pire des cas, c'est à dire dans la situation où l'algorithme est le moins performant.

Dans le cas des balles en mousse, le pire des cas est celui où toutes les balles en mousse se trouvent dans les 8 cases entourant une case donnée. Dans une telle situation, chaque pas de simulation nécessite de nouveau le calcul de la distance de chaque balle avec toutes les autres balles. Ainsi dans le pire des cas, la complexité de l'algorithme A_2 avec des balles en mousse évolue en $O(n^2)$ donc même résultat que pour l'algorithme A_1.

9.4.3 Autres exemples

9.4.3.1 Complexité croissant de façon exponentielle

On peut ainsi définir des classes de problème. Nous avons vu un problème dont la complexité augmente en $O(n)$, un autre en $O(n^2)$. On peut imaginer des problèmes qui augmentent en $O(n^3)$ ou en $O(n^9)$. Mais ce ne sont pas encore les problèmes qui donnent le plus de cauchemars aux informaticiens. La complexité de ces algorithmes augmente de façon *polynômiale*. Quel type de fonction croît plus rapidement qu'une fonction polynômiale ?

Soient N points. Trouver le chemin le plus long parcourant tous ces points. Pour $N = 3$ points, la solution est vite trouvée. Dans le cas général, pour faire court, cela revient à tester tous les chemins possibles passant une et une seule fois par chaque point. Quel est le nombre de chemins ? (Réponse : $n!$). Or on sait que $n!$ augmente de la même façon que n^n. La complexité de ce problème croît plus rapidement qu'une fonction exponentielle. On ne peut pas imaginer d'algorithme moins efficace.

Chapitre 10

Les algorithmes de tri

En construction