Frédéric Peschanski
Romain Demangeon

Éléments de programmation

De l'algorithme au programme Python



ellipses

Références sciences

Éléments de programmation

De l'algorithme au programme Python

Frédéric Peschanski
Romain Demangeon

ellipses

Collection Références sciences

dirigée par Paul de Laboulaye
paul.delaboulaye@editions-ellipses.fr

Retrouvez tous les livres de la collection et des extraits sur www.editions-ellipses.fr

ISBN 9782340-041066
©Ellipses Édition Marketing S.A., 2020
32, rue Bargue 75740 Paris cedex 15

DANGER
LE PHOTOCOPILLAGE TUE LE LIVRE

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5.2° et 3°a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective », et d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).
Cette représentation ou reproduction, par quelque procédé que ce soit constituerait une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

www.editions-ellipses.fr

Préface

Une polémique, presqu'aussi vieille que la science informatique, concerne la voie que doit prendre une initiation à cette science. L'informatique s'appuie sur des artefacts matériels (ordinateurs, routeurs, tablettes, téléphones, etc.) et sur des textes (logiciels, protocoles, normes, etc.) effectuables, c'est-à-dire capables d'agir sur le monde. Dès lors, une initiation passe nécessairement par l'écriture de tels textes afin que les apprenants se rendent compte de l'immense pouvoir qu'ils peuvent ainsi développer. Ainsi la programmation dans un langage dit de programmation s'impose classiquement.

Dès les années soixante (1960 !) plusieurs langages existaient, de nature assez différente, entre Fortran, Lisp, Cobol, Algol, et déjà s'écharpaient les tenants des uns et des autres. Sans rentrer dans plus d'un demi-siècle de batailles toujours actives, nous ne mentionnerons que ces deux seuls arguments : l'imprégnation et l'utilité.

L'imprégnation est d'essence subtile car l'exposition d'un apprenant à un langage de programmation façonne durablement sa manière de voir l'informatique. Aussi faut-il ne pas se rater ! Quant à l'utilité, nous la résumerons en « Si apprendre un langage est nécessaire, autant que ce langage soit utile ultérieurement. »

Et c'est là que le langage Python s'impose par ses qualités propres : il n'est pas trop compliqué, il est assez régulier, il procure de nombreux styles (il est dit multi-paradigmatique) et, en plus, il est très largement utilisé. C'est donc naturellement que ce livre l'emploie pour proposer une initiation à l'informatique.

Il existe une multitude d'ouvrages publiés sur la programmation en Python, et il est important de souligner l'originalité du travail effectué par Frédéric Peschanski et Romain Demangeon. Cette originalité tient à la présentation de différents aspects complémentaires fondamentaux pour la conception et l'écriture des programmes : spécification et implantation, validation et analyse. L'ouvrage ne suppose aucune connaissance préalable en informatique, mais se fixe des objectifs ambitieux : manuel de Python certes, mais aussi introduction rigoureuse à la programmation, avec des ouvertures sur d'autres types de langages et de paradigmes de programmation. Les bonnes pratiques enseignées par cet ouvrage d'initiation permettront aux non informaticiens de programmer avec rigueur dans ce langage omniprésent qu'est Python, et elles constituent aussi une couche de base solide pour les enseignements d'informatique ultérieurs.

Cet ouvrage est le fruit d'un cours délivré depuis 2014 aux étudiants de premier semestre de première année de licence à Sorbonne Université (naguère connue comme UPMC –

Université Pierre et Marie Curie – ou Paris 6). Des milliers d'étudiants l'ont suivi et ont ainsi contribué à le parfaire, par leurs questionnements et leurs errements, par leurs découvertes et leurs joies, Des centaines d'enseignants y ont aussi participé, apportant leur pierre à l'édifice.

Bien que Python soit un langage typé dynamiquement, les auteurs insistent, à bon escient, sur l'écriture explicite des signatures des fonctions, attirant ainsi l'attention sur la notion de contrat entre client et fournisseur de fonctions. Cette insistance de bon aloi nous semble précieuse car pavant la voie vers l'utilisation de langages typés fortement comme Java.

Ce livre impressionnant (plus de 300 pages) présente les notions en bon ordre, de façon très claire et très progressivement : boucles, listes, compréhensions, dictionnaires et enfin objets. Il comporte de plus de très nombreux exercices guidés et corrigés.

Fruit de réflexions pédagogiques multiples et d'expériences sur plusieurs années, cet ouvrage de Frédéric Peschanski et Romain Demangeon, soutenus par une large équipe enseignante, est une fort recommandable voie d'initiation à l'informatique. Nous sommes heureux de voir cette nouvelle équipe de Sorbonne Université poursuivre l'aventure que nous avions commencée, au tout début des années 2000 et pour le même public, autour de la programmation récursive [1] et nous sommes persuadés que de très nombreux lecteurs, d'horizons divers, éprouveront un grand plaisir à étudier cet ouvrage dense, riche et stimulant.

Paris, le 7 mai 2020

Christian Queinnec, Michèle Soria
– Professeurs émérites, Sorbonne Université

1. Anne Brygoo, Titou Durand, Maryse Pelletier, Christian Queinnec et Michèle Soria, *Programmation récursive (en Scheme)* Dunod, 2004, ISBN 978-2-10-007479-2.

Table des matières

Avant-propos **1**

1 Premiers pas **7**
1.1 Un exemple de programme . . . 7
1.2 Notion d'expression . . . 9
1.2.1 Expressions atomiques . . . 10
1.2.2 Expressions composées . . . 13
1.3 Définition de fonctions simples . . . 18
1.3.1 la spécification de la fonction . . . 20
1.3.2 l'implémentation du corps de la fonction . . . 22
1.3.3 la validation de la fonction par un jeu de tests . . . 22
1.4 Exercices corrigés . . . 24
Exercice 1.1 : Moyenne de trois nombres . . . 24
Exercice 1.2 : Calcul d'un prix TTC . . . 25
Exercice 1.3 : Calcul de fonctions polynomiales . . . 26

2 Instructions, variables et alternatives **27**
2.1 Suites d'instructions . . . 28
2.2 Variables et affectations . . . 30
2.2.1 Exemple de calcul avec variable : l'aire d'un triangle . . . 31
2.2.2 Utilisation des variables . . . 34
2.3 Alternatives . . . 39
2.3.1 Syntaxe et interprétation des alternatives simples . . . 40
2.3.2 Expressions booléennes . . . 43
2.3.3 Alternatives multiples . . . 48
2.4 Exercices corrigés . . . 49
Exercice 2.1 : Variables et affectations . . . 49
Exercice 2.2 : Calcul des mentions . . . 50
Exercice 2.3 : Couverture . . . 51
Exercice 2.4 : Mesures fiables . . . 52

3 Répétitions et boucles **53**
3.1 Motivation : répéter des calculs . . . 53
3.2 La boucle while . . . 55
3.2.1 Principe d'interprétation . . . 55

3.2.2 Simulation de boucle . . . 56
3.2.3 Tracer l'interprétation des boucles avec print . . . 58
3.3 Problèmes numériques . . . 60
3.3.1 Calcul des éléments d'une suite . . . 60
3.3.2 Calcul d'une somme . . . 61
3.3.3 Exemple de problème plus complexe : le calcul du PGCD . . . 64
3.3.4 Un exemple avec des boucles imbriquées . . . 66
3.4 Complément : les fonctionnelles . . . 67
3.5 Exercices corrigés . . . 69
Exercice 3.1 : Somme des impairs . . . 69
Exercice 3.2 : Fonction mystère . . . 70
Exercice 3.3 : Nombres premiers . . . 71

4 Plus sur les boucles 73
4.1 Notion de correction . . . 73
4.2 Notion de terminaison . . . 77
4.3 Notion d'efficacité . . . 79
4.3.1 Factoriser les calculs . . . 79
4.3.2 Sortie anticipée . . . 80
4.3.3 Efficacité algorithmique . . . 83
4.4 Complément : la récursion . . . 88
4.5 Exercices corrigés . . . 90
Exercice 4.1 : Retour sur la factorielle . . . 90
Exercice 4.2 : Fonction mystère . . . 91
Exercice 4.3 : Retour sur l'algorithme d'Euclide . . . 93

5 Intervalles et chaînes de caractères 95
5.1 Intervalles . . . 95
5.1.1 Construction d'un intervalle . . . 95
5.1.2 Itération . . . 96
5.2 Chaînes de caractères . . . 98
5.2.1 Définition . . . 98
5.2.2 Opérations de base sur les chaînes . . . 98
5.3 Problèmes sur les chaînes de caractères . . . 106
5.3.1 Réductions . . . 106
5.3.2 Transformations et filtrages . . . 114
5.3.3 Exemples de problèmes plus complexes . . . 116
5.4 Exercices corrigés . . . 120
Exercice 5.1 : Intervalles . . . 120
Exercice 5.2 : Fonction mystère . . . 121
Exercice 5.3 : Palindromes . . . 121
Exercice 5.4 : Suppressions . . . 122

6 Listes 125
6.1 Définition et opérations de base . . . 125
6.1.1 Construction explicite . . . 125
6.1.2 Longueur de liste . . . 126

6.1.3 Égalité et inégalité . . . 127
6.1.4 Construction par concaténation . . . 128
6.1.5 Construction par ajout en fin de liste . . . 129
6.1.6 Accès aux éléments . . . 131
6.2 Découpages de listes . . . 132
6.2.1 Découpage simple . . . 132
6.2.2 Premiers et derniers éléments . . . 133
6.2.3 Découpage selon des entiers quelconques . . . 134
6.2.4 Découpage avec un pas positif . . . 135
6.2.5 Découpage avec un pas négatif . . . 136
6.3 Problèmes sur les listes. . . . 136
6.3.1 Réductions de listes . . . 137
6.3.2 Transformations de listes : le schéma `map` . . . 139
6.3.3 Filtrages de listes : le schéma `filter` . . . 142
6.3.4 Problèmes plus complexes . . . 146
6.4 Exercices corrigés . . . 151
Exercice 6.1 : Listes de répétitions . . . 151
Exercice 6.2 : Maximum d'une liste . . . 152
Exercice 6.3 : Liste de diviseurs . . . 153
Exercice 6.4 : Fonction mystère . . . 153
Exercice 6.5 : Découpages . . . 154

7 N-uplets 157
7.1 Construction . . . 157
7.2 Déconstruction . . . 158
7.2.1 Déconstruction dans des variables . . . 158
7.2.2 Déconstruction par indice . . . 159
7.3 Utilisation des n-uplets . . . 160
7.3.1 n-uplets en paramètres de fonctions . . . 161
7.3.2 n-uplets en valeur de retour . . . 163
7.3.3 Listes de n-uplets . . . 164
7.4 Exercices corrigés . . . 167
Exercice 7.1 : Nombres complexes . . . 167
Exercice 7.2 : Nombre d'occurrences du maximum . . . 168
Exercice 7.3 : Fichiers texte et système de facturation . . . 169

8 Compréhensions de listes 175
8.1 Schéma de construction . . . 175
8.1.1 Principes de base . . . 176
8.1.2 Syntaxe et principe d'interprétation . . . 176
8.1.3 Expressions complexes dans les compréhensions . . . 178
8.1.4 Constructions à partir de chaînes de caractères . . . 178
8.2 Schéma de transformation . . . 179
8.2.1 Construction à partir d'une liste . . . 179
8.2.2 Complément : la fonctionnelle map . . . 180
8.3 Schéma de filtrage . . . 181
8.3.1 Compréhensions avec filtrage : syntaxe et interprétation . . . 183

8.3.2 Complément : la fonctionnelle filter 184
8.4 Plus loin avec les compréhensions 185
8.4.1 Compréhensions de listes à partir d'autres séquences 185
8.4.2 Compréhensions sur les n-uplets 185
8.4.3 Compréhensions multiples 186
8.5 Complément : le schéma de réduction 189
8.6 Exercices corrigés 192
Exercice 8.1 : Revisiter les listes 192
Exercice 8.2 : Lettres de l'alphabet 192
Exercice 8.3 : Crible d'Eratosthène 194

9 Ensembles **197**
9.1 Définition et opérations de base 197
9.1.1 Construction explicite d'un ensemble 198
9.1.2 Test d'appartenance 199
9.1.3 Ajout d'un élément 200
9.1.4 Complément : retrait d'un élément 202
9.1.5 Itération sur les ensembles 203
9.1.6 Égalité ensembliste et notion de sous-ensemble 204
9.2 Opérations ensemblistes 206
9.2.1 Union 206
9.2.2 Intersection 208
9.2.3 Différence 209
9.3 Exercices corrigés 210
Exercice 9.1 : Différence symétrique 210
Exercice 9.2 : Répétitions dans les listes 211

10 Dictionnaires **213**
10.1 Définition et opérations de base 213
10.1.1 Construction explicite d'un dictionnaire 214
10.1.2 Accès aux valeurs 215
10.1.3 Test d'appartenance d'une clé 216
10.1.4 Ajout d'une association 216
10.1.5 Complément : retrait d'une association 219
10.2 Itération sur les dictionnaires 220
10.2.1 Itération sur les clés 220
10.2.2 Itération sur les associations 221
10.3 Exercices corrigés 224
Exercice 10.1 : Recettes de cuisine 224
Exercice 10.2 : Statistiques sur les lettres 226

11 Itérables et compréhensions **229**
11.1 Notion d'itérable 229
11.2 Compréhensions d'ensembles 231
11.2.1 Compréhensions simples 231
11.2.2 Compréhensions avec filtrage 233
11.2.3 Complément : typage des éléments d'un ensemble 236

11.3 Compréhensions de dictionnaires 236
11.3.1 Compréhensions simples 236
11.3.2 Compréhensions avec filtrage 241
11.4 Synthèse sur les compréhensions 244
11.5 Exercices corrigés 245
Exercice 11.1 : Compréhensions en vrac 245
Exercice 11.2 : Gestion de bibliothèque 246

12 Procédures et tableaux **249**
12.1 Notion de procédure 249
12.2 Instruction de remplacement 253
12.3 Algorithmes de tableaux 254
12.3.1 Un premier exemple : le renversement 255
12.3.2 Étude de cas 1 : le tri en place par sélection 257
12.3.3 Étude de cas 2 : les matrices 261
12.4 Exercices corrigés 271
Exercice 12.1 : Procédures modifiant des listes en place 271
Exercice 12.2 : Rotation de 180° d'une matrice 272
Exercice 12.3 : Rotation de 90° en place 273
Exercice 12.4 : Tri à bulles 277
Exercice 12.5 : Tri rapide 278

13 Vers les objets **281**
13.1 La programmation orientée objet (POO) 281
13.2 Objets du monde réel 282
13.3 Types de données utilisateur 285
13.3.1 Enregistrements 285
13.3.2 Types numériques 287
13.3.3 Types itérables 289
13.4 Aller plus loin 290

A Solutions des exercices **293**
A.1 Solutions du chapitre 1 293
A.1.1 Correction de l'exercice 1.1 293
A.1.2 Correction de l'exercice 1.2 294
A.1.3 Correction de l'exercice 1.3 294
A.2 Solutions du chapitre 2 296
A.2.1 Correction de l'exercice 2.1 296
A.2.2 Correction de l'exercice 2.2 298
A.2.3 Correction de l'exercice 2.3 300
A.2.4 Correction de l'exercice 2.4 300
A.3 Solutions du chapitre 3 303
A.3.1 Correction de l'exercice 3.1 303
A.3.2 Correction de l'exercice 3.2 305
A.3.3 Correction de l'exercice 3.3 306
A.4 Solutions du chapitre 4 308
A.4.1 Correction de l'exercice 4.1 308

A.4.2 Correction de l'exercice 4.2 310
A.4.3 Correction de l'exercice 4.3 311
A.5 Solutions du chapitre 5 314
A.5.1 Correction de l'exercice 5.1 314
A.5.2 Correction de l'exercice 5.2 315
A.5.3 Correction de l'exercice 5.3 316
A.5.4 Correction de l'exercice 5.4 318
A.6 Solutions du chapitre 6 319
A.6.1 Correction de l'exercice 6.1 319
A.6.2 Correction de l'exercice 6.2 321
A.6.3 Correction de l'exercice 6.3 322
A.6.4 Correction de l'exercice 6.4 323
A.6.5 Correction de l'exercice 6.5 324
A.7 Solutions du chapitre 7 326
A.7.1 Correction de l'exercice 7.1 326
A.7.2 Correction de l'exercice 7.2 327
A.7.3 Correction de l'exercice 7.3 328
A.8 Solutions du chapitre 8 330
A.8.1 Correction de l'exercice 8.1 330
A.8.2 Correction de l'exercice 8.2 331
A.8.3 Correction de l'exercice 8.3 332
A.9 Solutions du chapitre 9 334
A.9.1 Correction de l'exercice 9.1 334
A.9.2 Correction de l'exercice 9.2 335
A.10 Solutions du chapitre 10 337
A.10.1 Correction de l'exercice 10.1 337
A.10.2 Correction de l'exercice 10.2 340
A.11 Solutions du chapitre 11 343
A.11.1 Correction de l'exercice 11.1 343
A.11.2 Correction de l'exercice 11.2 344
A.12 Solutions du chapitre 12 345
A.12.1 Correction de l'exercice 12.1 345
A.12.2 Correction de l'exercice 12.2 347
A.12.3 Correction de l'exercice 12.3 348
A.12.4 Correction de l'exercice 12.4 351
A.12.5 Correction de l'exercice 12.5 352

Index **355**

Avant-propos

Chère lectrice, cher lecteur, les auteurs de cet ouvrage vous remercient pour votre acquisition (ou votre emprunt à une bibliothèque) et espèrent qu'il pourra vous apporter de nouvelles connaissances sur cette intrigante discipline de la *programmation informatique*. Dans cette partie préliminaire, nous souhaitons vous décrire le plus précisément possible ce que vous allez trouver (ou ne *pas* trouver) dans ce livre.

Avant toute chose, le livre «Élément de programmation» est issu d'un cours du même intitulé et dispensé chaque année à Sorbonne Université (ex. Université Pierre et Marie Curie – Paris 6). Ce cours est le premier cours d'informatique pour les étudiants de licence première année (L1) en cursus scientifique (mathématiques, physiques, chimie, biologie et bien sûr informatique). Ce cours concerne environ 1500 étudiants chaque année, et engage plus de 80 enseignants pour le dispenser. Pour autant, le livre n'est pas directement le support de cours, nous l'avons pensé et rédigé de façon à le rendre indépendant et auto-suffisant.

De la programmation

La programmation est une discipline à la fois nouvelle (née sous sa forme dite de «haut niveau» dans les années 1960) et déjà *vaste*. Il n'existe pas *une* unique programmation, mais un ensemble d'idées, de méthodes et d'outils permettant d'expliquer à un ordinateur, donc un objet dénué de «véritable» intelligence (malgré ce qui est parfois raconté dans les œuvres de science-fiction, ou même les magazines d'actualités [2]). Donc l'intelligence, dans cette démarche de programmation, est sous la responsabilité de l'humain : la programmeuse/le programmeur (ou en «tordant» un peu la langue française la développeuse/le développeur).

Pour que l'humain – la programmeuse ou le programmeur – explique à l'ordinateur ce qu'il doit faire, il lui faut disposer d'un langage, le *langage de programmation*. Au début de l'ère des ordinateur, ce langage était le *langage machine*, c'est-à-dire le langage directement compréhensible par la machine, fait de codage binaire (les fameux *bits* 0 et 1), d'opérations arithmétiques sur les *octets* et *mots mémoire*. Ce langage, dit de *bas niveau*, a rapidement montré ses limites et, surtout, les concepteurs de langages

2. Lire par exemple *L'intelligence artificielle n'existe pas* par Luc Julia (chez First Edition), concepteur d'une des «intelligences artificielles» les plus connues : l'assistant vocal SIRI.

de programmation ont rapidement trouvé des idées pour rendre les programmes plus faciles à écrire, moins directement pensés pour la machine. Ces nouveaux concepts et ces nouvelles constructions de langage dits de *haut niveau*, permettant de guider la machine de façon beaucoup plus «agréable» aujourd'hui, sont ce que nous nommons dans ce livre des *éléments de programmation*.

Du problème, à l'algorithme, au programme Python

L'important, pour nous, n'est pas de *connaître* le langage de programmation dans ses moindres retranchements (et l'univers Python est bien vaste), mais de *comprendre* comment, à partir d'un problème posé, on arrive finalement à un programme permettant de le résoudre, de façon automatique par un ordinateur.

Le chaînon manquant, entre le problème et le programme est, pour nous, la notion d'*algorithme* qui est à la base un concept mathématique de *calcul effectif*. Bien sûr, certains problèmes sont trop simples pour que l'on puisse réellement parler d'algorithme : on sait directement écrire le programme sans *trop* réfléchir. Cela rejoint la notion de programmation «boîte noire» qui consiste à exploiter des problèmes déjà résolus (avec programmes associés) et de les «assembler» (de façon parfois non-triviale) pour résoudre des problèmes plus complexes. Dans ce livre, nous nous positionnons un peu à mi-chemin entre ces deux «pôles». Nous exploiterons certaines «boîtes noires» de programmation, et notamment les structures de données de haut niveau proposées par le langage Python : listes, ensembles et dictionnaires. Nous nous reposerons également sur une abstraction de la machine, et en particulier la gestion automatique de la mémoire opérée par l'interprète Python. Mais pour les problèmes que nous nous poserons (par exemple une gestion simplifiée de bibliothèque, ou encore un livre informatisé de recettes de cuisine), nous construirons un certain nombre de «boîtes noires» qui seront exploitées pour résoudre le problème de départ. Ces constructions seront autant de fonctions (tant au sens mathématique qu'au sens spécifique à Python) mettant en œuvre un algorithme non-trivial sur lequel il faudra, parfois un certain temps, *réfléchir*.

Notre *leitmotiv*, pour ce livre, est de considérer chaque problème «complexe» comme la composition de sous-problèmes «moins complexes». Cette méthodologie, parfaitement efficace, se nomme *la décomposition fonctionnelle* de problèmes. Une fois chaque *sous-problème* identifié, l'objectif est de trouver un algorithme «raisonnablement» simple permettant, dans les principes, de le résoudre. Et finalement, de cet algorithme «suffisamment simple» découle *naturellement* un (fragment de) programme que l'on appelle une *implémentation* de l'algorithme. Ce sera dans notre cas (à l'exclusion du chapitre 12 notamment) une définition de fonction, tant au sens mathématique qu'informatique (et «pythonistique») du terme. Pour résoudre notre problème «complexe» de départ, il ne «restera plus» qu'à assembler les sous-problèmes par *composition* des fonctions déjà définies.

Cette démarche demande une grande expérience en matière de résolution de problème et de programmation, ce qu'il est impensable d'imposer en tant que pré-requis pour un livre d'introduction comme celui-ci. Notre démarche, en ce sens, est plutôt d'*illustrer*

cette méthode de développement de programme au travers des nombreux exemples et exercices (pour la plupart corrigés) proposés dans cet ouvrage.

Les éléments de programmation...

Dans ce livre, nous n'allons pas aborder *tous* les éléments de programmation imaginés durant les quelques soixante dernières années, mais une sélection que l'on pourrait qualifier de *fondamentale* dans le style de programmation le plus couramment employé aujourd'hui et que l'on nomme le *style impératif*. Pour cela, nous verrons, au chapitre 1, les calculs simples (essentiellement arithmétiques) et la possibilité de les systématiser en leur donnant un nom (par exemple `perimetre` pour le calcul du périmètre d'un rectangle), un ou plusieurs paramètres (par exemple la `longueur` et la `largeur` du rectangle) ainsi qu'une définition (l'expression de calcul, en l'occurrence `2 * (longueur + largeur)`. Nous aborderons les calculs conditionnels avec les alternatives (le célèbre `if ... else`) au chapitre 2.

Les calculs répétitifs, bien plus complexes *et* intéressants sont abordés au chapitre 3 avec les boucles, et approfondis au chapitre 4. En particulier nous abordons les problématiques de correction, de terminaison et d'efficacité des programmes. Nous introduisons les concepts d'*invariant* (de correction pour les boucles) et de *variant* (de boucle pour la terminaison). Ces concepts sont véritablement et incontestablement *fondamentaux* en programmation, malheureusement bien trop souvent écartés de l'enseignement de la programmation. Nous aurons couvert, ici, les éléments *de base* de la programmation. Il s'agit d'une base commune à la plupart des langages (dits impératifs) de programmation, mais elle n'est pas suffisante pour permettre l'expression «confortable» des programmes.

A partir du chapitre 5 nous abordons des questions plus «informatiques» (comprendre «moins arithmétiques») avec notamment les *chaînes de caractères*. On y discute également la notion très utile de *boucle d'itération*. On enchaîne au chapitre 6 avec l'introduction des listes, une des structures de données les plus couramment employées dans le langage Python. Au chapitre 7 sont introduits les *n-uplets*, ainsi que leur utilisation comme éléments de listes. Au chapitre 8, nous exploitons une spécificité du langage Python : les expressions de compréhensions. Ces dernières permettent de résoudre une large gamme de problèmes de constructions de liste de façon élégante et concise. Les *ensembles* (au sens mathématique du terme), disponibles intrinsèquement en Python, sont présentés au chapitre 9. Les *dictionnaires*, l'autre structure de donnée phare de Python (avec les listes), sont présentés au chapitre 10. Et au chapitre 11 nous complétons notre présentation des expressions de compréhension, enrichie par la notion de structure (de donnée) itérable.

Le chapitre 12 est en quelque sorte en rupture avec les précédents chapitres. On ajoute une simple construction : le *remplacement* d'un élément de liste à un indice donné, ainsi qu'un concept : la notion de procédure. Ces petites «perturbations» linguistiques ont un impact très important, en matière de programmation bien sûr mais également d'un point de vue plus méthodologique. Les procédures, de fait, contredisent la démarche de décomposition fonctionnelle de problèmes puisqu'elles ne sont *pas*, justement, des

fonctions. Les programmes dits *procéduraux* sont aussi plus difficiles à tester et à mettre au point. Mais il y a une contrepartie positive : un gain potentiel en terme d'efficacité.

Finalement, au chapitre 13, nous abordons la *programmation orientée objet*. On peut trouver étonnant que, dans un ouvrage consacré essentiellement au langage Python, l'on ne trouve une explication d'une construction essentielle de ce langage – la *classe* et ses *objets* (instances) – qu'au dernier chapitre. La raison est simple : les concepts objets sont des concepts organisationnels des programmes (de même que les modules que nous ne présentons pas dans ce livre) et non des éléments fondamentaux au sens de la résolution (algorithmique) des problèmes.

Ceci illustre le positionnement de notre livre, qui n'est pas un manuel du langage Python mais plutôt un apprentissage des bases de la programmation, illustré dans ce langage : un livre *en* Python plutôt qu'un livre *sur* Python. Cependant, le sous-ensemble de Python que nous présentons est *complet*, dans le sens où rien ne manque pour résoudre les problèmes de même nature que ceux abordés dans ce livre. De plus, nous pensons (peut-être de façon légèrement présomptueuse) que les programmes proposés dans ce livre (notamment les corrections d'exercices) sont de *beaux* programmes : lisibles, documentés et proprement testés. Notre but est d'enseigner la «belle» programmation, celle qui conduit à des programmes fonctionnels, robustes et correctement testés.

. . . en Python

Dans ce livre nous ne couvrons donc pas l'intégralité des construction du langage Python. Cependant, si une partie des concepts étudiés (notamment ceux couverts par les chapitres 1 à 5) sont facilement transposables à d'autres langages, nous abordons tout de même des constructions spécifiques au langage Python. Ceci concerne notamment les expressions de compréhension qui nous semblent particulièrement intéressantes car illustrant parfaitement le concept d'*expressivité*. Il est possible de programmer en Python sans les exploiter, mais pour de nombreux problèmes le gain en clarté et en concision est indéniable. Autrement dit, il serait *dommage* de s'en passer.

Un choix qui fera peut-être «bondir» certains experts du langage Python (que l'on nomme parfois *les pythonistas*) concerne notre utilisation des annotations de type, récemment standardisées par le document de spécification PEP-484 [3]. Ce document introduit un *système de type* permettant d'annoter les programmes Python. D'un point de vue méthodologique, cette nouvelle addition au langage est très intéressante car elle permet d'encadrer la rédaction des programmes, et de grandement faciliter le lien entre le problème posé, l'algorithme pour le résoudre et finalement le programme exécutable par la machine. Certes, la possibilité de *ne pas* annoter les programmes, et de se reposer sur ce que l'on nomme le *typage dynamique*, augmente les possibilités du langage et aussi sûrement le confort de programmation, mais uniquement dans le cadre de *programmes complexes* pour lesquels un niveau d'expertise élevé est requis. Les auteurs de cet ouvrage, également enseignants, le répètent souvent à leurs étudiants : un (bon) programmeur connaît le type des informations qu'il manipule, que ce type soit explicite (statique) ou

3. Cf. https://www.python.org/dev/peps/pep-0484/.

implicite (dynamique). Avant de devenir expert, et de comprendre *quand* et *pourquoi* on peut se passer des annotations de types, nous recommandons donc de les exploiter de façon systématique. Pour le langage couvert par ce livre, on y gagne également une aide incontestable : un vérificateur automatique de types, soit par l'intermédiaire de l'outil *mypy*[4] plutôt pour l'expert Python, soit avec notre propre environnement *MrPython* (présenté ci-dessous).

À qui s'adresse ce livre ?

Arrivés à ce stade, nous pouvons *enfin* vous expliquer, chère lectrice, cher lecteur, ce que nous attendons de vous, et également ce que nous n'attendons pas de vous bien sûr ! Tout d'abord, en terme de programmation, ce livre n'impose *aucun* pré-requis. Une première approche d'un langage de programmation n'est bien sûr pas inutile, loin de là, en particulier s'il s'agit du langage Python. Mais cette prédisposition n'est pas nécessaire pour entamer la lecture du présent ouvrage. Notre expérience nous montre, d'ailleurs, que pour pouvoir acquérir les bonnes connaissances il est parfois nécessaire de *désapprendre* certaines pratiques «discutables» (notamment la volonté de vouloir écrire le programme le plus court – et le plus illisible – possible, ou encore la volonté de vouloir se démarquer par un style de programmation «personnalisé» que bien souvent personne d'autre ne pourra ou ne voudra relire).

Il n'y a donc pas de pré-requis de programmation pour ce livre, mais en revanche on supposera un certain niveau de familiarité avec l'ordinateur et la pratique des systèmes d'exploitation modernes : Windows, MacOs ou Linux en particulier. Nous n'expliquons pas, par exemple, comment installer l'environnement Python. De nombreux guides existent pour cela, que ce soit des documents textuels[5] ou des tutoriels en vidéo. Les programmes discutés dans ce livre sont compatibles avec la version 3.6 (minimum) du langage Python.

Finalement, on nous a parfois reproché le contenu «trop mathématiques» de notre support pédagogique. Il est vrai que nous nous basons sur un certain nombre de pré-requis mathématiques, correspondant *grosso modo* au niveau de la première (avec option scientifique) voire de la terminale au Lycée. La raison est en fait évidente : l'information «la plus simple» manipulée par un ordinateur est le *nombre*. Ainsi, il faut se rendre à l'évidence, l'arithmétique occupe un place essentielle en informatique en générale, et en programmation en particulier. Est-il possible de parler de programmation sans passer par une maîtrise des calculs arithmétiques sur ordinateur ? Pour nous, la réponse est clairement *non*. Comment pourrait-on par exemple parler de l'indice d'un élément dans une liste en faisant abstraction du fait qu'il s'agit d'une valeur numérique résultant donc d'un calcul arithmétique ? Se placer *au milieu* d'une liste, est par exemple un problème de division entière ! Il en est de même pour la logique dite *booléenne* : les fameux `True` (vrai) et `False` (faux) ainsi que les opérations associées (négation, conjonction, disjonction) sont incontournables. Cependant, dès que cela est possible (en pratique,

4. Cf. http://mypy-lang.org/.

5. Le document standard n'est peut-être pas le plus simple à suivre, mais cela semble tout de même un incontournable, cf. https://docs.python.org/fr/3/using/index.html.

au chapitre 5 avec les chaînes de caractères), nous essayons d'aborder des problèmes qui ne sont pas directement arithmétiques , et que l'on peut donc qualifier de «plus informatiques». Mais il ne faut pas se tromper, l'arithmétique joue toujours un rôle : chaque caractère de la chaîne est codé par un nombre (appelée Unicode) situé à une position dans la chaîne qui est également un nombre. Ceci ne veut pas dire que la programmation ne peut être abordée qu'après un apprentissage de l'arithmétique, elle peut bien sûr se faire *en parallèle* (mais notre ouvrage ne peut pas couvrir les deux problématiques). Bref, n'ayons pas peur des mathématiques !

MrPython et ressources en ligne

Nous proposons un environnement de programmation Python à vocation pédagogique, gratuit et *open source*, et adapté à cet ouvrage : *MrPython*[6]. Le principal intérêt de cet environnement concerne les nombreuses vérifications faites sur les programmes, en particulier concernant les erreurs de typage que l'outil vérifie statiquement, avec des messages clairs et précis (et en français) pour expliquer ces erreurs. Un site compagnon de l'ouvrage est également proposé[7]. On y trouvera notamment les programmes Python corrections des exercices de l'ouvrage. Des exercices supplémentaires (et non corrigés) sont également proposés.

Remerciements

Les auteurs tiennent à remercier tous les collègues qui ont participé à l'élaboration du cours «Éléments de programmation» dispensé depuis 2015 à Sorbonne Université. On citera notamment (dans le désordre) : Maryse Pelletier, Christophe Marsala, Antoine Genitrini, Clémence Magnien, Isabelle Mounier, Marie-Jeanne Lesot, Karine Heydemann, Fabien Tarissan, Choun Tong Lieu, Mathilde Carpentier et Pascal Manoury. Nous remercions chaleureusement nos *mentors* Christian Queinnec et Michèle Soria qui ont gentiment accepté de préfacer cet ouvrage. Nous remercions également tous les enseignants et étudiants, qui, de gré ou de force, participent à notre cours chaque année : tout ceci n'existerait pas sans eux ! Nous remercions finalement les éditions Ellipses qui permettent à nos cours de se diffuser au-delà de notre «petite» Université.

6. Le logiciel MrPython est disponible à l'adresse suivante : https://github.com/nohtyprm/MrPython.
7. Cf. https://github.com/elemprog-python.

Chapitre 1

Premiers pas

Dans ce chapitre, nous abordons nos premiers *éléments de programmation*, à partir d'un exemple de programme Python court (une seule fonction) mais complet. Parmi ces *éléments* nous étudierons dans un premier temps les notions fondamentales d'*expression* et de *fonction*.

1.1 Un exemple de programme

On donne ci-dessous le code Python d'une fonction dont le nom est `liste_premiers` et qui permet de calculer, pour un entier naturel `n` fixé, la liste triée des nombres premiers [1] inférieurs (strictement) à `n`.

Voici le code de cette fonction :

```
from typing import List

def liste_premiers(n : int) -> List[int]:
    """Précondition : n >= 0
       Retourne la liste des nombres premiers inférieurs à n.
    """

    i_est_premier : bool = False  # indicateur de primalité

    l : List[int] = []   # liste des nombres premiers en résultat

    i : int # Entier courant
    for i in range(2, n):
        i_est_premier = True
```

1. On rappelle qu'un nombre est premier s'il n'est divisible que par 1 ou par lui-même.

```
        j : int # Candidat diviseur
        for j in range(2, i - 1):
            if i % j == 0:
                # i divisible par j, donc i n'est pas premier
                i_est_premier = False

        if i_est_premier:
            l.append(i)

    return l
```

On ne cherche pas, pour l'instant, à comprendre ce que fait cette fonction (il faudra pour cela atteindre le chapitre 5, au minimum). Notre objectif est d'extraire un petit catalogue (non-exhaustif mais significatif) des *éléments de programmation* que cette fonction met en œuvre.

Avant cela, nous pouvons peut-être *tester* notre fonction dans l'interprète de commandes de Python :

```
>>> liste_premiers(30)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Voici quelques *éléments de programmation* à l'œuvre dans le programme ci-dessus :

— **Définitions de fonction**

La construction :

```
def liste_premiers(n : int) -> List[int]:
    ... etc ...
```

est une définition de fonction dont le principe est de nommer un calcul. Ici le calcul concerne la construction d'une liste des nombres premiers, et le nom choisi pour la fonction, `liste_premiers`, semble ainsi parfaitement adapté. Ce calcul est également *paramétré* par un entier `n` appelé un **paramètre formel**. Le rôle de ce paramètre est clairement expliqué dans la documentation de la fonction : elle «*retourne la liste des nombres premiers inférieurs à* `n`».

— **Variables**

L'identifiant `i_est_premier` est un nom de variable. Les variables s'apparentent à des cases mémoire que l'on identifie par un nom (plutôt que directement une adresse mémoire dans l'ordinateur). Elles sont abordées au chapitre 2.

— **Expressions**

On trouve de nombreuses expressions dans notre programme. C'est le cas par exemple de l'expression atomique `1`, l'expression arithmétique `i-1` ou encore l'expression arithmético-logique `i % j == 0`. Une expression dénote une *valeur* obtenue d'après un *principe d'évaluation*. Nous discutons des expressions et de leur évaluation un peu plus loin dans ce chapitre.

— **Instructions**

Une instruction est, de façon simplifiée, un *ordre* donné à l'ordinateur (dans notre cas, plus précisément, à l'interprète Python) dans le but de réaliser une *action*.

Par exemple, la construction `i_est_premier = True` est une instruction dite d'*affectation* à la variable de nom `i_est_ premier`. L'action réalisée est une écriture dans la mémoire.

La construction :

```
if i_est_premier:
   ... etc ...
```

est une instruction dite *alternative*. Les alternatives permettent les calculs conditionnés, et seront vues au chapitre 2.

La construction :

```
for i in range(1, n):
   ... etc ...
```

est une instruction dite de *boucle* (ici une boucle d'itération), que l'on aborde au chapitre 3.

1.2 Notion d'expression

Nous allons commencer par expliquer de façon un peu plus approfondie le premier type fondamental d'élément de programmation que l'on nomme **expression**. Une expression en informatique est une écriture formelle textuelle que l'on peut saisir au clavier. Le langage des expressions que l'on peut saisir est un langage informatique, beaucoup plus contraint que les langages naturels comme le Français par exemple. En effet, l'ordinateur doit «comprendre» ce que l'on écrit.

Dans ce livre, les expressions sont écrites dans le langage Python, mais il existe bien sûr de nombreux autres langages informatiques (C, C++, java, Ocaml, etc.), et dans tous ces langages la même notion d'expression est présente.

Les expressions sont principalement de deux types :

— **expressions atomiques** (ou **atomes**) : la forme la plus simple d'expression. On parle également d'*expressions simples*.
— **expressions composées** : expressions (plus) complexes composées de **sous-expressions**, elles-mêmes à nouveau simples ou composées.

La propriété fondamentale d'une expression est d'être *évaluable*, c'est-à-dire que chaque expression possède une **valeur** que l'on peut obtenir par calcul. Ce calcul, effectué par la machine, est appelé un **principe d'évaluation**.

1.2.1 Expressions atomiques

Une expression atomique `v` possède un type `T` et - c'est ce qui caractérise la propriété d'atomicité - sa valeur est également notée `v`.

Considérons l'interaction suivante avec l'interprète Python :

```
>>> 42
42
```

Ici, on a saisi l'entier `42` au clavier et Python nous a répondu également `42`. Le type de cette expression atomique est `int`, le type des nombres entiers. Nous pouvons vérifier ce fait par la fonction prédéfinie `type` qui retourne la description du type d'une expression.

```
>>> type(42)
int
```

Le processus d'évaluation mis en jeu pour évaluer l'expression `42` est plus complexe qu'il n'y paraît. Lorsque l'on tape `42` au clavier et qu'on soumet cette expression à l'*évaluateur* de Python, ce dernier doit transformer ce texte en une valeur entière représentable sur ordinateur. Cette traduction est assez complexe, il faut notamment représenter `42` par une suite de 0 et de 1 - les fameux *bits* d'information - au sein de l'ordinateur (on parle alors de *codage binaire*). Cette représentation dans la machine se nomme en python un **objet**. On dit que le *type de l'expression* `42` est `int` mais pour la représentation interne, on dit que la représentation de 42 en mémoire est un objet instance de la **classe** `int`.

Cette terminologie fait de Python un **langage objet** et nous reviendrons sur ce point mais pas avant le dernier chapitre du livre (et nous expliquerons pourquoi ce n'est pas un thème central de ce livre).

Le processus dit d'**auto-évaluation** des expressions atomiques n'est pas terminé. Dans un second temps, Python nous «explique» qu'il a bien interprété l'expression saisie en produisant un affichage de la valeur. Dans cette deuxième étape l'objet en mémoire qui représente 42 est converti en un texte finalement affiché à l'écran.

Pour résumer, nous avons fait la distinction entre :

- une *expression d'un type donné*, saisie par le programmeur, par exemple l'expression `42` de type `int`,
- la valeur de l'expression : un *objet représenté en mémoire* d'une *classe* donnée, par exemple la représentation interne (codée en binaire) de l'entier 42, objet de la classe `int`,
- *l'affichage de cet objet en sortie*, par exemple la suite de symboles `42` affiché par l'interprète Python.

Avec un peu de pratique, le programmeur ne voit qu'un seul 42 à toutes les étapes mais il faut être conscient de ces distinctions pour comprendre pourquoi et comment l'ordinateur est capable d'effectuer *nos* calculs.

Retenons donc **le principe simplifié d'évaluation des expressions atomiques**:

> *Une expression atomique s'évalue en elle-même, directement, sans calcul ni travail particulier.*

Effectuons maintenant un tour d'horizon des principales expressions atomiques fournies par Python.

1.2.1.1 Les constantes logiques (ou booléens)

La vérité logique est représentée par l'expression `True` qui signifie *vrai* et l'expression `False` qui signifie *faux*. Ces deux atomes forment le type booléen noté `bool` du nom du logicien *George Boole* qui, au XIXème siècle, a établi un lien fondamental entre la logique et le calcul.

```
>>> type(True)
bool
```

```
>>> type(False)
bool
```

Nous étudions les booléens de façon plus approfondie au chapitre 2.

1.2.1.2 Les entiers

Les entiers sont écrits en notation mathématique usuelle.

```
>>> type(4324)
int
```

Une remarque importante est que les entiers Python peuvent être de taille arbitraire.

```
>>> 232392873298372983827392847398473948374394873984792837293823922832323928732983729838273928473984739483743948739847928372938239228323928732983729838273928473984739483743948739847928372938239228
```

Dans beaucoup de langages de programmation (exemple : le langage C) les entiers sont ceux de l'ordinateur et le résultat aurait donc été tronqué.

Sur une machine 32 bits, l'entier (signé) maximal que l'on peut stocker dans une seule case mémoire est 2^{31}.

```
>>> 2 ** 31   #  l'opérateur puissance se note **  en python.
2147483648
```

Ceci illustre un aspect important de Python : l'accent est mis sur la précision et la généralité des calculs plutôt que sur leur efficacité. Le langage C, par exemple, fait plutôt le choix opposé de se concentrer sur l'efficacité au détriment de la précision et de la généralité. On dit que Python est (plutôt) un *langage de haut-niveau*, et que le langage C est (plutôt) un *langage de bas-niveau*.

1.2.1.3 Les constantes à virgule flottante.

Les expressions atomiques `1.12` ou `-4.3e-3` sont de type flottant noté `float`.

Les flottants sont des approximations informatiques des *nombres réels* de $\mathbb{R}$, qui eux n'existent qu'en mathématiques. Dans ce livre d'introduction, nous ne nous occuperons pas trop des problèmes liés aux approximations – du fait de leur complexité – mais nous discutons un peu de cette problématique au chapitre 3.

```
>>> type(-4.3e-3)
float
```

Il est important de remarquer que les entiers et les flottants sont des types *disjoints* mais comme beaucoup d'autres langages de programmation, Python convertit implicitement les entiers en flottants si nécessaire.

Par exemple :

```
>>> type(3 + 4.2)
float
```

Ici `3` est un entier de type `int` et `4.2` est de type `float`. Le résultat de l'addition privilégie les flottants puisqu'en effet on s'attend au résultat suivant :

```
>>> 3 + 4.2
7.2
```

Remarque : lors de certains calculs, on ne veut pas distinguer entre entiers et flottants (par exemple le calcul de la valeur absolue). Dans ce cas on utilisera le type `float`. En effet, dans la spécification des types pour Python [2], les entiers du type `int` doivent être acceptés en remplacement de `float`. C'est un peu comme l'ensemble des entiers que l'on peut considérer en mathématiques comme sous-ensemble des nombres réels.

1.2.1.4 Les chaînes de caractères

Les chaînes de caractères de type `str` ne sont pas à proprement parler atomiques, mais elles s'évaluent de façon similaire.

Une chaîne de caractères est un texte encadré :

— soit par des apostrophes ou guillemets simples `'...'` :

```
>>> 'une chaîne entre apostrophes'
'une chaîne entre apostrophes'
```

— soit par des guillemets à l'anglaise ou guillemets doubles `"..."` :

```
>>> "une chaîne entre guillemets"
'une chaîne entre guillemets'
```

2. Cf. PEP 484 https://www.python.org/dev/peps/pep-0484/.

On remarque que Python privilégie les apostrophes pour les affichages des chaînes de caractères. Ceci nous rappelle d'ailleurs bien ici le processus en trois étapes : lecture de l'expression (avec guillemets doubles), conversion en un objet en mémoire, puis écriture de la valeur correspondante (avec guillemets simples).

1.2.2 Expressions composées

Les expressions composées sont formées de combinaisons de sous-expressions, atomiques ou elles-mêmes composées.

Pour ne pas trop charger ce premier chapitre nous nous limiterons dans nos exemples aux expressions arithmétiques, c'est-à-dire aux expressions usuelles des mathématiques. Nous aborderons d'autres types d'expressions lors des prochains chapitres, mais il faut retenir que la plupart des concepts étudiés ici dans le cadre arithmétique restent valables dans le cadre plus général.

Pour composer des expressions arithmétiques, le langage fournit diverses constructions, notamment :

- les expressions atomiques d'entiers et de flottants vues précédemment
- des opérateurs arithmétiques `+`, `-`, `*`, etc.
- des applications de fonctions prédéfinies en langage Python ou définies par le programmeur.

1.2.2.1 Opérateurs arithmétiques

Le langage Python fournit la plupart des opérateurs courants de l'arithmétique :

- les opérateurs *binaires* d'addition, de soustraction, de multiplication et de division
- l'opérateur *moins unaire*
- le parenthésage

La notation utilisée suit l'usage courant des mathématiques.

Par exemple, on peut calculer de tête assez rapidement les expressions suivantes :

```
>>> 2 + 1
3
```

```
>>> 2 + 3 * 9
29
```

```
>>> (2 + 3) * 9
45
```

```
>>> (2 + 3) * -9
-45
```

Remarque importante sur la division : en informatique on distingue généralement deux types de division entre nombres :

1. la division entière ou *euclidienne* qui est notée `//` en Python
2. la division flottante qui est notée `/`.

Voici quelques exemples illustratifs.

```
>>> 7.0 / 2.0
3.5
```

Ici on a divisé deux flottants par la division flottante. Le résultat est aussi un flottant. Divisons maintenant dans les entiers :

```
>>> 7 // 2
3
```

Ici le résultat est bien un entier : 3 est le quotient de la division entière de 7 par 2. Nous pouvons d'ailleurs obtenir le reste de la division entière avec l'opérateur *modulo* (qui est noté `%` en Python).

```
>>> 7 % 2
1
```

Le reste de la division entière de 7 par 2 est bien 1, on a : 7 qui vaut $2 * 3 + 1$

Mais maintenant, que se passe-t-il si on utilise la division flottante pour diviser des entiers ?

```
>>> 7 / 2
3.5
```

Puisque la division flottante nécessite des opérandes de type `float`, l'entier 7 a été converti implicitement en le flottant 7.0 et l'entier 2 a été converti en le flottant 2.0. C'est donc *presque* sans surprise que le résultat produit est bien le flottant 3.5.

On retiendra de cette petite digression que les divisions informatiques ne sont pas simples.

Priorité des opérateurs

Les règles que nous appliquons implicitement dans nos calculs mentaux doivent être explicitées pour l'ordinateur. Pour cela, les opérateurs sont ordonnés par priorité.

Retenons **les règles de base de priorité des opérateurs** :

— la multiplication et la division sont prioritaires sur l'addition et la soustraction
— le moins unaire est prioritaire sur les autres opérateurs
— les sous-expressions parenthésées sont prioritaires

Dans l'expression `2 + 3 * 9` précédente, on évalue la multiplication *avant* l'addition. Le processus de calcul est donc le suivant :

```
2 + 3 * 9
==> 2 + 27
==> 29
```

Pour *forcer* une priorité et changer l'ordre d'évaluation, on utilise, comme en mathématiques, des parenthèses. On a ainsi :

```
(2 + 3) * 9
==> 5 * 9
==> 45
```

Exercice - Donner la valeur des expressions suivantes :

- `4 + 7 * 3 - 9 * 2`
- `(4 + 7) * 3 - 9 * 2`
- `4 + 7 * (3 - 9) * 2`
- `4 + (7 * 3 - 9) * 2`
- `4 + (7 * - 3 - 9) * 2`
- `4 + 3 / 2`
- `4 + 3 // 2`

1.2.2.2 Applications de fonctions prédéfinies

Le langage Python fournit un certain nombre de fonctions prédéfinies (en fait plusieurs centaines !). Les fonctions prédéfinies dites *primitives* sont accessibles directement.

Pour pouvoir utiliser une fonction, prédéfinie ou non, il nous faut sa **spécification**.

Prenons l'exemple de la fonction `max` qui retourne le maximum de deux nombres et qui pourrait être spécifiée de la façon suivante[3] :

```
def max(a : int, b : int) -> int:
  """Retourne le maximum des nombres a et b."""
```

Par exemple :

```
>>> max(2,5)
5

>>> max(-5.12, -7.4)
-5.12
```

Dans la spécification de fonction, par exemple `max`, on trouve des informations essentielles, notamment :

- la **signature de la fonction**, `def max(a : int, b : int) -> int`, qui donne le *nom de la fonction* (`max`) et de ses *paramètres formels* (`a` et `b`). Elle indique

3. En pratique, la spécification de la fonction `max` est plus complexe et ne s'applique pas qu'aux entiers, mais elle reste compatible avec la spécification proposée ici.

aussi les *types des paramètres* ainsi que le *type de retour* de la fonction. Ici, il est indiqué que le premier paramètre (`a`) est de type `int`, le second paramètre (`b`) également de type `int` et le type de retour indiqué après la flèche `->` est également `int`. Ainsi la fonction prend deux valeurs entières en entrée, et retourne un entier en sortie. Une notation alternative pour cette signature est la suivante :
`int * int -> int`
(l'étoile `*` représente ici le produit cartésien qui sépare les types des paramètres).

— la **documentation de la fonction** qui explique, de façon textuelle, le problème résolu par la fonction. Ici, la fonction *«retourne le maximum des nombres a et b»*.

Nous reviendrons longuement dans ce livre sur ce concept fondamental de spécification.

Le **principe d'évaluation des appels de fonctions prédéfinies** est expliqué ci-dessous :

Pour une expression de la forme `fun(arg1, ..., argn)`

— on évalue d'abord les expressions en arguments d'appels: `arg1, ..., argn`
— puis on remplace l'appel par la valeur calculée par la fonction.

Considérons l'exemple suivant :

```
max(2+3, 4*12)
==> max(5, 48)   [évaluation des arguments d'appel]
==> 48           [remplacement par la valeur calculée]
```

Un second exemple :

```
3 + max(2, 5) * 9
```

L'opérateur de multiplication est prioritaire mais on doit d'abord évaluer son premier argument, c'est-à-dire l'appel à la fonction `max`. Cela donne :

```
==> 3 + 5 * 9
```

Et on procède comme précédemment :

```
==> 3 + 45
==> 48
```

Ce que l'on vérifie aisément avec l'interprète python :

```
>>> 3 + max(2, 5) * 9
48
```

Python fournit de nombreuses *bibliothèques de fonction prédéfinies* que l'on nomme des **modules**.

Pour ce qui concerne l'arithmétique, la plupart des fonctions intéressantes se trouvent dans le module `math`.

Pour pouvoir utiliser une fonction qui se trouve dans une bibliothèque, il faut tout d'abord *importer* le module correspondant à cette bibliothèque. Par exemple, pour importer le module `math` on écrit :

```
import math # bibliothèque mathématique
```

Prenons l'exemple de la fonction de calcul du *sinus* d'un angle, dont la spécification est la suivante :

```
# fonction présente dans le module math
def sin(x : float) -> float:
    """Retourne le sinus de l'angle x exprimé en radians."""
```

Attention : la fonction `sin` du module `math` s'appelle précisément `math.sin`.

Voici quelques exemples d'utilisation.

```
>>> math.sin(0)
0.0
```

```
>>> math.sin(math.pi / 2)
1.0
```

On remarque ici que la constante π est aussi définie dans le module `math` et qu'elle s'écrit `math.pi` en Python.

```
>>> math.pi
3.141592653589793
```

1.2.2.3 Principe d'évaluation des expressions arithmétiques

Nous pouvons maintenant résumer **le principe d'évaluation des expressions arithmétiques**.

Pour évaluer une expression e:

1. si e est une expression atomique alors on termine l'évaluation avec la valeur e.
2. si e est une expression composée alors :
   a. on détermine la sous-expression e' de e à évaluer en premier (cf. règles de priorité des opérateurs).
      — si e' est unique (il n'y a qu'une seule sous-expression prioritaire) alors on lui applique le principe d'évaluation (on passe à l'étape 1 donc).
      — s'il existe plusieurs expressions à évaluer en premier, alors on procède de «gauche à droite» par priorité.
   b. on réitère le processus d'évaluation jusqu'à sa terminaison (obtention d'une expression atomique, auto-évaluée).

Considérons l'exemple suivant :

```
(3 + 5) * ( max(2, 9) - 5 ) - 9
```

Il s'agit bien sûr d'une expression composée. Les sous-expressions prioritaires sont entre parenthèses (le parenthésage est toujours prioritaire). Il y en a deux ici : `(3 + 5)` et `( max(2, 9) - 5 )`. On procède de gauche à droite donc on commence par la première :

```
==> 8 * ( max(2, 9) - 5 ) - 9    [parenthésage - gauche]
```

La deuxième sous-expression prioritaire est la soustraction dont on évalue les arguments de gauche à droite, en commençant donc par l'application de la fonction `max` :

```
==> 8 * ( 9 - 5 ) - 9          [application de la fonction max]
```

Maintenant on peut évaluer la soustraction prioritaire car entre parenthèses :

```
==> 8 * 4 - 9                  [soustraction entre parenthèses]
```

On s'occupe ensuite de la multiplication prioritaire :

```
==> 32 - 9                     [multiplication prioritaire]
```

Et finalement la dernière soustraction est possible :

```
==> 23                         [soustraction]
```

```
>>> (3 + 5) * ( max(2, 9) - 5 ) - 9
23
```

Ces principes d'évaluation peuvent paraître un peu complexes mais la plupart du temps ils sont en fait assez proches de notre propre manière de calculer.

1.3 Définition de fonctions simples

Considérons le problème suivant :

> Calculer le périmètre d'un rectangle défini par sa `largeur` et sa `longueur`.

La solution mathématique du problème consiste à calculer la valeur de l'expression suivante :

```
2 * (largeur + longueur)
```

Par exemple, si on souhaite calculer en Python le *périmètre* d'un rectangle de `largeur` valant 2 unités (des mètres par exemple) et de `longueur` valant 3 unités, on saisit l'expression suivante :

```
>>> 2 * (2 + 3)
10
```

Le périmètre obtenu est donc de 10 unités.

Maintenant si l'on souhaite calculer le périmètre d'un rectangle de `largeur` valant 4 unités et de `longueur` valant 9 unités, on saisit alors l'expression :

```
>>> 2 * (4 + 9)
26
```

Le premier usage que nous ferons des fonctions est de permettre de paramétrer, et donc de généraliser, les calculs effectués par une expression arithmétique.

Pour le calcul du périmètre, une traduction presque directe est possible en Python :

```
def perimetre(largeur : int, longueur : int) -> int:
    """Précondition : (longueur >= 0) and (largeur >= 0)
    Précondition : longueur >= largeur
    retourne le périmètre du rectangle défini par
    sa largeur et sa longueur.
    """
    return 2 * (largeur + longueur)
```

Une fois écrite (et avant de l'étudier un peu plus loin), on peut interagir avec cette fonction avec quelques *applications* :

```
>>> perimetre(2, 3)
10

>>> perimetre(4, 9)
26
```

De ces exemples d'application de la fonction `perimetre` nous pouvons déduire un **jeu de tests** nous servant à *valider* la fonction [4].

```
# Jeu de tests
assert perimetre(2, 3) == 10
assert perimetre(4, 9) == 26
assert perimetre(0, 0) == 0
assert perimetre(0, 8) == 16
```

Remarque : le mot-clé `assert` permet de définir un test. Le premier test ci-dessus signifie *«Le programmeur de la fonction certifie que le périmètre d'un rectangle de largeur 2 et de longueur 3 vaut 10»*.

Si une telle assertion ne correspond pas à la définition de fonction, une exception est levée et une erreur se produit. Par exemple :

```
>>> assert perimetre(2, 3) == 12
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
  ...
----> 1 assert perimetre(2, 3) == 12

AssertionError:
```

Illustrons un autre intérêt de bien spécifier les fonctions :

```
>>> help(perimetre)
Help on function perimetre in module __main__:

perimetre(largeur:int, longueur:int) -> int
    Précondition : (longueur >= 0) and (largeur >= 0)
    Précondition : longueur >= largeur
```

4. En Python «professionnel», plutôt que d'écrire de simples assertions – largement suffisantes dans le cadre de ce livre – on utilise des outils dédiés au test logiciel. On peut citer notamment le module `unittest` pour le *test unitaire*, cf. https://docs.python.org/fr/3/library/unittest.html.

```
    retourne le périmètre du rectangle défini par
    sa largeur et sa longueur.
```

La commande `help` de l'interprète Python permet d'obtenir la documentation des fonctions, ce qui renforce l'intérêt de fournir des descriptions claires et précises.

Détaillons le processus qui conduit du problème de calcul de périmètre à sa solution informatique sous la forme de la fonction `perimetre`.

Les trois étapes fondamentales sont les suivantes :

1. La **spécification** du problème posé et devant être résolu par la fonction.
2. L'**implémentation** dans le **corps de la fonction** de l'**algorithme** de calcul fournissant une solution au problème posé.
3. La **validation** de la fonction par le biais d'un **jeu de tests**.

Détaillons ces trois étapes pour la fonction `perimetre`.

1.3.1 la spécification de la fonction

La spécification d'une fonction permet de décrire le problème que la fonction est censée résoudre. Cette spécification s'énonce dans un cadre standardisé pour ce livre.

Le rôle fondamental de la spécification est de permettre à un programmeur de comprendre comment utiliser la fonction sans avoir besoin d'une lecture détaillée de son code d'implémentation en Python. On retiendra la maxime suivante :

> *On n'écrit pas une fonction uniquement pour soi, mais également et surtout pour toute personne susceptible de l'utiliser dans un futur plus ou moins proche (soit en appelant la fonction, soit en lisant son code source pour la comprendre).*

La partie spécification de la fonction `perimetre` est répétée ci-dessous.

```
def perimetre(largeur : int, longueur : int) -> int:
    """Précondition : (longueur >= 0) and (largeur >= 0)
    Précondition : longueur >= largeur

    retourne le périmètre du rectangle défini par
    sa largeur et sa longueur.
    """
```

La ligne :

```
def perimetre(largeur : int, longueur : int) -> int:
```

se nomme la **signature** de la fonction. Son rôle est de donner *le nom* de la fonction (qui est souvent à la fois le nom porté par le problème et sa solution), ici `perimetre`, ainsi que les noms et les types de ses *paramètres formels* (ou plus simplement *paramètres* tout court). Pour notre problème de périmètre, les paramètres se nomment naturellement `largeur` et `longueur` et sont tous les deux entiers (type `int`). Le type de la *valeur de retour* de la fonction se situe à droite de la flèche `->` est également de type `int`.

En Python les annotations de type sont optionnelles, mais on verra que la signature joue un rôle essentiel notamment lorsque l'on manipule des types de données complexes comme les listes, les ensembles ou les dictionnaires. C'est pour cela que dans ce livre nous utilisons la version typée du langage Python. La signature de la fonction peut également être résumée de la façon suivante :

```
perimetre : int * int -> int
```

Dans cette écriture compacte, on retient uniquement que la fonction possèdent deux paramètres entiers, et retourne un entier en sortie (on rappelle que l'étoile `*` représente ici le produit cartésien). On peut appeler cette écriture *le type de la fonction* `perimetre`.

La partie de la spécification en dessous de l'en-tête, quie se trouve entre des triples guillemets `""" ... """`, est ce que l'on nomme en Python la *docstring* (chaîne de caractères de documentation). En Python, le format de la *docstring* n'est pas imposé, l'objectif est simplement de bien expliquer ce que fait la fonction. Dans ce livre, nous adoptons un format assez précis, que nous expliquons ci-dessous.

> **Important** : Python impose les indentations dans les programmes. Ainsi, après l'en-tête de fonction, il faut indenter par 4 espaces les lignes qui composent la spécification et le corps de la fonction.

Pour la première partie de la *docstring*, nous complétons, lorsque cela est nécessaire, la signature de la fonction par des **préconditions** permettant de préciser les domaines de valeurs possibles pour les paramètres.

La fonction `perimetre` indique deux préconditions :

1. La largeur et la longueur doivent être positives, ce qui correspond à l'expression booléenne : `(longueur >= 0) and (largeur >= 0)`
2. La largeur doit être inférieure ou égale à la longueur : `largeur <= longueur`.

Lorsque cela est possible, nous exprimons ces préconditions sous la forme d'expressions logiques Python, c'est-à-dire de type `bool`, qui s'évaluent donc soit en `True`, soit en `False`. Nous revenons sur les expressions logiques avec l'alternative dans le chapitre suivant.

Finalement, on donne la **documentation** proprement dite de la fonction, c'est-à-dire la description en français du problème résolu par la fonction. L'objectif de cette description est d'indiquer le QUOI de la fonction, mais *sans* expliquer le COMMENT, c'est-à-dire sans préciser les détails des calculs à effectuer.

1.3.2 l'implémentation du corps de la fonction

L'implémentation de la fonction consiste à programmer en Python un algorithme de calcul permettant de résoudre le problème posé. Cette implémentation représente **le corps de la fonction** composé d'une *instruction* ou d'une suite d'instructions.

Dans le cadre des problèmes consistant à paramétrer une expression arithmétique, comme pour notre calcul de périmètre, le corps de la fonction se résume à une seule instruction de la forme suivante :

```
return <expression>
```

Cette instruction s'exécute ainsi :

1. évaluation de l'`<expression>`
2. retour de la fonction à son appelant avec la valeur calculée, dite *valeur de retour*.

Dans notre cas c'est bien sûr notre expression paramétrée par la largeur et la longueur du rectangle :

```
return 2 * (largeur + longueur)
```

Important : en règle générale, un même problème peut être résolu de différentes façons. Ainsi on parle de *la* spécification et *d'une* implémentation (possible) de la fonction. Par exemple, pour notre fonction de calcul du périmètre, nous pouvons proposer une implémentation différente :

```
return (largeur + longueur) + (largeur + longueur)
```

Pour l'instant nos définitions de fonctions sont très simples et concises en comparaison de la spécification du problème. Mais plus nous avancerons dans le livre, plus cette tendance s'inversera, avec des fonctions de plus en plus complexes.

1.3.3 la validation de la fonction par un jeu de tests

Il faut distinguer deux parties distinctes dans l'approche d'un problème informatique :

— Le *client* qui pose le problème à résoudre.
— Le *fournisseur* qui propose une solution informatique au problème posé par le client.

Pour chaque problème posé par le client, une définition de fonction est rédigée par le fournisseur.

On retiendra donc la maxime suivante :

> *un problème = une fonction pour le résoudre.*

Cependant, proposer une spécification et une définition de fonction pour résoudre le problème ne suffit pas pour ce que l'on appelle l'étape de *validation*, c'est-à-dire l'acceptation (ou non !) de la solution par le client.

Pour assurer cette validation, l'objectif est de proposer un **jeu de tests** sous la forme d'une suite d'applications de la fonction définie.

Une expression d'**application de fonction** (ou *expression d'appel*) est de la forme suivante :

```
nom_fonction(arg1, arg2, ...)
```

Il s'agit d'appeler la fonction en donnant des valeurs précises à ses paramètres. Les expressions qui décrivent la valeur prise par les paramètres se nomment les **arguments d'appel**.

Par exemple, dans notre premier test pour la fonction `perimetre` :

```
perimetre(2, 3)
```

— l'argument d'appel pour le paramètre `largeur` est l'expression `2`
— l'argument d'appel pour la `longueur` est l'expression `3`.

Si la définition de fonction répond bien au problème posé, la valeur retournée par l'expression d'application sera valide.

```
>>> perimetre(2, 3)
10
```

Les expressions `2` et `3` sont ici des expressions simples, mais on peut aussi utiliser des expressions de complexité arbitraire.

```
>>> perimetre(1 * 14 - 2 * 6, (3 * 121 // 11) - 30)
10
```

Question : que pensez-vous de l'application suivante ?

```
perimetre(2, 3.1)
```

On «casse» ici une **règle fondamentale** dans le cadre de ce livre (et plus généralement dans le cadre des *bonnes pratiques de programmation*) :

> *Les expressions d'application de fonction doivent respecter la signature et les préconditions spécifiées par la fonction. Dans le cas contraire, aucune garantie n'est fournie concernant les effets éventuellement désastreux de cette application de fonction erronée.*

Il est possible (voire probable) que Python «accepte» l'application de fonction et produise une valeur, mais cette valeur obtenue ne correspond à aucun problème posé, en particulier celui censé être résolu par la fonction.

Dans l'application ci-dessus, l'expression `3.1` de type `float` ne respecte pas la signature de la fonction `perimetre` qui indique que le second paramètre, la `longueur`, doit être de type `int`. On dit que cette expression n'est pas valide, et dans ce cas la meilleure correction est de tout simplement l'effacer ... ou la modifier pour la rendre valide.

Dans l'expression d'application ci-dessous :

```
perimetre(-2, 3)
```

l'erreur commise, qui rend l'expression invalide, est que le premier argument d'appel est négatif alors qu'une précondition de la fonction `perimetre` indique que les valeurs prises par le premier paramètre `largeur` doivent être positives.

Dans le même ordre d'idée, l'expression : `perimetre(3, 2)`

est invalide puisqu'elle contredit la précondition indiquant que la largeur doit être inférieure à la longueur.

On retiendra finalement que le jeu de tests permettant de valider la fonction doit :

— être composé uniquement d'expressions d'applications valides, qui respectent les signatures et préconditions de la fonction appelée
— couvrir suffisamment de cas permettant d'avoir confiance dans la validité de la solution proposée.

Bien sûr, c'est finalement le client qui décidera si la solution proposée est bien valide ou non.

L'écriture du jeu de test proprement dit se compose d'une suite d'assertions. Nous répétons ci-dessous le jeu de tests proposé pour la fonction périmètre.

```
# Jeu de tests
assert perimetre(2, 3) == 10
assert perimetre(4, 9) == 26
assert perimetre(0, 0) == 0
assert perimetre(0, 8) == 16
```

1.4 Exercices corrigés

Exercice 1.1 : Moyenne de trois nombres

Cet exercice a pour but de trouver les paramètres d'une fonction pour résoudre un problème, et d'écrire la spécification et la définition de fonctions très simples.

Question 1

Donner une définition de la fonction `moyenne_trois_nb` qui effectue la moyenne arithmétique de trois nombres flottants.

Dans le jeu de tests, on vérifiera notamment le calcul de moyenne des nombres 3, 6 et -3 puis de -3, 0 et 3 puis de 1.5, 2.5 et 1.0 (on pourra ajouter d'autres tests en complément).

Question 2 : moyenne pondérée

Écrire une définition de la fonction `moyenne_ponderee` qui effectue la moyenne de trois nombres flottants `a`, `b`, `c` avec des coefficients de pondération, respectivement `pa` (pondération en `a`), `pb` et `pc`.

Proposer un jeu de tests comprenant *au moins* trois tests.

Exercice 1.2 : Calcul d'un prix TTC

Le but de cet exercice est d'effectuer des conversions simples entre des *prix hors-taxes* (HT) et des *prix toutes taxes comprises* (TTC). Ceci permet de s'intéresser au passage d'un problème de la vie courante à une solution informatique.

Question 1

Écrire une définition de la fonction `prix_ttc` qui calcule le prix toutes taxes comprises (TTC) à partir d'un prix hors taxe (HT) et d'un taux de TVA exprimé en pourcentage, par exemple `20.0` pour une TVA de 20%.

Remarque : La signature de la division `/` est: `float * float -> float`.

Par exemple :

```
>>> prix_ttc(100.0, 20.0)
120.0

>>> prix_ttc(100, 0.0) # remarque : conversion implicite de l'entier
                       #             100 en flottant équivalent 100.0
100.0

>>> prix_ttc(100, 100.0)
200.0

>>> prix_ttc(0, 20)
0.0

>>> prix_ttc(200, 5.5)
211.0
```

Question 2

Donner une définition de la fonction `prix_ht` qui calcule le prix hors taxe à partir du prix toutes taxes comprises et du taux de TVA.

Remarque : votre jeu de tests doit correspondre à celui proposé pour la fonction `prix_ttc`.