Alexandre Casamayou-Boucau
Pascal Chauvin
Guillaume Connan

Programmation en Python pour les mathématiques

Cours et exercices

2e édition

DUNOD

5 rue Laromiguière, 75005 Paris
www.dunod.com

ISBN 978-2-10-074476-3

Préface à la première édition

Dans cet ouvrage consacré à l'étude du langage Python, Alexandre Casamayou-Boucau, Guillaume Connan et Pascal Chauvin ont réussi une difficile synthèse : celle de présenter une introduction claire au langage proprement dit, tout en explicitant ses liens profonds avec l'algorithmique mathématique.

Il est certain que le choix de Python rendait *a priori* cette synthèse possible : grâce à sa syntaxe de programmation multi-paradigme, l'écriture de programmes en Python est à la fois puissante et abordable ; elle est susceptible de couvrir la plupart des domaines de la programmation, et de façon éminente ceux pouvant intéresser l'algorithmique.

De ce point de vue, l'apprentissage d'un langage « fondamental » comme Python nous semble être une alternative didactique bien supérieure à celle qui consisterait à apprendre seulement l'usage de logiciels de calcul formel : pour donner une analogie, disons que c'est un peu la différence qu'il y a entre maîtriser en profondeur le fonctionnement et la mécanique d'une automobile, et le fait de simplement savoir la conduire ! Le fait que Python est entièrement en source libre et très utilisé dans l'industrie garantit en outre son accessibilité, sa pérennité et son évolutivité dans le temps ; ce n'est pas nécessairement le cas des logiciels propriétaires qui, pour cette raison, ne sauraient être recommandés au même titre pour des programmes d'enseignement.

L'ouvrage devrait conduire le lecteur à approfondir beaucoup sa connaissance des principes de la programmation et sa compréhension des algorithmes mathématiques fondamentaux. Il est construit à partir d'exemples nombreux et très riches qui en rendent la lecture attrayante. Ceux-ci devraient aussi grandement faciliter la tâche des enseignants qui l'utiliseront, depuis le lycée jusqu'aux classes préparatoires et à l'université.

Saint-Martin d'Hères, le 5 octobre 2011,

Jean-Pierre Demailly
Professeur à l'Université de Grenoble I
Membre de l'Académie des Sciences

Table des matières

Avant-propos

La réalisation d'un programme informatique de façon traditionnelle passe nécessairement par l'écriture de son code source. C'est cet aspect-là de la programmation qui nous intéresse tout particulièrement. Véritable activité de rédaction en soi, si fréquente dans les apprentissages, c'est par cette phase justement — qui peut être laborieuse mais aussi tellement gratifiante —, que l'on obtient le résultat désiré. Il nous semble important d'en convaincre les élèves.

Tout le travail consiste à analyser un problème et à décrire un moyen d'obtenir une solution. Dépourvu de toute capacité de déduction ou d'anticipation, le « robot », lui — interpréteur ou compilateur —, se contente de n'exécuter strictement que ce que l'auteur du programme aura explicité. La plus grande rigueur s'impose donc. Cette exigence requise dans la programmation, dont la pratique est encore toute nouvelle pour les collégiens et les lycéens, ne peut que leur être à terme bénéfique dans les autres apprentissages.

Certes, les environnements modernes de développement logiciel (le « *Rapid Application Development* » dans la terminologie anglo-saxonne) foisonnent de dispositifs d'assistance au programmeur, dans les outils employés. Mais il y a surtout, depuis quelques années, la programmation modulaire et la conception objet, qui permettent de segmenter de gigantesques projets pour une réalisation commune, partagée par des centaines d'individus. Il n'en demeure pas moins que les phases d'écriture perdurent, qui plus est avec des langages de programmation qui sont en ce début de troisième millénaire plus de deux mille, alors que cette activité a réellement pris son essor un peu avant la Seconde Guerre mondiale.

Si la programmation se sert de l'algorithmique pour être efficace, elle doit aussi être en mesure de fournir en plus des programmes à la fois lisibles, facilement utilisables et modifiables par d'autres utilisateurs. Depuis FORTRAN (1 956), langage qui laissait l'utilisateur très proche de la machine, les langages ne cessent d'évoluer vers un plus « haut niveau », à savoir, deviennent toujours plus accessibles.

Par exemple, voici trois routines (la première fonction est exprimée en langage C, la seconde en langage CaML, la troisième en langage **Python**) calculant la factorielle d'un entier naturel n :

factorielle.c

```
long factorielle(long n) {
    long resultat = 1;
    unsigned long i;
    if (n < 0)
        return -1;
    for (i = 1; i < n+1; ++i)
        resultat *= i;
    return resultat;
}
```

factorielle.ml

```
let rec factorielle = function
   | 0 -> 1
   | n -> n*factorielle(n-1);;
```

factorielle.py

```
def factorielle(n):
   if n > 0: return n*factorielle(n-1)
   else: return 1
```

Hormis le fait que la version C soit écrite dans un style itératif là où le langage CaML est redoutablement efficace en ce qui concerne la récursivité, quel est le langage de plus haut niveau ?...

En mathématiques, les langages de haut niveau basés sur un formalisme logique nous intéressent au plus haut point car ils ont cette rigueur qui sied à notre discipline et sont en même temps moins parasités par les détails technologiques des langages trop près de la machine ou trop lâches logiquement parlant.

Cependant, les langages comme le C sont largement employés (car ils sont liés aux systèmes UNIX), tout comme la programmation objet, si prisée pour la réalisation des interfaces homme/machine des programmes industriels.

C'est pourquoi nous avons choisi le langage **Python**. Il a fait ses preuves en tant que langage orienté objet, tout en permettant également une programmation impérative et récursive. **Python** présente une syntaxe claire et reste particulièrement abordable pour les débutants, tout en offrant des constructions de haut niveau.

Les langages cités ici sont, parmi d'autres non moins importants, largement répandus et développés par des équipes à la pointe de la recherche informatique. Ils ont été pensés, améliorés depuis des années et évitent les écueils de certaines interfaces prétendument simples à manipuler mais qui peuvent cacher de nombreux vices et ne déboucheront sur aucune utilisation en dehors du lycée.

Comme on continue à faire des mathématiques et à calculer avec un papier et un crayon, de même on préférera un contexte de programmation le plus sobre possible, dépouillé de tout ce qui est inutile et qui nuit à la réflexion. Un éditeur de texte simple (et non un logiciel de traitement de texte) fait amplement l'affaire, accompagné d'une console (`*N*X` de préférence) pour l'interprétation ou la compilation, puis l'exécution d'un programme. Il s'agit vraiment de se concentrer sur l'essentiel, et nous voyons les élèves accepter ce travail, dans la mesure où la récompense est immédiate : soit le programme fonctionne, soit il ne fonctionne pas, ou bien il effectue autre chose que ce que son auteur prévoyait. Aucune autre récompense n'est visée que la satisfaction intellectuelle propre à l'apprentissage. Or c'est là une priorité de l'école.

Comment utiliser ce livre ?

Le présent ouvrage vise deux objectifs : introduire la programmation en **Python** et appliquer les notions ainsi acquises aux mathématiques.

Après un premier chapitre présentant les fondamentaux du langage **Python**, un deuxième chapitre aborde la notion de module. D'une part, il est expliqué comment programmer un module personnel ; d'autre part, on présente quelques-uns des très nombreux modules fournis par défaut avec **Python**[1]. En ce qui concerne les modules de tierces parties les plus connus des scientifiques (`NumPy` et `SciPy` pour le calcul numérique, `Matplotlib` pour les graphiques, `SymPy` pour le calcul formel), nous ne les mentionnerons que sommairement ici ou

1. Nous n'aborderons pas les modules permettant de manipuler des bases de données SQL avec **Python**. Le lecteur intéressé par ce sujet pourra, par exemple, consulter le chapitre 16 de l'ouvrage [Swi10] dont une version électronique est librement téléchargeable sur le site de son auteur.

là : en effet, leur présentation détaillée dépasse largement le cadre de cet ouvrage. En outre, pour le tracé des fonctions, nous avons fait le choix de développer, à titre pédagogique, un petit module permettant de générer un graphique au format PostScript à partir d'une liste de points à tracer.

Le chapitre suivant se propose de présenter une liste variée d'algorithmes mathématiques programmés le plus simplement possible en **Python** : sont illustrés plusieurs algorithmes d'arithmétique (algorithme d'EUCLIDE, tests de primalité, etc.), de cryptographie (chiffrement de HILL, de VIGENÈRE, système RSA, etc.), les problématiques d'approximation décimale (calcul des premières décimales de π par les méthodes de Nicolas DE CUES, de John MACHIN, de BRENT et SALAMIN, etc.), des problèmes de probabilités et de théorie des graphes.

Ensuite, sont passées en revue les méthodes classiques d'analyse numérique : résolution d'équations diverses, d'équations différentielles ordinaires, de dérivation et d'intégration numériques.

Les deux derniers chapitres sont davantage tournés vers l'algorithmique. Après avoir traité de la notion de récursivité, le dernier chapitre prend comme fil conducteur plusieurs implémentations d'un petit calculateur formel : sont abordées quelques structures de données classiques (arbres, graphes, piles, queues, etc.) et la conception orientée objet, dont un objectif est d'obtenir, au moyen de l'abstraction, une bonne modularité, si utile pour les grands projets.

Certains programmes un peu trop longs pour figurer sur la version papier sont disponibles dans une archive, qui contient en outre l'ensemble de ceux du livre, et que l'on peut télécharger sur la page du site `www.dunod.com` consacrée à cet ouvrage.

Les notions exposées dans cet ouvrage sont illustrées par des exercices répartis au long de chaque chapitre. De plus, en fin de chapitre sont regroupés des exercices d'entraînement dont les corrigés sont disponibles sur le site dunod.com à partir de la page d'accueil de l'ouvrage.

L'ouvrage s'adresse notamment aux professeurs de mathématiques au lycée. Ils pourront vérifier que l'utilisation du langage **Python** est un très bon choix pour l'enseignement de l'algorithmique inscrit dans les nouveaux programmes de mathématiques. Les élèves du lycée et les étudiants de licence et des classes préparatoires pourront également trouver ici de quoi stimuler leur apprentissage de la programmation.

Aucune notion en informatique n'est requise pour entreprendre la lecture de cet ouvrage.

Conventions pour la présentation du code

Pour l'écriture du code, vous rencontrerez plusieurs présentations :

– les instructions précédées de chevrons dans une boîte grisée sont à saisir dans une session interactive ;

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

– les instructions sans chevrons dans une boîte grisée sont des bouts de code à écrire dans un fichier ;

```
print('Bonjour !')
```

– les instructions dans une boîte grisée avec un filet sombre à gauche et un nom de fichier en italique au-dessus de la boîte sont des extraits d'un fichier se trouvant dans l'archive téléchargeable sur le site de l'éditeur ; dans ce cas, si le résultat de l'exécution du script est présentée, elle apparaît immédiatement après le script sur fond blanc ;

fichier.py

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

print('Voici le résultat.')
```

```
Voici le résultat.
```

– deux traits horizontaux délimitent une session dans un terminal Unix :

Terminal

```
$ python3 fichier.py
Voici le résultat.
$ python3
Python 3.1.3 (r313:86834, Nov 28 2010, 11:28:10)
[GCC 4.4.5] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 1 + 1
2
```

Remerciements

Nous tenons à remercier vivement tous ceux qui ont relu le manuscrit de cet ouvrage : Alain BUSSER, Stéphane GROGNET, Hubert H. HUPKES, Gérard KUNTZ, François PANTIGNY et Aymar DE SAINT-SEINE.

Nous remercions également nos étudiants et élèves pour avoir servi de « cobayes » et proposé des améliorations de nos programmes.

Enfin, nous remercions vivement Jean-Pierre DEMAILLY qui a accepté de préfacer cet ouvrage.

1

Introduction au langage Python

Sommaire

1 Pourquoi Python ?

Le langage de programmation **Python** est un très bon choix aussi bien pour l'initiation à la programmation que pour la programmation elle-même. C'est un langage de très haut niveau dont la syntaxe encourage à écrire du code clair et de qualité. Dans le domaine de la gestion de la mémoire, nombre de détails de bas niveau propres aux langages comme le C disparaissent. De plus l'apprentissage de **Python** est facilité par l'existence d'une interface interactive. Cela dit, son intérêt ne se réduit pas à l'apprentissage de la programmation ou de l'algorithmique ;

en témoigne sa popularité croissante. Il a été choisi par des acteurs majeurs : Google, YouTube, la NASA, etc.
Techniquement parlant, **Python** est un langage où l'on peut choisir plusieurs styles de programmation. Il favorise la programmation impérative structurée et la programmation orientée objet ; dans une moindre mesure, il permet de programmer dans un style fonctionnel [1]. Il est doté d'un typage dynamique fort, d'une gestion automatique de la mémoire par ramasse-miettes et d'un système de gestion d'exceptions. C'est un langage multi-plateforme, polyvalent (jusque dans les domaines comme le web, les graphiques, le réseau), « open source », et gratuit.
Enfin, l'utilisation de **Python** pourra être couplée à celle du logiciel libre de calcul formel `Sagemath` [2] puisque ce dernier est écrit en **Python**.
Si ce bref plaidoyer ne vous a pas convaincu(e), essayez **Python**, vous l'adopterez certainement.

2 Avant de commencer...

Pour installer **Python**, il suffit de télécharger la version 3 qui correspond à votre système d'exploitation (Windows ou Mac) à l'adresse : `http://www.Python.org/`
Pour ce qui est des systèmes Linux, **Python** est généralement déjà installé par défaut, et `Idle` se trouve dans les dépôts de la plupart des distributions. Pour les systèmes BSD, si **Python** n'est pas déjà installé comme dépendance, utiliser les paquetages ou les ports.
En complément de ce chapitre de présentation des bases du langage **Python**, le lecteur pourra également consulter avec profit les cinq premières sections du tutoriel officiel de **Python** :

`http://docs.python.org/py3k/tutorial/`

3 Utiliser Python comme une calculette

Si vous n'avez jamais programmé, le plus simple pour exécuter des instructions **Python** est d'utiliser l'environnement spécialisé `Idle`. Cet environnement se compose d'une fenêtre appelée indifféremment *console*, *shell* ou *terminal* **Python**.
L'invite de commande se compose de trois chevrons ; il suffit de saisir à la suite une instruction puis d'appuyer sur la touche « Entrée » de votre clavier.
Nous ne dérogerons pas à la tradition informatique qui consiste à commencer l'apprentissage d'un langage par l'affichage [3] d'une salutation :

```
>>> print("Bonjour !")
Bonjour !
```

La console **Python** fonctionne comme une simple calculatrice : vous pouvez y saisir une expression dont la valeur est renvoyée dès que vous pressez la touche « Entrée ».

1. Cet aspect ne sera pas abordé dans cet ouvrage ; le lecteur intéressé pourra se reporter à la page « Functional Programming HOWTO » de la documentation : `http://docs.python.org/py3k/howto/functional.html`.
2. cf. `http://www.sagemath.org/` et `http://www.sagemath.org/fr/`.
Pour une introduction à `Sagemath`, on pourra consulter le livre électronique *Calcul mathématique avec Sage* librement téléchargeable à l'adresse `http://sagebook.gforge.inria.fr/`.
3. Les utilisateurs de **Python** 2 remarqueront qu'à partir de la version 3, l'*instruction* `print` est remplacée par la *fonction* `print()`. Pour plus de détails, cf. `http://docs.python.org/3.1/whatsnew/3.0.html`.

```
>>> 2 * 5 + 6 - (100 + 3)
-87
>>> 7 / 2; 7 / 3
3.5
2.3333333333333335
>>> 34 // 5; 34 % 5 # quotient et reste de la division euclidienne de 34 par 5
6
4
>>> 2 ** 7  # pour l'exponentiation (et non pas 2^7 !)
128
```

Au passage, nous avons utilisé le symbole « dièse » # pour placer des commentaires dans les lignes de commande ; tout ce qui se situe à droite d'un symbole # (jusqu'au changement de ligne) est purement et simplement ignoré par l'interpréteur.
Pour naviguer dans l'historique des instructions saisies dans la console **Python**, on peut utiliser les raccourcis `Alt+p` (`p` comme *previous*) et `Alt+n` (`n` comme *next*).

4 Variables et affectations

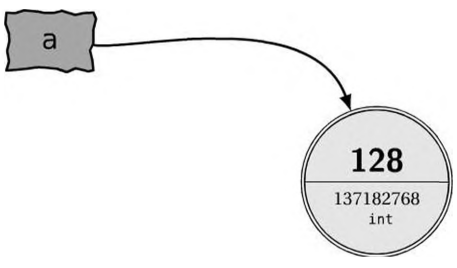
Que se passe-t-il au juste lorsqu'on saisit un nombre (par exemple 128) dans la console **Python** ? Eh bien, disons, en première approximation, que l'interpréteur crée un nouvel « objet » (sans préciser pour l'instant le sens de ce mot) et le garde en mémoire.

```
>>> 128, id(128), type(128)
(1, 137182768, <class 'int'>)
```

Cet objet possède une *valeur* (ici 128), un *identifiant*, c'est-à-dire une « carte d'identité » permettant de savoir où il est gardé en mémoire (ici 137182768), et enfin un *type*[4] (ici le type entier dénommé `int`).

Le moins que l'on puisse dire, c'est que l'identifiant ne nous parle guère... D'où l'intérêt des *affectations*. Au lieu de désigner 128 par son identifiant, on va lui donner un nom commode à manipuler. Au lieu d'appeler l'objet « Monsieur 137180768 », on l'appellera « Monsieur A. », après avoir indiqué à l'interpréteur une fois pour toute l'identification opérée par un message du type : « Monsieur A. *alias* Monsieur 137180768 ».
En pratique, une affectation s'effectue à l'aide du symbole « = », comme ceci :

```
>>> a = 128
>>> a
128
>>> a, id(a), type(a)
(128, 137180768, <class 'int'>)
>>> 2 * a
256
```



4. La notion de type sera détaillée à la section 11 page 31.

Ainsi chaque fois que nous appellerons le nombre 128, il nous suffira d'invoquer la variable a. Notez que l'objet désigné par a est toujours l'objet 128, rangé encore à la même adresse.
Il faut bien prendre garde au fait que l'instruction d'affectation « = » n'a pas la même signification que le symbole d'égalité « = » en mathématiques[5]. Par exemple, le premier n'est pas symétrique, alors que le second l'est : vouloir échanger l'ordre des éléments dans une instruction d'affectation produira immanquablement une erreur dans l'interpréteur :

```
>>> 128 = a
  File "<stdin>", line 1
SyntaxError: can't assign to literal
```

Effectuons à présent la suite d'affectations suivantes :

```
>>> b = a * 2
>>> b                    # rép.: 256
>>> b, id(b), type(b)    # rép.: (256, 137184816, <class 'int'>)
>>> a = 0
>>> a, id(a), type(a)    # rép.: (0, 137180720, <class 'int'>)
>>> b                    # rép.: 256
```

Ici se situe une petite difficulté pour ceux qui débutent la programmation. Contrairement à ce que nos habitudes de calcul algébrique pourraient nous laisser penser, l'instruction `b = a*2` n'affecte pas à b le double de la valeur a quelle que soit la valeur de a au long de la session **Python**. Au contraire, l'instruction `b = a*2` procède en deux temps :

- l'expression située à droite du signe « = » est évaluée, c'est-à-dire calculée en fonction de l'état de la mémoire *à cet instant* : ici l'interpréteur évalue le double de la valeur de a ; le résultat est un objet de type entier, de valeur 256, et placé en mémoire avec l'identifiant `137180720`.
- ensuite, et seulement ensuite, l'interpréteur affecte au nom situé à gauche de l'instruction d'affectation (à savoir b) l'objet obtenu après évaluation de l'expression de droite.

On remarque que l'identifiant de l'objet auquel renvoie la variable b n'a plus rien à voir avec a. Autrement dit, l'objet nommé b n'a plus aucune relation avec l'objet nommé a. Ainsi, une réaffectation ultérieure de la variable a n'entraînera aucun changement pour la variable b.
Avez-vous bien compris ? Alors exercez-vous en lisant les suites d'instructions suivantes et en notant sur un papier le contenu de chacune des variables à chaque étape ; puis exécutez chacune de ces suites d'instructions dans la console et vérifiez que ce que vous avez noté concorde avec ce qu'affiche l'interpréteur.

```
>>> a = 100
>>> b = 17
>>> c = a - b
>>> a = 2
>>> c = b + a
>>> a, b, c
```

```
>>> a = 3
>>> b = 4
>>> c = a
>>> a = b
>>> b = c
>>> a, b, c
```

5. Ceci explique que dans les livres d'algorithme, l'affection de `expr` à `x` se note souvent `x ← expr`.

Allons un peu plus loin ; il est fréquent qu'en programmation, on se serve d'une variable comme d'un compteur et que l'on ait donc besoin d'incrémenter (c'est-à-dire augmenter) ou de décrémenter (c'est-à-dire diminuer) la valeur de la variable d'une certaine quantité. On procède alors de la manière suivante.

```
>>> x = 0
>>> x = x + 1
>>> x, id(x), type(x)        # rép.: (1, 137180736, <class 'int'>)
>>> x = x + 1
>>> x, id(x), type(x)        # rép.: (2, 137180752, <class 'int'>)
>>> x = x + 1
>>> x, id(x), type(x)        # rép.: (3, 137180768, <class 'int'>)
```

Encore une fois, notez bien la différence avec le calcul algébrique : alors que l'équation $x = x + 1$ n'a pas de solution, l'instruction `x = x + 1` est parfaitement licite, et même utilisée couramment en programmation.
Détaillons la première instruction `x = x + 1` ci-dessus ; cette instruction procède en deux temps :

- l'interpréteur évalue la valeur de `x + 1` à l'instant donné ; le résultat est un objet de type entier, de valeur 1, et placé en mémoire avec l'identifiant `137180736`.
- ensuite, et seulement ensuite, l'interpréteur affecte au nom qui se trouve à gauche de l'instruction d'affectation (à savoir `x`) l'objet obtenu après évaluation de l'expression de droite.

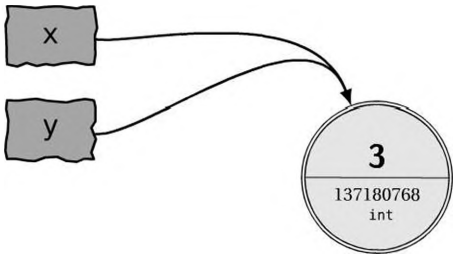
Signalons un raccourci propre à **Python** très utile en pratique et qui a l'avantage d'éviter la confusion avec la manipulation d'équations en algèbre.

```
>>> x += 1  # remplace x par x+1
>>> x -= 3  # remplace x par x-3
```

```
>>> x *= 3  # remplace x par x*3
>>> x /= 2  # remplace x par x/2
```

Autre raccourci intéressant, on peut assigner un même objet à plusieurs variables simultanément. Ces deux variables renvoient alors au même objet (on dit parfois que ce sont deux *alias* du même objet).

```
>>> x = y = 3
>>> x, y
(3, 3)
>>> id(x), id(y)
(137180768, 137180768)
```



On peut aussi effectuer des *affectations parallèles* à l'aide d'un seul opérateur « = ». Toutes les expressions sont alors évaluées *avant* la première affectation.

```
>>> x, y = 128, 256
```

Encore une fois, il faut bien comprendre qu'une affectation se déroule en deux temps : évaluation de l'expression de droite puis création de l'alias avec le nom du terme de gauche.
Pourriez-vous prévoir le résultat des deux suites d'instructions suivantes ?

```
>>> x = 19
>>> x = x + 2 ; y = x * 2
>>> x, y
```

```
>>> x = 19
>>> x, y = x + 2, x * 2
>>> x, y
```

Voici à présent un exercice qu'il est nécessaire de comprendre parfaitement avant de continuer plus avant.

Un exercice fondamental : l'échange des contenus de deux variables

On suppose que les variables x et y ont pour valeurs respectives des entiers α et β. On souhaite échanger le contenu de ces deux variables.

a) *Première méthode :* Proposer une méthode qui utilise une variable auxiliaire `tmp`.

b) *Deuxième méthode :* On exécute la séquence d'instructions suivante :

```
>>> x = x + y; y = x - y; x = x - y
```

Quel sont les contenus des variables x et y en fin de séquence ?

c) *Troisième méthode (la plus « pythonique ») :* Utiliser une affectation parallèle.

Solution.

a)
```
>>> tmp = x
>>> x = y
>>> y = tmp
```

b) On a échangé les valeurs de x et y.

c)
```
>>> x, y = y, x
```

Signalons rapidement que pour supprimer une variable, on dispose de la fonction `del`.
Avant de clore cette section, précisons que les noms de variables peuvent être non seulement des lettres, mais aussi des mots ; ils peuvent contenir des chiffres (à condition toutefois de ne pas commencer par un chiffre), ainsi que certains caractères spéciaux comme le tiret bas « _ » (appelé *underscore* en anglais). Le bon programmeur s'efforce bien entendu de choisir les noms de variables les plus pertinents possible.

5 Fonctions

Supposons que nous cherchions à calculer les images de certains nombres par une fonction polynomiale donnée. Si la fonction en question est un peu longue à saisir, par exemple, $f : x \mapsto x^7 - 6x^6 + 15x^4 + 23x^3 + x - 9$, il est rapidement fastidieux de la saisir à chaque fois que l'on souhaite calculer l'image d'un nombre par cette fonction.
Une première idée est d'utiliser l'historique de la console pour éviter de saisir à chaque fois la fonction :

```
>>> x = 2
>>> x**7 - 6*x**6 + 15*x**4 + 23*x**3 + x - 9
161
>>> x = 3
>>> x**7 - 6*x**6 + 15*x**4 + 23*x**3 + x - 9
```

```
-357
>>> x = 4
>>> x**7 - 6*x**6 + 15*x**4 + 23*x**3 + x - 9
-2885
```

Néanmoins, on se doute bien qu'il y a un moyen de s'économiser ce bricolage...
Il est tout à fait possible de définir une fonction (au sens du langage **Python**) qui ressemble à une fonction mathématique. La syntaxe est alors la suivante :

```
>>> def f(x):
...     return x**7 - 6*x**6 + 15*x**4 + 23*x**3 + x - 9
...
>>> f(2), f(3), f(4)     # rép.: (161, -357, -2885)
```

Observons de près cette définition. Tout d'abord, la déclaration d'une nouvelle fonction commence par le mot-clé `def`. Ensuite, toujours sur la même ligne, vient le nom de la fonction (ici `f`) suivi du *paramètre formel*[6] de la fonction, placé entre parenthèses (le paramètre formel `x` de la fonction **Python** correspond ici à la variable muette de la fonction mathématique), le tout ponctué de deux-points.
Une fois la première ligne saisie, on appuie sur la touche « Entrée » et on constate que les chevrons de l'invite de commande ont été remplacés par des points. Cela signifie que l'interpréteur attend obligatoirement la suite des instructions. Il faut alors saisir quatre espaces pour les « indenter ». Enfin, une ligne vide signifiera à l'interpréteur que la définition de notre fonction est terminée, et qu'il peut désormais lire la totalité du bloc d'instructions.
Pour mieux comprendre les règles de définition de fonctions, donnons brièvement en guise de bêtisier trois erreurs à ne pas reproduire :

```
>>> # ---> Erreur 1 : l'oubli des deux points en fin de ligne
>>> def f(x)
  File "<stdin>", line 1
    def f(x)
           ^
SyntaxError: invalid syntax

>>> # ---> Erreur 2 : le non-respect de l'indentation
>>> def f(x):
... return x**7 - 6*x**6 + 15*x**4 + 23*x**3 + x - 9
  File "<stdin>", line 2
    return x**7 - 6*x**6 + 15*x**4 + 23*x**3 + x - 9
         ^
IndentationError: expected an indented block

>>> # ---> Erreur 3 : l'oubli du mot return
>>> def f(x):
...     x**7 - 6*x**6 + 15*x**4 + 23*x**3 + x - 9
...
>>> f(2), f(3), f(4)
(None, None, None)
```

6. Les paramètres figurant entre parenthèses dans l'en-tête d'une fonction se nomment *paramètres formels*, par opposition aux paramètres fournis lors de l'appel de la fonction appelés *paramètres effectifs*.

Que s'est-il passé dans la dernière tentative de définition de la fonction `f` ? Tout simplement, au cours de l'exécution de la fonction `f`, l'expression `x**7-6*x**6+15*x**4+23*x**3+x-9` est calculée, mais l'interpréteur, n'ayant pas reçu l'instruction de renvoyer le résultat, la garde pour lui : il reste muet et se contente de renvoyer comme valeur l'objet `None`.
D'aucuns pourraient être tentés de remplacer l'instruction `return` par la fonction `print` que nous avons entrevue au tout début de ce chapitre.

```
>>> def f(x):
...     print(x**7 - 6*x**6 + 15*x**4 + 23*x**3 + x - 9)
...
>>> f(2), f(3), f(4)
19
113
55
(None, None, None)
```

Voici un exemple de calcul qui nous permettra de mieux comprendre la différence fondamentale entre l'instruction `return` et la fonction `print`. Supposons que nous cherchions à calculer la somme des images de notre fonction `f` évaluée en certaines valeurs :

```
>>> def f(x):
...     return x**7 - 6*x**6 + 15*x**4 + 23*x**3 + x - 9
...
>>> f(2) + f(3) + f(4)
187
>>> def f(x):
...     print(x**7 - 6*x**6 + 15*x**4 + 23*x**3 + x - 9)
...
>>> f(2) + f(3) + f(4)
19
113
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'
```

Que se passe-t-il dans le deuxième cas ? L'interpréteur affiche à l'écran la valeur de l'expression `x**7 - 6*x**6 + 15*x**4 + 23*x**3 + x - 9` lorsqu'il rencontre la fonction `print`, mais ensuite, il ne renvoie aucun objet réutilisable ultérieurement, sinon toujours cet objet `None` qui résiste à toute tentative d'addition. En fait, la fonction `print` est utilisée pour son *effet de bord*.
La morale de cette histoire, c'est qu'il vaut mieux éviter de prendre la mauvaise habitude d'utiliser la fonction `print` à l'intérieur des fonctions, sauf si c'est explicitement demandé. La plupart des fonctions que nous utiliserons dans cet ouvrage seront des fonctions contenant dans leur définition l'instruction `return`. Bien sûr, la fonction `print` est utile et sera utilisée, mais elle le sera surtout en dehors des fonctions, dans des suites d'instructions regroupées dans des fichiers appelés *scripts*.
Une propriété remarquable de l'instruction `return` est qu'elle interrompt systématiquement l'exécution de la fonction. Dès que l'interpréteur atteint l'instruction `return` *qqchose*, il renvoie l'objet *qqchose* et abandonne aussitôt après l'exécution de la fonction. Inutile donc de

placer des instructions après un `return` : ces instructions ne seront jamais lues par l'interpréteur. On parle parfois de *code mort* pour désigner les lignes qui suivent (à l'intérieur de la définition d'une fonction) l'instruction `return`.

```
>>> def mult_7(x):
...     return 7 * x
...     print("Ceci ne s'affichera jamais") # c'est du code mort !
...     return 0                  #  c'est encore du code mort !!!
...
```

Autre fait notable lorsqu'on définit une fonction : les variables définies à l'intérieur d'une fonction ne sont pas « visibles » depuis l'extérieur de la fonction ; elles correspondent aux variables muettes en mathématiques. Aussitôt l'exécution de la fonction terminée, l'interpréteur efface toute trace des variables internes à la fonction. On exprime cela en disant qu'une telle variable est *locale* à la fonction.
Dans l'exemple suivant, la variable `x` est une variable locale à la fonction `f` : créée au cours de l'exécution de la fonction `f`, elle est supprimée une fois l'exécution terminée.

```
>>> def f(y):
...     x = 1
...     return y
...
>>> f(2)
2
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Si une variable `x` existait déjà avant l'exécution de la fonction, tout se passe comme si, durant l'exécution de `f`, cette variable était masquée momentanément, puis restituée à la fin de l'exécution de la fonction.

```
>>> x = 0
>>> def f(y):
...     x = 1
...     return y
...
>>> f(2)
2
>>> x
0
```

Dernier point avant de clore cette section : signalons qu'une fonction peut comporter autant de paramètres formels qu'on le souhaite (et éventuellement aucun) :

```
>>> def ma_fonction(x, y):
...     return x * 2**y
...
```

6 Instructions d'écriture et de lecture

Au fur et à mesure de l'apprentissage du langage, on enchaîne de plus en plus d'instructions. Si l'on veut corriger une instruction dans une telle succession, on est obligé de saisir à nouveau les instructions suivantes. De plus, on souhaite souvent enregistrer une telle suite particulière d'instructions pour pouvoir s'en resservir ultérieurement ou la modifier. D'où la nécessité de travailler avec des fichiers dans lesquels on place des suites d'instructions.
Sous l'environnement `Idle`, il suffit d'ouvrir un fichier texte en choisissant l'entrée `NewWindow` dans le menu `File` ou en utilisant le raccourci `Ctrl-N` : on écrit alors directement dans le fichier des instructions ; puis on sauvegarde son fichier sous un nom de la forme `toto.py` (le choix de l'extension `py` permettant de bénéficier de la coloration syntaxique) ; enfin la commande `RunModule` du menu `Run` vous permettra de lancer l'exécution du contenu de votre fichier dans la console **Python**.
Ceux qui ont déjà appris à se servir d'un éditeur de texte (comme `Vim` ou `Emacs`) préfèreront écrire leurs scripts à l'aide d'un tel éditeur et exécuter leur script soit à l'aide d'un raccourci qu'ils auront défini, soit directement dans une console.
Commençons par un exemple. Après écriture du fichier suivant, exécutez-le :

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

x = 2 ** 8
x
```

```
==== No Subprocess ====
>>>
>>>
```

Déception... rien ne s'affiche. En fait, le comportement de l'interpréteur diffère légèrement suivant que l'on travaille dans une console **Python** ou dans un fichier texte.
En effet, lorsque l'interpréteur lit un fichier texte, il effectue les instructions les unes à la suite des autres, mais il n'affiche rien tant que cela ne lui est pas demandé explicitement.
D'où, dans le fichier précédent, la nécessité de rajouter la fonction `print` si l'on souhaite voir s'afficher la valeur de la variable `x`. Du reste, c'est à partir du moment où l'on écrit des scripts dans des fichiers que l'utilisation de la fonction `print` prend tout son intérêt.

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

x = 2 ** 8
print(x)
```

```
==== No Subprocess ====
>>>
>>>
256
>>>
```

Dans le fichier précédent, nous avons placé deux lignes qui, pour être facultatives, n'en restent pas moins énigmatiques pour un débutant : la première, dite ligne de « shebang », précise où est l'interpréteur[7] à utiliser pour exécuter le fichier. Les utilisateurs de systèmes d'exploitation Linux ou MacOS pourront alors exécuter ce script dans un Terminal en saisissant : `./toto.py` aussi bien que `python3 toto.py`

7. Dans un terminal, taper `which python3` pour connaître l'adresse absolue de l'exécutable.

Terminal

```
$ python3 toto.py
256
$ chmod u+x toto.py # nécessaire pour accorder le droit d'exécution
$ ./toto.py
256
```

Parfois, il arrive que l'on souhaite exécuter un script, puis passer dans la foulée en mode interactif : il suffit d'ajouter l'option `-i` :

Terminal

```
$ python3 -i monfichier.py
```

La deuxième ligne du fichier source précise l'*encodage*, c'est-à-dire le format de codage des caractères utilisé dans votre fichier texte. Là encore cette ligne est facultative, mais elle augmente la portabilité de vos programmes d'un système d'exploitation à un autre. Suivant que votre système d'exploitation utilise par défaut l'encodage Latin-1 ou l'encodage Utf-8 (cette dernière norme étant fortement recommandée), utilisez

```
# -*- coding : Latin-1 -*-    ou    # -*- coding : Utf-8 -*-
```

Dorénavant, par souci de gain de place, nous ne mentionnerons plus ces deux lignes.
Maintenant que nous savons exécuter une suite d'instructions dans un fichier, revenons un instant sur l'indentation dans la définition d'une fonction. Écrivez dans des fichiers les deux scripts suivants, et prévoyez le déroulement de leur exécution, puis vérifiez en les exécutant :

```
def f():
    print('Hello')
print('Bonjour')
f()
```

```
def f():
    print('Hello')
    print('Bonjour')
f()
```

Arrêtons-nous quelques instants sur la fonction d'écriture `print`. Cette fonction comporte de nombreuses options qui permettent de personnaliser la présentation des données :

```
x, y = 3, 1000000000
z = 3.1416
print(x, y, z, sep='')
print(x, y, z, sep='; ')
print(x, y, z, sep='\n')
print('x=', x, sep='', end='; ')
print('y=', y, sep='', end='; ')
print('z=', z, sep='')
```

```
==== No Subprocess ====
>>>
310000000003.1416
3; 1000000000; 3.1416
3
1000000000
3.1416
x=3; y=1000000000; z=3.1416
```

Bien que n'ayant pas encore parlé des chaînes de caractères, nous les avons déjà employées au détour de nos exemples, à commencer par le fameux `print("hello")`. Une *chaîne de caractères* (c'est-à-dire un objet de type `string`) est une succession de caractères typographiques

de longueur quelconque, délimitée par de simples « quotes » (apostrophes) ou des doubles « quotes » (guillemets). Les triples guillemets permettent d'inclure des retours à la ligne à l'intérieur de la chaîne.

```
print('-> Ici on peut employer des "guillemets" !')
print("-> Ici on peut employer des 'apostrophes' !")
print("""-> Voici un saut...
   de ligne.""")
print("-> On peut aussi\n   passer à la ligne ainsi.")
```

```
-> Ici on peut employer des "guillemets" !
-> Ici on peut employer des 'apostrophes' !
-> Voici un saut...
   de ligne.
-> On peut aussi
   passer à la ligne ainsi.
```

Pour disposer sur plusieurs lignes une chaîne de caractère un peu longue dans un fichier de script, il est déconseillé d'utiliser une contre-oblique ; la syntaxe suivante est préférable [8] :

```
print('-> Comment couper une ligne '
      'trop longue dans le fichier source ?')
```

```
-> Comment couper une ligne trop longue dans le fichier source ?
```

La méthode `format` de l'objet `string` est un outil très puissant permettant de créer des chaînes de caractères en remplaçant certains champs (entre accolades) par des valeurs (passées en argument de la fonction `format`) après conversion de celles-ci. On peut préciser à l'intérieur de chaque accolade un code de conversion, ainsi que le gabarit d'affichage. Donnons quelques exemples.

```
>>> x = 1037.123456789
>>> '{:g}'.format(x)   # choisit le format le plus approprié
'1.04e+03'
>>> '{:.3f}'.format(x) # fixe le nombre de décimales
'1037.123'
>>> '{:.3e}'.format(x) # notation scientifique
'1.037e+03'
>>> '{0:20.3f}'.format(x)  # précise la longueur de la chaîne
'            1037.123'
>>> '{0:>20.3f}'.format(x) # justifié à droite
'            1037.123'
>>> '{0:<20.3f}'.format(x) # justifié à gauche
'1037.123            '
>>> '{0:^20.3f}'.format(x) # centré
'      1037.123      '
>>> '{0:+.3f} ; {1:+.3f}'.format(x, -x) # affiche toujours le signe
'+1037.123 ; -1037.123'
>>> '{0: .3f} ; {1: .3f}'.format(x, -x) # affiche un espace si x>0
```

8. Pour plus de détails, cf. la dernière section de http://docs.python.org/release/3.1.3/howto/doanddont.html

```
' 1037.123 ; -1037.123'
>>> '{0:-.3f} ; {1:-.3f}'.format(x, -x) # équivaut à '{0:.3f}'
'1037.123 ; -1037.123'
```

Pour une description détaillée de la méthode `format`, on renvoie à la documentation de **Python** : http://docs.python.org/py3k/library/string.html
Après l'instruction d'écriture `print` qui permet à un programme en cours d'exécution d'afficher certains résultats, passons maintenant à l'instruction de lecture `input` qui permet, en sens inverse, de demander à l'utilisateur de fournir des données au programme en cours d'exécution.
Soit à écrire un petit script qui demande à l'utilisateur de fournir un nombre, qui l'affecte à la variable `x`, et qui renvoie le carré de ce nombre.

```
x = input('Entrez une valeur pour la variable x :')
print("{}^2 = {}".format(x, x**2))
```

```
Entrez une valeur pour la variable x :3
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    print("{}^2 = {}".format(x, x**2))
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

Déception, notre tentative échoue : on remarque que la valeur 3 a pourtant bel et bien été récupérée par le programme et que la variable `x` semble contenir l'entier 3. Or il n'en est rien. Ce que contient la variable `x`, c'est la chaîne de caractères `"3"`. Et une chaîne de caractère ne peut être élevée au carré.
D'où la nécessité de parler tôt ou tard du *type* d'un objet... Sans trop entrer dans les détails du fonctionnement d'un ordinateur, rappelons que toute information, et en particulier le contenu d'une variable, doivent être codés en binaire. Mais le « dictionnaire » pour coder ou décoder le contenu d'une variable ne sera pas le même suivant que l'on a affaire à un entier, un nombre en notation scientifique ou une chaîne de caractères. Dans ces conditions, la connaissance du contenu binaire d'une variable ne suffit pas pour déterminer l'information correspondante. Il est nécessaire de savoir, en outre, comment la valeur qui s'y trouve est codée. Cette distinction correspond à la notion de type.
Pour connaître le type d'un objet, nous avons déjà vu qu'on disposait de la fonction `type` :

```
>>> type(3), type("3"), type(x)
(<class 'int'>, <class 'str'>, <class 'str'>)
```

Ainsi 3 est du type entier (`int` étant l'abréviation du mot « integer »), tandis que `"3"` est du type chaîne de caractères (`str` étant l'abréviation du mot « string »).
Pour rendre notre script fonctionnel, il est alors nécessaire de faire appel à la fonction `eval` qui évalue l'expression représentée par une chaîne de caractères.

```
x = eval(input('Entrez une valeur pour la variable x :'))
print("{}^2 = {}".format(x, x**2))
```

```
Entrez une valeur pour la variable x :3
3^2 = 9
```

Un peu dans le même ordre d'idées, citons la fonction `exec` qui permet d'exécuter une instruction représentée par une chaîne de caractères.
Donnons un petit exemple sans le détailler (pour se mettre en appétit, il fait apparaître le module `math` qui permet d'*importer* les fonctions mathématiques usuelles, ainsi que certaines constantes usuelles comme le nombre π).

```
from math import *
x = eval(input('x=?'))
fonction = input('f=?')
code = ("def f(x):"
        "    return {}".format(fonction))
exec(code)
print('{:.6f}'.format(f(x)))
```

```
x=?pi/4
f=?sin(x)
0.707107
```

Pour terminer cette section, où nous avons appris à écrire de petits scripts dans des fichiers, signalons qu'il est bon de prendre tout de suite l'habitude de bien présenter son code en **Python** selon les conventions suivantes :

- taille des indentations : 4 espaces ;
- taille maximale d'une ligne : 79 caractères ;
- toujours placer un espace après une virgule, un point-virgule ou deux-points (sauf pour la syntaxe des tranches) ;
- ne jamais placer d'espace avant une virgule, un point-virgule ou deux-points ;
- toujours placer un espace de chaque côté d'un opérateur ;
- ne pas placer d'espace entre le nom d'une fonction et sa liste d'arguments.

```
# Déconseillé
def f(x)  :
        return 1

x=1 ; y=2;z = 3
x,y ,z
f  (z)
```

```
# Conseillé
def f(x):
    return 1

x = 1; y = 2; z = 3
x, y, z
f(z)
```

Pour plus de précisions, on se reportera au lien suivant : http://docs.python.org/py3k/tutorial/controlflow.html#intermezzo-coding-style

7 La structure conditionnelle

Supposons que nous souhaitions définir la fonction valeur absolue : $|x| = \begin{cases} x \text{ si } x \geq 0 \\ -x \text{ si } x < 0 \end{cases}$
Nous devons alors utiliser une instruction qui opère une disjonction de cas.
En **Python**, il s'agit de l'instruction de choix introduite par le mot-clé `if`. La syntaxe est alors la suivante :

```
def vabs(x):
    if x >= 0:
        return x
    else:
        return -x
```

```
print('f({})={}'.format(2, vabs(2)))
print('f({})={}'.format(-2, vabs(-2)))
```

```
f(2)=2
f(-2)=2
```

Observons la syntaxe de la structure de choix `if`. Tout d'abord, le mot `if` est suivi d'une condition de choix : quel est l'objet renvoyé par l'interpréteur lorsqu'il évalue cette condition ?

```
>>> 2 >= 0              # rép.: True
>>> b = 2 >= 0
>>> b, id(b), type(b)   # rép.: (True, 137009620, <class 'bool'>)
```

La condition « `2 >= 0` » est d'un type que nous n'avons pas encore rencontré, à savoir le type *booléen*[9]. Une variable booléenne ne peut prendre que deux valeurs : `True` (*vrai*) ou `False` (*faux*).

La condition de choix est vérifiée quand l'évaluation de cette condition renvoie le booléen `True` et l'interpréteur exécute alors la suite d'instructions qui se trouve dans le premier bloc d'instructions. Dans le cas contraire l'interpréteur saute au bloc d'instructions situé après le mot-clé `else` et délimité par l'indentation.

Il faut bien noter le rôle essentiel de l'*indentation* qui permet de délimiter chaque bloc d'instructions et la présence des deux points après la condition du choix et après le mot clé `else`.

Attardons-nous quelques instants sur les booléens. Pour obtenir une variable booléenne, on utilise en général, comme dans l'exemple précédent, un ou plusieurs *opérateurs de comparaison*. Ces opérateurs sont au nombre de huit :

`x == y`	x est égal à y
`x != y`	x est différent de y
`x > y`	x est strictement supérieur à y
`x < y`	x est strictement inférieur à y
`x >= y`	x est supérieur ou égal à y
`x <= y`	x est inférieur ou égal à y
`x is y`	`id(x) == id(y)`
`x in y`	x appartient à y (voir le type `list`)

Il faut bien distinguer l'instruction d'affectation « = » du symbole de comparaison « == ».

Pour exprimer des conditions complexes (par exemple $x > -2$ et $x^2 < 5$), on peut combiner des variables booléennes en utilisant les trois *opérateurs booléens* (par ordre de priorité croissante) : `or` (ou), `and` (et), et `not` (non).

Noter que **Python**, contrairement à beaucoup de langages, offre aussi quelques raccourcis syntaxiques agréables comme : `(x<y<z)` qui est un raccourci pour `(x<y) and (y<z)`.

9. Le nom booléen vient du nom du mathématicien anglais G. BOOLE (1815-1864).

```
>>> x = -1
>>> (x > -2) and (x**2 < 5)  # rép.: True
>>> (x <= -2) or (x**2 > 5)  # rép.: False
>>> not(x >= -2)             # rép.: False
>>> -2 < x <= 0              # rép.: True
```

Les opérateurs non booléens sont prioritaires sur les opérateurs de comparaison, qui sont prioritaires sur les opérateurs logiques. Cependant, pour améliorer la lisibilité du code, il est préférable d'utiliser des parenthèses lorsqu'on écrit des conditions complexes.
Il est parfois utile de définir des *fonctions booléennes*, c'est-à-dire qui renvoient toujours une valeur booléenne, pour effectuer un test complexe.
Voici deux versions d'une fonction booléenne qui prend en argument trois nombres et renvoie le booléen `True` s'ils sont rangés par ordre croissant :

```
def ordre(x, y, z):
    if x <= y <= z:
        return True
    else:
        return False
```

```
# ou encore plus simplement
def ordre(x, y, z):
    return x <= y <= z
```

Notons que lorsque les conditions et les instructions d'une structure conditionnelle sont assez brèves, on peut les écrire sur une seule ligne :

```
def valeur_absolue(x):
    return x if x >=0 else -x
```

En algorithmique, il est assez fréquent que l'on ait à enchaîner des disjonctions de cas. Au lieu d'utiliser des choix imbriqués, on peut présenter cet enchaînement en une succession de choix, cette deuxième rédaction représentant alors une alternative syntaxique plus agréable. Notons une fois de plus combien **Python** offre des facilités de syntaxe qui en font un langage pertinent pour l'apprentissage de la programmation.
Illustrons ces deux possibilités pour définir une fonction qui prend en argument trois nombres a, b et c et qui affiche une phrase indiquant le nombre de solutions réelles de l'équation du second degré $ax^2 + bx + c = 0$.

```
def nbre_solutions(a, b, c):
    if a == 0:
        print("L'équation est du premier degré.")
    else:
        delta = b**2 - 4*a*c
        if delta > 0:
            print("L'équation possède deux solutions réelles.")
        else:
            if delta == 0:
                print("L'équation possède une solution réelle.")
            else:
                print("L'équation ne possède pas de solution réelle.")

def nbre_solutions(a, b, c):
    delta = b**2 - 4*a*c
```

```
    if a == 0:
        print("L'équation est du premier degré.")
    elif delta > 0:
        print("L'équation possède deux solutions réelles.")
    elif delta == 0:
        print("L'équation possède une solution réelle.")
    else:
        print("L'équation ne possède pas de solution réelle.")
```

Dans la succession de choix, le mot-clé `elif` est une abréviation pour `else if`. Le principal intérêt de la deuxième syntaxe est surtout que les différents résultats potentiellement renvoyés par la fonction apparaissent avec le même décalage d'indentation, d'où une syntaxe plus lisible.

Exercice 1. Écrire une fonction `max2` qui renvoie le plus grand des deux nombres x et y. Construire une fonction `max3` qui calcule le maximum de 3 nombres.

Solution.

```
def max2(a, b):
    if a <= b: return b
    else: return a
```

```
def max3(a, b, c):
    return max2(a, max2(b, c))
```

Exercice 2. (Années bissextiles)

a) Écrire une fonction booléenne `bissextile(n)` qui permet de tester si une année est bissextile. On rappelle que les années bissextiles reviennent tous les 4 ans, sauf les années séculaires, si celles-ci ne sont pas multiples de 400. Ainsi, 1900 n'était pas une année bissextile, alors que 2000 l'était.

b) Écrire la même fonction en utilisant uniquement des opérateurs logiques (sans branchement conditionnel).

Solution.

```
def bissextile(n):
    if n % 4 != 0:
        return False
    elif n % 100 != 0:
        return True
    elif  n % 400 != 0:
        return False
    else:
        return True

def bissextile2(n):
    return ((n % 4 == 0) and (n % 100 != 0)) or (n % 400 == 0)
```

La position des parenthèses dans la fonction `bissextile2` est importante.

8 Les boucles `while`

Après avoir introduit les structures de choix, nous nous intéressons aux structures de répétition : ces structures permettent d'exécuter à plusieurs reprises un bloc d'instructions. Ces répétitions se classent en deux catégories :

- les répétitions conditionnelles : le bloc d'instructions est à répéter autant de fois qu'une condition est vérifiée.
- les répétitions inconditionnelles : le bloc d'instructions est répété un nombre donné de fois.

Commençons par les répétitions conditionnelles. Une telle répétition est introduite par le mot-clé `while` suivi de la condition ponctuée d'un deux-points, puis vient le bloc d'instructions à répéter, ce dernier étant délimité par l'indentation.
Par exemple, soit à écrire une fonction prenant en argument un entier N et renvoyant (sans recourir à la fonction logarithme) le plus petit entier n tel que 2^n est supérieur à l'entier N.

```
def essai(N):
    n = 0
    while 2**n <= N:
        n += 1
    return n
```

Le déroulement de la boucle est le suivant :

- on teste la condition ;
- si la condition est vérifiée, c'est-à-dire si l'évaluation de l'expression booléenne renvoie la valeur `True`, **Python** effectue les instructions et on revient au début de la boucle ;
- si elle n'est pas vérifiée, c'est-à-dire si l'évaluation de l'expression booléenne renvoie la valeur `False`, on sort de la boucle.

Si la condition mentionnée ne devient jamais fausse, le bloc d'instructions est répété indéfiniment et le programme ne se termine pas. Quand on écrit une boucle, il faut impérativement s'assurer qu'elle va réellement s'arrêter (dans les conditions normales d'utilisation). [10]
Notons que la fonction de l'exemple précédent peut être programmée différemment :

```
def essai(N):
    n = 0
    while True:
        if 2**n > N:
            return n
        n += 1
```

Évidemment, l'apparence de cette boucle est paradoxale car la condition de la boucle `while` ne varie jamais, et on pourrait avoir l'impression qu'elle ne va jamais se terminer ; mais c'est sans compter sur la possibilité de sortir en force des boucles par renvoi du résultat d'une fonction (grâce au mot-clé `return`).

10. Pour interrompre un programme (mal conçu) qui ne se termine pas, on utilise la combinaison de touches `Ctrl-C`.

Exercice 3.

a) Écrire une fonction `somme(n)` qui renvoie la somme des carrés des n premiers entiers.

b) Écrire une fonction `depasse(M)` qui, pour tout entier M, renvoie le plus petit entier n tel que $1^2+2^2+\cdots+n^2 \geq M$.

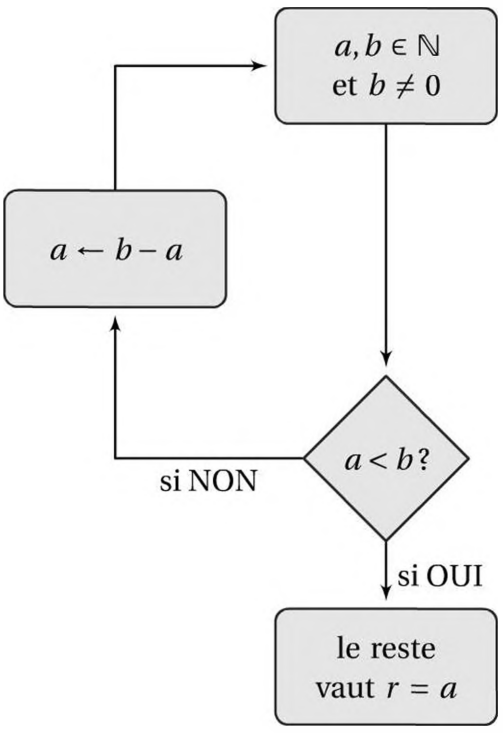
Solution.

```
def somme(n):
    s, i = 0, 0
    while i < n:
        i += 1
        s += i ** 2
    return s
```

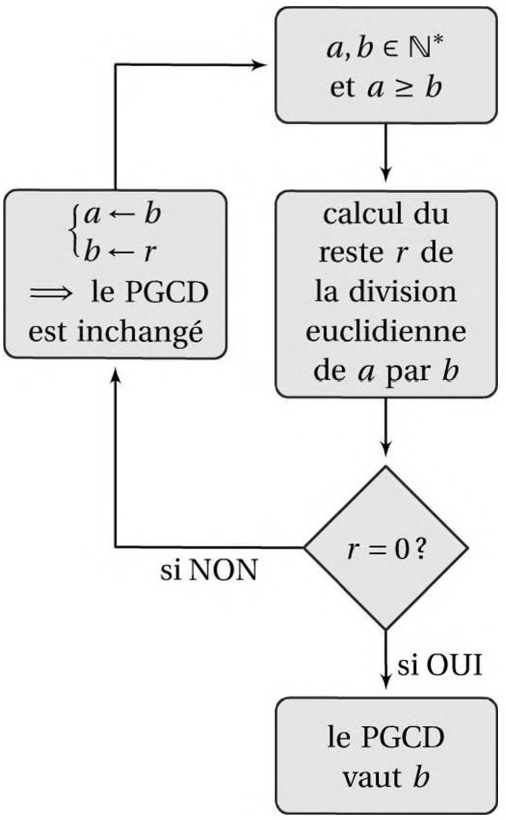
```
def depasse(M):
    s, i = 0, 0
    while s < M:
        i += 1
        s += i ** 2
    return i
```

Exercice 4. (Algorithme d'EUCLIDE)
Pour mémoire, on rappelle le principe de l'algorithme des soustractions successives pour le calcul du reste de la division euclidienne de deux entiers ainsi que l'algorithme d'EUCLIDE pour le calcul du PGCD (pour une démonstration de ces algorithmes, voir le chapitre 41 de [CBP10]).

Algorithme des différences successives



Algorithme d'Euclide



a) En utilisant l'algorithme des soustractions successives, écrire une fonction `reste(a,b)` qui renvoie le reste de la division euclidienne de `a` par `b`.

b) En utilisant l'algorithme d'EUCLIDE, écrire une fonction `pgcd(a,b)` qui renvoie le plus grand commun diviseur de deux entiers naturels `a` et `b`.

Solution.

```
def reste(a, b):
    if b == 0:
        return None
    while a >= b:
        a = a - b
    return a
```

```
def pgcd(a, b):
    while b > 0:
        a, b = b, reste(a, b)
    return a
```

Remarquons que dans les affectations parallèles `a, b = b, reste(a, b)`, les affectations sont *simultanées* et non *successives.*

Avant de nous intéresser aux répétitions inconditionnelles, nous aurons besoin d'introduire la notion de liste.

9 Les listes

Dès que l'on commence à manipuler un certains nombre de données, la notion de variable numérique s'avère insuffisante. Imaginons que l'on souhaite placer en mémoire les notes des élèves d'une classe en vue de déterminer la moyenne, l'écart-type et la médiane de cette série de notes. Une solution naïve consisterait à définir pour chaque élève une variable contenant sa note. Même pour une classe d'une trentaine d'élèves, il serait fastidieux de définir une trentaine de variables : `n1`, `n2`, `n3`, `n4`, etc. puis de calculer la moyenne avec une formule comme celle-ci : `moy = (n1 + n2 +···+ n30) / 30`
On voit combien il serait avantageux d'attribuer à une seule variable l'ensemble des notes, chaque note étant repérée par un indice.
En **Python**, la structure de données qui va nous sortir d'affaire s'appelle une *liste.*

```
>>> liste = ['2nde3', 12, 12.23, 18, 7, 15]
>>> liste, id(liste), type(liste)
(['2nde3', 12, 12.23, 18, 7, 15], 3072088364, <class 'list'>)
```

Comme on le voit dans l'exemple précédent, une liste est une séquence d'éléments, rangés dans un certain ordre ; de plus, en **Python**, une liste n'est pas nécessairement homogène : elle peut contenir des objets de types différents les uns des autres.
La première manipulation que l'on a besoin d'effectuer sur une liste, c'est d'en extraire un élément : la syntaxe est alors `liste[indice]`. Par exemple, cherchons à extraire un élément de notre liste :

```
>>> liste = [12, 11, 18, 7, 15, 3]; liste[2]      # rép.: 18
```

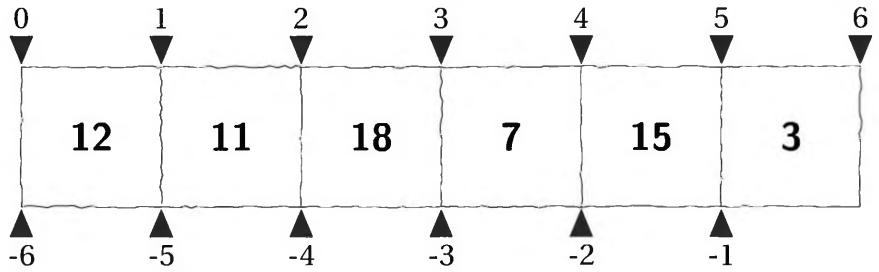
Le résultat peut surprendre : on aurait peut-être attendu comme réponse `11` au lieu de `18`. En fait, les éléments d'une liste sont indexés *à partir de 0* et non de 1.

```
>>> liste = [12, 11, 18, 7, 15, 3]
>>> liste[0], liste[1], liste[2], liste[3], liste[4], liste[5]
(12, 11, 18, 7, 15, 3)
>>> liste[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Si l'on tente d'extraire un élément avec un indice dépassant la taille de la liste, le message d'erreur « `IndexError: list index out of range` » est renvoyé.
On peut extraire une sous-liste en déclarant l'indice de début et l'indice de fin, séparés par deux-points. Cette opération est connue sous le nom de tranchage (*slicing* en anglais), ou encore « saucissonnage » (*most frenchy* !).
Essayons d'extraire la sous-liste comprenant le troisième, le quatrième et le cinquième élément :

```
>>> liste[2:4]        # rép.: [18, 7]
```

Encore une fois, le résultat n'est peut-être pas tout à fait celui que l'on attendait. En fait, pour bien comprendre les techniques de saucissonnage, il faut penser les indices comme repérant non pas les tranches de saucisson, mais les coups d'opinel qui ont permis de couper les tranches (en partant évidemment de 0 pour l'entame).



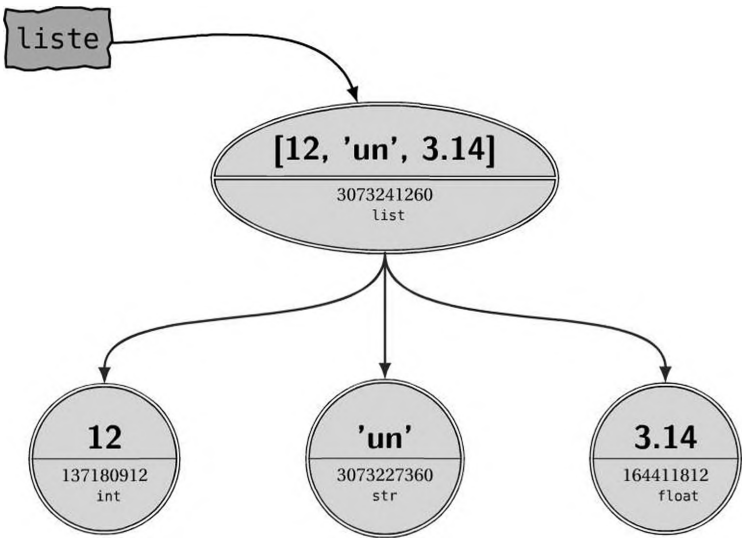
Remarquez que l'on peut même se servir d'indices négatifs. À présent, les résultats suivants doivent devenir limpides :

```
>>> liste[2:]
[18, 7, 15, 3]
>>> liste[:2]
[12, 11]
>>> liste[0:len(liste)]
[12, 11, 18, 7, 15, 3]
>>> liste[:] # le même en mieux
[12, 11, 18, 7, 15, 3]
>>> liste[2:5]
[18, 7, 15]
```

```
>>> liste[2:7]
[18, 7, 15, 3]
>>> liste[-2:-4]
[]
>>> liste[-4:-2]
[18, 7]
>>> liste[len(liste)-1]
3
>>> liste[-1] # le même en mieux
3
```

Pour extraire le dernier élément d'une liste, l'instruction `liste[-1]` remplace avantageusement l'instruction `liste[len(liste)-1]` !
À noter que lorsqu'on utilise des tranches, les dépassements d'indices sont licites.
À présent, examinons où sont rangés les éléments d'une liste.

```
>>> liste = [12, 'un', 3.14]
>>> liste, id(liste), type(liste)
([12, 'un', 3.14], 3073241260, <class 'list'>)
>>> liste[0], id(liste[0]), type(liste[0])
(12, 137180912, <class 'int'>)
>>> liste[1], id(liste[1]), type(liste[1])
('un', 3073227360, <class 'str'>)
>>> liste[2], id(liste[2]), type(liste[2])
(3.14, 164411812, <class 'float'>)
```

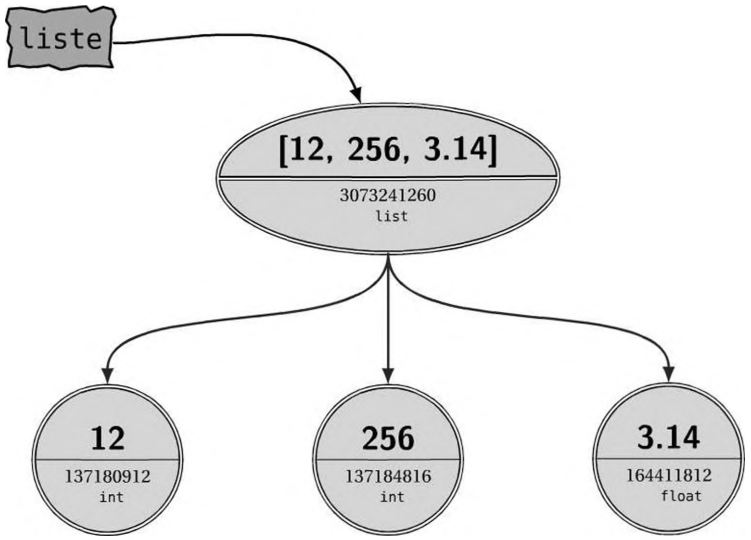


On constate donc qu'une liste fonctionne comme un carnet d'adresses qui contient les emplacements en mémoire des différents éléments de la liste.

En **Python**, les listes sont des objets *modifiables*[11], c'est-à-dire qu'on peut modifier un ou plusieurs éléments de la liste.

```
 >>> liste[1] = 2**8
>>> liste, id(liste), type(liste)
([12, 256, 3.14], 3073241260, <class 'list'>)
>>> liste[0], id(liste[0]), type(liste[0])
(12, 137180912, <class 'int'>)
>>> liste[1], id(liste[1]), type(liste[1])
(256, 137184816, <class 'int'>)
>>> liste[2], id(liste[2]), type(liste[2])
(3.14, 164411812, <class 'float'>)
```

11. On parle aussi d'objet *mutable* : un tel objet peut changer de valeur sans changer d'identifiant !



Remarquez bien que la liste n'a pas changé d'identifiant. Autrement dit, la liste `liste` est le même objet que précédemment, on en a seulement modifié la *valeur* !
Les techniques de saucissonnage peuvent être utilisées pour les modifications de listes, comme le montrent les exemples suivants.

```
>>> liste = [12, 11, 18, 7, 15, 3]
>>> liste[2:5] = ['a', 'b', 'c']
>>> liste
[12, 11, 'a', 'b', 'c', 3]
>>> liste = [12, 11, 18, 7, 15, 3]
>>> liste[2:5] = ['a', 'b', 'c']
>>> liste
[12, 11, 'a', 'b', 'c', 3]
>>> liste[2:5] = [100, 200]
>>> liste
[12, 11, 100, 200, 3]
>>> liste[2:4] = ['un', 'deux', 'trois', 'quatre']
>>> liste
[12, 11, 'un', 'deux', 'trois', 'quatre', 3]
```

Il est assez fréquent de copier des listes. Mais là encore, **Python** nous réserve quelques surprises...

```
>>> liste = [12, 'un', 3.14]
>>> copie = liste
>>> liste[0] = 100  # modification de l'objet liste
>>> liste, copie
([100, 'un', 3.14], [100, 'un', 3.14])
>>> copie[1] = 'ah ?'  # modification de l'objet copie
>>> liste, copie
([100, 'ah ?', 3.14], [100, 'ah ?', 3.14])
```

Pour bien comprendre ce qui s'est passé, examinons une fois de plus les identifiants :

```
>>> liste, id(liste), type(liste)
([12, 'un', 3.14], 3071856812, <class 'list'>)
```

```
>>> copie, id(copie), type(copie)
([12, 'un', 3.14], 3071856812, <class 'list'>)
```

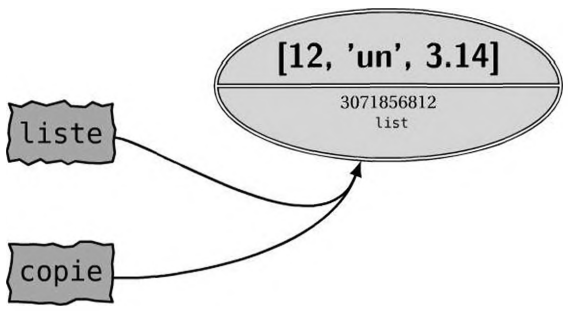
En fait, nous avons créé un nouvel alias nommé `copie` pour notre liste. Autrement dit, les variables `liste` et `copie` « pointent » vers le même objet. Ainsi, lorsqu'on modifie la valeur de l'objet, la modification sera visible depuis les deux alias ! Ce mécanisme a été implémenté par les développeurs de **Python** dans un souci évident d'optimisation de la mémoire.
Mais alors, cela signifie-t-il qu'on ne pourra pas copier une liste en obtenant un objet distinct et indépendant de la liste copiée pour, par exemple, modifier la nouvelle liste tout en gardant en mémoire une copie de la version de départ ?
Évidemment, cette possibilité existe, mais il faut alors extraire de la liste la sous-liste en entier, avant de la réaffecter à une autre variable. Comme on l'a vu, extraire toute la liste d'une liste se fait avec l'écriture `liste[:]` et on écrira alors :

```
copie = liste[:] # équivaut en fait à l'instruction : >>> copie = list(liste)
```

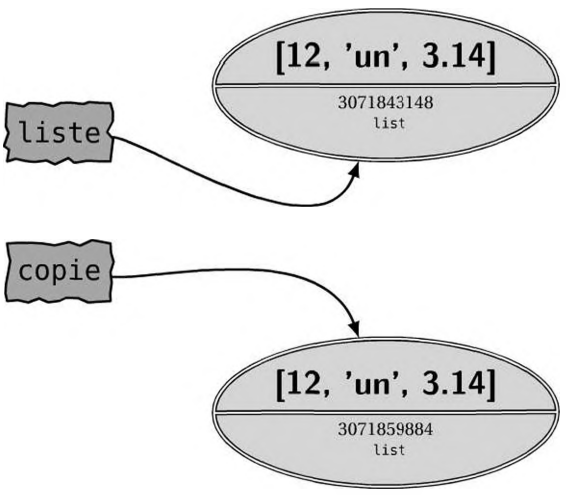
Dans ce cas, **Python** effectue une *copie superficielle* de la liste (en anglais, *shallow copy*). En fait, ce ne sont pas les objets eux-mêmes qui sont copiés, mais les pointeurs vers les objets copiés. D'où le nom de *copie de références* (ou *copie de pointeurs*) encore donné à ce type de copie.

```
>>> copie = liste[:]
>>> liste[0] = 100
>>> liste, copie
([100, 'un', 3.14], [12, 'un', 3.14])
>>> copie[1] = 'ah ?'
>>> liste, copie
([100, 'un', 3.14], [12, 'ah ?', 3.14])
>>> liste, id(liste), type(liste)
([100, 'un', 3.14], 3071843148, <class 'list'>)
>>> copie, id(copie), type(copie)
([12, 'ah ?', 3.14], 3071859884, <class 'list'>)
```

Création d'un alias : `>>> copie=liste`



Copie superficielle : `>>> copie=liste[:]`



Cette notion de copie superficielle réserve quelques surprises. En effet, lorsque la liste contient des objets modifiables (comme des listes), ce sont bien les références de ces listes qui sont copiées, et non les objets eux-mêmes. Ce qui permet de comprendre les résultats suivants :

```
>>> x = ['a', 'b', [4, 5, 6]]
>>> y = x          # création d'un alias
>>> z = x[:]       # copie superficielle
>>> x[0] = 'c'     # affecte x et y mais pas z
>>> z[2][2] = 365  # affecte x, y et z !
>>> x; y; z
['c', 'b', [4, 5, 365]]
['c', 'b', [4, 5, 365]]
['a', 'b', [4, 5, 365]]
```

Pour effectuer une copie vraiment indépendante d'une liste (ou d'un objet en général), on peut utiliser la fonction `deepcopy` du module `copy` : cette fonction permet de copier récursivement tous les objets d'une liste ; d'où le nom de *copie profonde* ou de *copie récursive* (en anglais, *deep copy*).

```
>>> import copy
>>> x = ['a', 'b', [4, 5, 6]]
>>> y = x                 # création d'un alias
>>> z = x[:]              # copie superficielle
>>> w = copy.deepcopy(x)  # copie profonde
>>> x[0] = 'c'            # affecte x et y mais ni z, ni w
>>> z[2][2] = 365         # affecte x, y et z mais pas w
>>> x; y; z; w
['c', 'b', [4, 5, 365]]
['c', 'b', [4, 5, 365]]
['a', 'b', [4, 5, 365]]
['a', 'b', [4, 5, 6]]
```

Un examen attentif des identifiants permet de mieux comprendre la gestion de la mémoire effectuée par **Python** au cours de cette suite d'instructions :

```
>>> for l in ['x', 'y', 'z', 'w']:
...     print('id({}) = {}'.format(l, id(eval(l))))
...     for t in eval(l):
...         print('    id({}) = {}'.format(t, id(t)))
...
```

```
id(x) = 3072672108
    id(c) = 3073232288
    id(b) = 3073285696
    id([4, 5, 365]) = 3071864268
id(y) = 3072672108
    id(c) = 3073232288
    id(b) = 3073285696
    id([4, 5, 365]) = 3071864268
```

```
id(z) = 3071786220
    id(a) = 3072939136
    id(b) = 3073285696
    id([4, 5, 365]) = 3071864268
id(w) = 3072672172
    id(a) = 3072939136
    id(b) = 3073285696
    id([4, 5, 6]) = 3071864748
```

Nous avons vu trois sortes de copies de listes. Autre opération très courante : la concaténation de deux listes à l'aide de l'opérateur + :

```
>>> [1, 2] + [4, 7, 9]     # rép.: [1, 2, 4, 7, 9]
```

Autre modification très courante : rajouter un élément en fin de liste. La première méthode qui vient à l'esprit est la suivante :

```
>>> liste = [12, 11, 18, 7, 15, 3]
>>> liste, id(liste), type(liste)
([12, 11, 18, 7, 15, 3], 3071843148, <class 'list'>)
>>> liste = liste + [5]
>>> liste, id(liste), type(liste)
([12, 11, 18, 7, 15, 3, 5], 3071858764, <class 'list'>)
```

L'inconvénient de cette méthode est qu'elle est coûteuse du point de vue de la gestion de la mémoire : en fait, l'interpréteur a effectué une copie profonde de la première liste, y a ajouté l'élément 5 et a réaffecté la variable `liste`. Pour éviter d'avoir à copier implicitement l'objet `liste`, on peut employer la syntaxe suivante, qui ne fait que modifier la valeur de l'objet `liste` :

```
>>> liste = [12, 11, 18, 7, 15, 3]
>>> liste, id(liste), type(liste)
([12, 11, 18, 7, 15, 3], 3071843148, <class 'list'>)
>>> liste += [5]
>>> liste, id(liste), type(liste)
([12, 11, 18, 7, 15, 3, 5], 3071843148, <class 'list'>)
```

Une troisième possibilité nous est donnée par la *méthode* `append()`. Nous avons déjà employé couramment le mot « objet » sans pour autant le définir précisément ; cependant, nous avons dit qu'un objet possède toujours une valeur, un identifiant et un type. Certains objets **Python** possèdent en outre un certain nombre de fonctions qui ne s'appliquent qu'à un type donné d'objets ; on parle alors de *méthode* associée à un objet. Parmi les nombreuses méthodes que possède une liste, il y en a une qui permet d'ajouter un élément en fin de liste, c'est la méthode `append()`. Voici comment l'utiliser :

```
>>> liste = [12, 11, 18, 7, 15, 3]
>>> liste, id(liste), type(liste)
([12, 11, 18, 7, 15, 3], 3071858764, <class 'list'>)
>>> liste.append(5)
>>> liste, id(liste), type(liste)
([12, 11, 18, 7, 15, 3, 5], 3071858764, <class 'list'>)
```

Remarquons cette syntaxe qui peut surprendre la première fois qu'on la rencontre ; il s'agit de la notation point : l'instruction `liste.append(5)` signifie que l'on modifie la valeur de l'objet `liste` en lui appliquant la méthode `append()` avec comme paramètre effectif 5. De plus, étant donné que la méthode `append()` renvoie comme valeur l'objet `None`, ce serait une erreur de rajouter une opération d'affectation.

Parmi les trois méthodes que l'on vient de citer pour ajouter un élément à une liste, c'est l'utilisation de la méthode `append()` qui est la plus efficace du point de vue de l'utilisation de la mémoire ; elle est donc à privilégier.

Enfin, parmi les méthodes associées aux listes, mentionnons les plus utiles :

la méthode	son effet
`list.append(x)`	ajoute l'élément `x` en fin de liste
`list.extend(L)`	ajoute en fin de liste les éléments de L
`list.insert(i, x)`	insère un élément `x` en position `i`
`list.remove(x)`	supprime la première occurrence de `x`
`list.pop([i])`	supprime l'élément d'indice `i` et le renvoie
`list.index(x)`	renvoie l'indice de la première occurrence de `x`
`list.count(x)`	renvoie le nombre d'occurrences de `x`
`list.sort()`	modifie la liste en la triant
`list.reverse()`	modifie la liste en inversant l'ordre des éléments

Remarquons l'équivalence : `liste.insert(i, x)` $\Longleftrightarrow$ `liste[i:i] = [x]`
Parallèlement à la méthode `sort()` (qui *remplace* la liste par une liste triée), il existe une *primitive*[12] `sorted` qui renvoie à partir d'une liste donnée une liste triée, mais ne modifie pas la liste prise en argument.

```
>>> liste = [3, 1, 4, 2]
>>> sorted(liste)
[1, 2, 3, 4]
```

```
>>> liste; liste.sort(); liste
[3, 1, 4, 2]
[1, 2, 3, 4]
```

Avant d'aborder la structure de choix inconditionnelle, introduisons une primitive qui nous sera très utile pour la suite, à savoir la fonction `len` (du mot anglais *length*) qui renvoie la longueur d'une liste. Notons de plus que la suppression d'un élément peut s'effectuer à l'aide de la fonction `del` (du mot anglais *delete*).

```
>>> liste = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> len(liste)     # rép.: 7
>>> del liste[4]
>>> liste, len(liste)
(['a', 'b', 'c', 'd', 'f', 'g'], 6)
```

Exercice 5. Écrire une fonction booléenne qui cherche si un élément existe dans une liste.

Solution.
Évidemment, pour une bonne programmation, le balayage de la liste doit s'arrêter à la première occurrence de x éventuellement trouvée.

```
def existe(x, liste):
    for y in liste:
        if x == y: return True
    return False
```

La fonction précédente est une réécriture de l'opérateur infixe `in` :

```
>>> liste = [1, 3, 0]; (1 in liste, 2 in liste)
(True, False)
```

12. Une fonction *primitive* désigne une fonction de base fournie par le langage. cf. section 14 page 37.

10 Les boucles for

Lorsque l'on souhaite répéter un bloc d'instructions un nombre déterminé de fois, on peut utiliser un *compteur actif*, c'est-à-dire une variable qui compte le nombre de répétitions et conditionne la sortie de la boucle `while`. C'est le cas de l'exemple suivant où l'on définit une fonction qui prend en argument un entier `n` et affiche `n` fois le même message.

```
def f(n):
    i = 0 # on initialise le compteur i
    while i < n:
        print('Je me répète {} fois.'
              ' (i={})'.format(n, i))
        i += 1 # on incrémente i
f(5)
```

```
Je me répète 5 fois. (i=0)
Je me répète 5 fois. (i=1)
Je me répète 5 fois. (i=2)
Je me répète 5 fois. (i=3)
Je me répète 5 fois. (i=4)
```

Pour effectuer une telle répétition, on dispose d'une structure de répétition nous économisant d'une part l'initialisation du compteur (`i = 0`), et d'autre part son incrémentation (`i += 1`) : c'est la structure introduite par le mot-clé `for` :

```
def f(n):
    for i in range(n):
        print('Je me répète {} fois.'
              ' (i={})'.format(n, i))
f(5)
```

```
Je me répète 5 fois. (i=0)
Je me répète 5 fois. (i=1)
Je me répète 5 fois. (i=2)
Je me répète 5 fois. (i=3)
Je me répète 5 fois. (i=4)
```

Nous voyons apparaître ici pour la première fois la fonction `range`. Cette fonction crée un *itérateur*, c'est-à-dire en quelque sorte un distributeur d'entiers consécutifs [13]. Au lieu de créer et garder en mémoire une liste d'entiers, cette fonction génère les entiers au fur et à mesure des besoins, toujours dans un souci d'optimisation de la mémoire.
Avec un argument, `range(n)` renvoie un itérateur parcourant l'intervalle $[\![0, n-1]\!]$; avec deux arguments, `range(n, p)` parcourt l'intervalle $[\![n, p-1]\!]$; enfin employée avec trois arguments, `range(n, p, k)` parcourt l'intervalle $[\![n, p-1]\!]$ avec un pas égal à k.
Signalons que la fonction `range` peut servir à créer des listes d'entiers, moyennant une conversion de type opérée par la fonction `list`. Examinons quelques exemples.

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -6, -1))
[0, -1, -2, -3, -4, -5]
```

```
>>> list(range(0)), list(range(1, 0))
[] []
```

```
>>> list(range(9, 0, -1))
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

13. Plus précisément, un itérateur est un objet possédant une méthode `__iter()__` qui renvoie les éléments d'une collection un par un et qui provoque une exception du type `StopIteration` lorsqu'il n'y a plus d'éléments.

Autre point important, on peut aussi utiliser comme itérateur dans une boucle `for` une liste ou une chaîne de caractère [14].

```
for i in [10, 'a', 1.414]:
    print(i, end=' -> ')
for i in 'mot':
    print(i, end=' -> ')
```

```
10 -> a -> 1.414 -> m -> o -> t ->
```

Exercice 6. Écrire deux fonctions `somme` et `maximum` qui renvoient respectivement la somme et le maximum d'une liste d'entiers. Vérifiez votre résultat en vous servant des primitives `sum` et `max`.

Solution.

```
def somme(liste):
    som = 0
    for data in liste:
        som += data
    return som
```

```
def maximum(liste):
    maxi = liste[0]
    for data in liste:
        if data > maxi:
            maxi = data
    return maxi
```

```
# Les tests :
from random import sample
liste = sample(range(100), 30)
print(liste, somme(liste), sum(liste), maximum(liste), max(liste))
```

Exercice 7. Programmer « à la main » des fonctions qui imitent les méthodes `remove`, `pop`, `index`, `count` et `reverse` des listes.
Essayez les fonctions en effectuant les tests suivants :

```
>>> liste = [1, 2, 3, 2, 4, 5, 3, 7, 8, 10, 11, 3]
>>> index(liste, 3)                                    # rép.: 2
>>> remove(liste, 3)
>>> liste                      # rép.: [1, 2, 2, 4, 5, 3, 7, 8, 10, 11, 3]
>>> pop(liste, 9), liste       # rép.: 11 [1, 2, 2, 4, 5, 3, 7, 8, 10, 3]
>>> count(liste, 3)                                    # rép.: 2
>>> reverse(liste)
>>> liste                         # rép.: [3, 10, 8, 7, 3, 5, 4, 2, 2, 1]
```

On prendra soin de vérifier que la valeur de la liste d'origine est bel et bien modifiée par les fonctions `remove`, `pop`, `reverse`. On précise que si au cours de l'exécution d'une fonction, une variable n'est pas trouvée dans la table des variables locales à la fonction, l'interpréteur consulte la table des variables globales.

Solution.

14. Plus généralement, le mot-clé `for` peut être utilisé avec n'importe quel objet *itérable* ; un objet itérable étant un objet possédant soit une méthode `__iter__()` soit une méthode `__getitem__()`.

```
def index(L, x):
    for i in range(len(L)):
        if L[i] == x:
            return i
    return -1

def remove(L, x):
    i = index(L, x)
    if i >= 0:
        del L[i]

def pop(L, i):
    x = L[i]
    del L[i]
    return x
```

```
def count(L, x):
    compteur = 0
    for data in L:
        if data == x:
            compteur += 1
    return compteur

def reverse(L):
    liste = L[:]
    for i in range(len(L)):
        L[i] = liste[len(L)-1-i]
```

Pour créer des listes, **Python** fournit une facilité syntaxique particulièrement agréable, à savoir les listes (définies) par compréhension (en anglais, *list-comprehensions*). Elles permettent de générer des listes d'une manière très concise, sans avoir à utiliser de boucles [15].
La syntaxe pour définir une liste par compréhension est proche de celle utilisée en mathématiques pour définir un ensemble par compréhension :

$$\{ \; f(x) \;\; | \;\; x \;\; \in \;\; E \;\; \}$$

```
[ f(x)  for  x  in  liste ]
```

Voici quelques exemples :

```
>>> liste = [2, 4, 6, 8, 10]
>>> [3*x for x in liste]
[6, 12, 18, 24, 30]
>>> [[x, x**3] for x in liste]
[[2, 8], [4, 64], [6, 216], [8, 512], [10, 1000]]
>>> [3*x for x in liste if x > 5] # on filtre avec une condition
[18, 24, 30]
>>> [3*x for x in liste if x**2 < 50] # idem
[6, 12, 18]
>>> liste2 = list(range(3))
>>> [x*y for x in liste for y in liste2]
[0, 2, 4, 0, 4, 8, 0, 6, 12, 0, 8, 16, 0, 10, 20]
```

Voici par exemple, un moyen efficace d'obtenir la liste des années bissextiles dans un intervalle donné :

```
>>> bissextile = [b for b in range(2000, 2100)
...               if (b % 4 ==0 and b % 100 != 0) or (b % 400 == 0)]
```

15. Les listes définies par compréhension remplacent simultanément un « mappage » et un « filtrage » comme il y en a dans tous les langages fonctionnels.

Pour concaténer les éléments d'une liste de listes, on peut imbriquer deux boucles `for` dans la définition d'une liste par compréhension :

```
>>> xll = [[1, 2, 3], [4, 5], [6, 7, 8]]
>>> [x for xl in xll for x in xl]
[1, 2, 3, 4, 5, 6, 7, 8]
```

Pour obtenir les diviseurs d'un entier n, on peut également utiliser une liste par compréhension :

```
>>> n = 100; [d for d in range(1, n+1) if n % d == 0]  # les diviseurs de 100
[1, 2, 4, 5, 10, 20, 25, 50, 100]
```

11 Récapitulatif sur les principaux types

Passons en revue les types de variables que nous avons rencontrés au long de ce premier chapitre, et mentionnons quelques nouveaux types.

a) Parmi, les *types scalaires*, citons :

- le type entier (`int`), réservé à une variable dont la valeur est un entier relatif stocké en valeur *exacte*. Les valeurs possibles pour une telle variable ne sont limitées que par les capacités de l'ordinateur ;
- le type flottant (`float`) : un nombre à virgule flottante correspond à une variable dont la valeur est un nombre réel, stocké en valeur *approchée* sous la forme d'un triplet (s, m, e) où s désigne le signe (dans $\{-1, 1\}$), m la mantisse (aussi appelée significande) et e l'exposant. Un tel triplet représente le nombre décimal $s \cdot m \cdot b^e$ en notation scientifique où b est la base de représentation, à savoir : 2 sur les ordinateurs. En faisant varier e, on fait « flotter » la virgule décimale.
  Les nombres réels sont stockés en **Python** selon le format « double précision » spécifié par la norme IEEE 754. Ainsi, le signe est codé sur 1 bit, l'exposant sur 11 bit et la mantisse sur 53 bits dont le 1 à gauche de la virgule est omis, soit $11 + 1 + 53 - 1 = 64$ bits ; la précision étant de 53 bits, soit 16 chiffres significatifs.
  Les calculs en virgule flottante sont pratiques, mais présentent divers désagréments, notamment :
  - leur précision limitée, qui se traduit par des arrondis (dus aux opérations, ainsi qu'aux changements de base implicites, si la base est différente de 10) qui peuvent s'accumuler de façon gênante. En particulier, la soustraction de deux nombres très proches provoque une grande perte de précision relative : on parle de *cancellation*.
  - une plage d'exposants limitée, pouvant donner lieux à des *overflows* (lorsque le résultat d'une opération est plus grand que la plus grande valeur représentable) et à des *underflows* (lorsqu'un résultat est plus petit, en valeur absolue, que le plus petit flottant normalisé positif), puis à des résultats n'ayant plus aucun sens.

  Nous reviendrons sur ces objets à la section 1 du chapitre 4 ;

- le type complexe (`complex`), cité pour mémoire, correspond à une variable dont la valeur est un nombre complexe, stocké comme un couple de `float` (donc en valeur approchée). Le nombre complexe $3+4i$ est noté `3+4J`, le nombre i est noté `1J` (on peut remplacer `J` par `j`) ;
- le type booléen (`bool`).

b) Parmi les types séquentiels, il faut distinguer les types modifiables (en anglais « *mutables* ») des types non modifiables (en anglais « *immutables* ») :

- les listes (`list`) sont des objets séquentiels modifiables ;
- les chaînes de caractères (`str`) sont des objets séquentiels non modifiables. Les techniques de « slicing » s'appliquent aux chaînes de caractères ;
- les objets de la classe `tuple` (que l'on pourrait traduire en français par « t-uplets ») sont des objets séquentiels qui s'apparentent quelque peu aux listes, mais qui pour leur part ne sont pas modifiables. Pour définir un t-uplet, on place entre parenthèses ses éléments. Cependant les parenthèses sont facultatives, et nous les avons utilisées fréquemment sans le dire lorsque par souci de gain de place nous avons affiché simultanément le contenu de plusieurs variables en les plaçant sur une même ligne de commande et en les séparant par des virgules, ou lorsque nous avons effectué des affectations simultanées ;
- les dictionnaires (`dict`) sont des tableaux associatifs, c'est-à-dire des collections modifiables d'objets indexés par des objets de n'importe quel type ;
- les ensembles (`set`) sont des ensembles d'éléments uniques (l'ordre des éléments n'étant pas précisé par l'utilisateur, mais choisi par l'interpréteur).

Voici quelques exemples tirés du tutoriel officiel de **Python**.

```
>>> x = 1, 2 # exemple de t-uplet
>>> x, id(x), type(x)
((1, 2), 3072478316, <class 'tuple'>)
>>> tel = {'jack': 4098, 'sape': 4139} # exemple de dictionnaire
>>> tel['guido'] = 4127; tel, id(tel), type(tel)
({'sape': 4139, 'jack': 4098}, 3073177636, <class 'dict'>)
>>> fruits = {'apple', 'orange', 'apple', 'pear', 'orange'} # exemple d'ensemble
>>> fruits, id(fruits), type(fruits)
({'orange', 'pear', 'apple'}, 3073170428, <class 'set'>)
```

Exercice 8.
Écrire une fonction `trouve(mot, lettre)` qui renvoie l'indice de la première occurrence de `lettre` dans la chaîne de caractères `mot` et renvoie -1 si `lettre` n'apparaît pas dans `mot`.

Solution. La boucle s'arrête dès que le premier `return` est rencontré.

```
def trouve(mot, lettre):
    indice = 0
    while indice < len(mot):
        if mot[indice] == lettre:
            return indice
        indice += 1
    return -1
```

Remarquons que les objets de type séquentiel ont en commun les opérations résumées dans le tableau suivant où `s` et `t` désignent deux séquences du même type et `i`, `j`, `k` des entiers :

l'opération	son effet
`x in s`	`True` si `s` contient `x`, `False` sinon
`x not in s`	`True` si `s` ne contient pas `x`, `False` sinon
`s + t`	concaténation de `s` et `t`
`s * n, n * s`	`n` copies (superficielles) concaténées de `s`
`s[i]`	`i`-ième élément de `s` (à partir de 0)
`s[i:j]`	tranche de `s` de `i` (inclus) à `j` (exclu)
`s[i:j:k]`	tranche de `s` de `i` à `j` avec un pas de `k`
`len(s)`	longueur de `s`
`max(s), min(s)`	plus grand, plus petit élément de `s`
`s.index(i)`	indice de la 1re occurence de `i` dans `s`
`s.count(i)`	nombre d'occurences de `i` dans `s`

12 Quelques mots sur la récursivité

On appelle *fonction récursive* une fonction qui comporte un appel à elle-même.
Plus précisément, une fonction récursive doit respecter ce qu'on pourrait appeler les « trois lois de la récursivité » :

a) Une fonction récursive contient un cas de base.
b) Une fonction récursive doit modifier son état pour se ramener au cas de base.
c) Une fonction récursive doit s'appeler elle-même.

Une fonction mathématique définie par une relation de récurrence (et une condition initiale, bien entendu), peut de façon naturelle être programmée de manière récursive.
Par exemple, pour un réel x fixé, x^n peut se définir, comme fonction de n, par récurrence à partir des relations :

$$x_0 = 1 \quad \text{et} \quad x^n = x \cdot x^{n-1} \text{ si } n > 1$$

Le programme en **Python** s'écrit :

```
def puissance(x,n):
    if n == 0: return 1
    else: return x * puissance(x, n - 1)
```

Modifions légèrement ce programme, pour bien en comprendre l'exécution :

puissance.py

```
def puissance(x,n):
  if n == 0: return 1
  else:
    print('--'*n + '> appel de puissance({},{})'.format(x, n-1))
    y = x * puissance(x, n - 1)
    print('--'*n + '> sortie de puissance({},{})'.format(x, n-1))
    return y

puissance(2,5)
```

```
----------> appel de puissance(2,4)
--------> appel de puissance(2,3)
------> appel de puissance(2,2)
----> appel de puissance(2,1)
--> appel de puissance(2,0)
--> sortie de puissance(2,0)
----> sortie de puissance(2,1)
------> sortie de puissance(2,2)
--------> sortie de puissance(2,3)
----------> sortie de puissance(2,4)
```

La machine applique la règle : `puissance(x, n) = x * puissance(x, n-1)` tant que l'exposant est différent de 0, ce qui introduit des calculs intermédiaires jusqu'à aboutir au cas de base : `puissance(x,0) = 1`. Les calculs en suspens sont alors achevés dans l'ordre inverse jusqu'à obtenir le résultat final.
Pour élever un nombre à la puissance n, il existe un algorithme bien plus performant que la méthode naïve consistant à multiplier ce nombre $(n-1)$ fois par lui-même : il s'agit de la méthode dite d'*exponentiation rapide*[16]. Étant donnés un réel positif a et un entier n, on remarque que :

$$a^n = \begin{cases} \left(a^{\frac{n}{2}}\right)^2 & \text{si } n \text{ est pair,} \\ a \cdot \left(a^{\frac{n-1}{2}}\right)^2 & \text{si } n \text{ est impair.} \end{cases}$$

Cet algorithme se programme naturellement par une fonction récursive :

exponentiation.py

```
def expo(a, n):
    if n == 0: return 1
    else:
        if n%2 == 0: return expo(a, n//2)**2
        else: return a*(expo(a, (n-1)//2)**2)
```

Exercice 9. Écrire une version itérative, puis une version récursive pour calculer la factorielle d'un entier :

$$n! = \begin{cases} 1 \text{ si } n = 0 \\ (n-1)! \times n \text{ si } n \geq 1 \end{cases}$$

Solution.

```
def factIter(n):
    p = 1
    for i in range(2, n+1):
        p *= i
    return p
```

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

16. En fait, c'est l'algorithme d'exponentiation rapide qui est à la base de l'implémentation de l'opérateur puissance en **Python**. Avec cette méthode, on effectue $\log_2(n)$ élévations au carré et, au plus, $\log_2(n)$ multiplications, contre $n-1$ multiplications dans le cas de la méthode naïve ! Pour diverses variantes, cf. http://en.wikipedia.org/wiki/Exponentiation_by_squaring

Parmi les algorithmes élémentaires à connaître absolument, citons la méthode de HORNER qui permet d'évaluer efficacement un polynôme en une valeur donnée.
Supposons que l'on souhaite calculer la valeur en x_0 du polynôme

$$p(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n$$

La première idée pour évaluer p en x_0 est de calculer chaque puissance de x_0 de manière naïve, de multiplier par les coefficients, puis de tout additionner (ce qui nécessite $n + (n-1) + \cdots + 2 + 1 = n(n+1)/2$ produits).
On peut certes diminuer le nombre de multiplications à effectuer en utilisant l'algorithme d'exponentiation rapide [17].
La méthode de HORNER permet de réduire encore le nombre de multiplications, en remarquant que :

$$p(x_0) = ((\cdots((a_n x_0 + a_{n-1})x_0 + a_{n-2})x_0 + \cdots)x_0 + a_1)x_0 + a_0$$

Ce faisant, il n'y a plus que n multiplications à effectuer !
Pour programmer cet algorithme, il suffit de calculer les n valeurs de la suite b_n définie par :

$$\begin{cases} b_n = a_n \\ \forall k \in [\![0, n-1]\!], \ b_k = a_k + b_{k+1} x_0 \end{cases}$$

Voici une programmation récursive de l'algorithme de HORNER (le polynôme p étant représenté par la liste de ses coefficients $[a_0, \cdots, a_n]$) :

horner.py

```
def horner_rec(p, x):
    if len(p) == 1:
        return p[0]
    p[-2] += x * p[-1]
    return horner_rec(p[:-1], x)
```

Si vous avez testé les fonctions récursives précédentes avec des paramètres nécessitant un très grand nombre d'appels de la fonction, vous aurez sans doute remarqué que, par défaut, **Python** limite le nombre d'appels à 1000. Si l'on souhaite augmenter la taille limite de la pile d'appels récursifs, il suffit d'utiliser les instructions suivantes :

```
import sys
sys.setrecursionlimit(100000)
```

13 Quelques méthodes pour trier une liste

Dans cette section, on s'intéresse à un problème très courant en informatique, à savoir celui de trier une liste. On donne ici trois algorithmes de tri. Pour tester chacun de ces algorithmes, on pourra utiliser des listes générées aléatoirement en important le module `random` (cf. section 2.2 du chapitre 2) :

```
from random import randrange
n = 20; liste = [randrange(100) for i in range(n)]
```

17. Ceci ramène à un nombre de multiplications dominé par $n \log_2 n$.

Tri par sélection

Le *tri par sélection* est une des méthodes de tri les plus simples. On commence par rechercher le plus grand élément du tableau que l'on va mettre à sa place, c'est-à-dire l'échanger avec le dernier élément. Puis on recommence avec le deuxième élément le plus grand, etc.

a) Écrire une fonction `maxi(l, n)` qui prend en argument une liste `l` et un nombre `n` et qui retourne le maximum et son indice parmi les `n` premiers éléments de `l`.

b) En utilisant la fonction `maxi`, écrire une fonction `tri_selec` qui trie une liste selon la méthode de tri par sélection.

Solution.

tri.py

```
def maxi(l, n):
    indice = 0
    for i in range(n):
        if l[i] > l[indice]:
            indice = i
    return [l[indice], indice]
```

tri.py

```
def tri_selec(l):
    i = len(l) - 1
    while i > 0:
        j = maxi(l, i + 1)[1]
        if j != i:
            l[j], l[i] = l[i], l[j]
        i -= 1
    return l
```

Tri par insertion

Le *tri par insertion* est généralement le tri que l'on utilise pour classer des documents : on commence par prendre le premier élément à trier que l'on place en position 1. Puis on insère les éléments dans l'ordre en plaçant chaque nouvel élément à sa bonne place.
Pour procéder à un tri par insertion, il suffit de parcourir une liste : on prend les éléments dans l'ordre. Ensuite, on les compare avec les éléments précédents jusqu'à trouver la place de l'élément qu'on considère. Il ne reste plus qu'à décaler les éléments du tableau pour insérer l'élément considéré à sa place dans la partie déjà triée.
Par exemple, si on veut trier la liste `[12, 3, 17, 9, 4, 2, 16]`, on obtient successivement :

```
[12,    3, 17, 9, 4, 2, 16]
[3, 12,    17, 9, 4, 2, 16]
[3, 12, 17,    9, 4, 2, 16]
[3, 9, 12, 17,    4, 2, 16]
[3, 4, 9, 12, 17,    2, 16]
[2, 3, 4, 9, 12, 17,    16]
[2, 3, 4, 9, 12, 16, 17   ]
```

a) Écrire une fonction `insertion(l, n)` qui prend en argument une liste `l` dont on suppose que les `n` premiers éléments sont triés et qui insère l'élément `l[n]` à sa place parmi les `n` premiers éléments de `l`.

b) Écrire une fonction `tri_insert(l)` qui effectue un tri par insertion de la liste `l`.

Solution.

tri.py

```
def insertion(l, n):
    while l[n] < l[n-1] and n > 0:
        l[n-1], l[n] = l[n], l[n-1]
        n -= 1
    return l
```

tri.py

```
def tri_insert(l):
    n = len(l)
    for i in range(1, n):
        l = insertion(l, i)
    return l
```

Tri rapide

Le *tri rapide* (ou « *quicksort* ») est une des méthodes de tri les plus rapides. L'idée est de prendre un élément au hasard (par exemple le premier) que l'on appelle *pivot* et de le mettre à sa place définitive en plaçant tous les éléments qui sont plus petits à sa gauche et tous ceux qui sont plus grands à sa droite. On recommence ensuite le tri sur les deux sous-listes obtenues jusqu'à ce que la liste soit triée. Par exemple :

- Pour la liste `[10,1,5,19,3,3,2,17]` le premier pivot choisi est 10.
- On place les éléments plus petits que 10 à gauche et les plus grands à droite :
  `[1,5,3,3,2, 10, 19,17]`
- Il reste deux sous-listes à trier `[1,5,3,3,2]` et `[19,17]`.

Écrire un algorithme *récursif* `tri_rapide(l)` triant une liste en utilisant l'algorithme.

Solution.

tri.py

```
def tri_rapide(l):
    if l == []: return []
    return (tri_rapide([x for x in l[1:] if x < l[0]])
            + [l[0]] + tri_rapide([x for x in l[1:] if x >= l[0]]))
```

14 Quelques primitives usuelles

Dans cette section, on présente les primitives (ou « *built-in functions* » en anglais) les plus usuelles du langage. Nous avons déjà rencontré les primitives `id`, `type`, `len`, `eval`, `exec`, `input`, `print`, `format`, `range`, etc. La liste complète et la description des primitives fournies par **Python** se trouve à la page : `http://docs.python.org/py3k/library/functions.html`
La primitive la plus utile pour connaître les autres est la fonction `help`, qui permet d'accéder dans l'interpréteur à la documentation d'une fonction :

```
>>> help(divmod)
Help on built-in function divmod in module builtins:
divmod(...)
    divmod(x, y) -> (div, mod)
    Return the tuple ((x-x%y)/y, x%y). Invariant: div*y + mod == x.
```

Les touches fléchées du clavier et les commandes de l'éditeur de texte `less` permettent de naviguer dans la page d'aide :

`j` : descendre `k` : monter `q` : quitter `h` : aide sur l'aide

Après importation d'un module, on peut également accéder à la documentation correspondante :

```
>>> import math
>>> help(math)
>>> help(math.hypot)
```

14.1 Quelques primitives d'un usage courant

```
>>> divmod(47, 5)      # quotient, reste            # rép.: (9, 2)
>>> abs(-1.3)          # valeur absolue d'un nombre # rép.: 1.3
>>> round(-1.3)        # arrondi d'un flottant      # rép.: -1
>>> pow(10, -5)        # exponentiation             # rép.: 1e-05
>>> max([3, 8, 9, -1]) # maximum d'une séquence     # rép.: 9
>>> min("alphabet")    # minimum d'une séquence     # rép.: a
>>> sum(range(100))    # somme d'une séquence       # rép.: 4950
```

14.2 Primitives de conversion de type

Les primitives suivantes permettent de modifier le type d'un objet :

`dict`, `bin`, `int`, `str`, `bool`, `ord`, `float`, `complex`, `set`, `tuple`, `chr`, `list`.

De plus, la primitive `isinstance` permet de tester l'appartenance d'un objet à un type (c'est-à-dire à une classe).

Exercice 10.

En utilisant la primitive `str`, écrire une fonction qui renvoie le nombre de chiffres d'un nombre entier naturel dans son écriture décimale.

Combien l'entier 2^{1000}, écrit en base 10, possède-t-il de chiffres ?

Solution.

On peut utilise la primitive `str` pour convertir un entier en chaîne de caractères, puis en calculer la longueur.

```
def nbre_chiffres(n):
    return len(str(n))

print(nbre_chiffres(2**1000))
```

Exercice 11.
En utilisant la primitive `int`, écrire une fonction qui renvoie le la partie entière $\lfloor x \rfloor$ d'un nombre décimal x. On rappelle que $\lfloor -3.1 \rfloor = -4$ (et non pas -3).

Solution.

```
def E(x):
    return int(x) if x>=0 else int(x)-1

print(E(3.1), E(0), E(-3.1))
```

```
3 0 -4
```

14.3 Itérateurs

Nous avons déjà rencontré un *itérateur* lorsque nous avons abordé la fonction `range`. Les générateurs sont des objets qui permettent de parcourir une séquence sans avoir à stocker la séquence complète en mémoire. Le principe est équivalent à un curseur de données placé sur la première donnée et qui découvre les éléments au fur et à mesure de l'avancée dans la séquence.
Noter que l'on peut convertir facilement un itérateur en liste à l'aide de la fonction `list` :

```
>>> list(range(2, 15, 3))
```

La fonction `enumerate` crée à partir d'une séquence un itérateur qui renvoie à chaque étape le t-uplet formé de l'élément de la séquence précédé de son indice de position :

```
L = [1, 5, 3, 'a']
for i in range(len(L)):   print("L[{0}]={1}".format(i,L[i]))
# Variante plus pythonique :
for i, x in enumerate(L): print("L[{0}]={1}".format(i, x))
```

Cette primitive nous permet, par exemple, d'écrire simplement une fonction qui renvoie la liste des positions où apparaît un élément dans une liste :

```
def occurences(x, liste):
    return [i for i, y in enumerate(liste) if x == y]
```

La fonction `reversed` crée un itérateur inversé :

```
for i in reversed(range(10)):
    print(i)
```

La fonction `reversed` permet par exemple de programmer succinctement l'algorithme de HORNER (cf. 12) de manière itérative :

horner.py

```
def horner(p, x):
    somme = 0
    for c in reversed(p):
        somme = c + x*somme
    return somme
```

Lorsque l'on souhaite écrire une boucle `for` en parcourant les couples formés par les éléments de deux (ou plusieurs) séquences, on utilise la fonction `zip` :

```
>>> list(zip([1, 2, 3, 4, 5], ['a', 'b', 'c']))
[(1, 'a'), (2, 'b'), (3, 'c')]
```

Noter que dans le cas où les deux séquences n'ont pas la même taille, la fonction `zip` utilise une version tronquée de la plus longue liste.
Lorsqu'on représente des vecteurs par des listes (ici u et v), il est alors très simple de les additionner composante par composante :

```
>>> [x[0] + x[0] for x in zip(u, v)] # addition vectorielle
>>> [x + y for x, y in zip(u, v)]    # addition vectorielle
```

On peut même transposer une matrice très simplement en la dépaquetant grâce à l'opérateur $*$:

```
>>> matrice = [[1, 2, 3], [6, 5, 4]]
>>> print(list(zip(*matrice)))  # transposition matricielle
[(1, 6), (2, 5), (3, 4)]
```

Il existe encore d'autre moyens de créer des itérateurs : soit en utilisant la primitive `iter` ou des fonctions du module `itertool` (que nous ne détaillerons pas ici), soit en utilisant un *générateur d'expression* (en anglais `generator expression`). Les générateurs d'expression sont aux itérateurs ce que les listes par compréhensions sont aux listes.

```
>>> (i**2 for i in range(10))   # générateur d'expression
```

Par exemple, on peut programmer la méthode naïve d'évaluation d'un polynôme avec un générateur d'expression :

```
def evaluation_naïve(p, x):
    return sum(c * x**i for i, c in enumerate(p))
```

Les générateurs d'expression nous fournissent un moyen très efficace de calculer le produit scalaire de deux vecteurs représentés par des listes :

```
>>> sum(x * y for x, y in zip(u, v))  # produit scalaire de u et v
```

Enfin signalons les primitives `all` et `any` qui prennent en argument un objet itérable : la première renvoie `True` si tous les éléments de l'itérable sont vrais et la seconde renvoie `True` si au moins un des éléments est vrai. Voici une fonction qui teste si les éléments d'une liste sont triés dans l'ordre croissant :

```
def est_triee(liste):
    return all(x <= y for x, y in zip(liste, liste[1:]))
```

15 Un mot sur les exceptions

Lorsqu'une erreur se produit dans le déroulement d'un programme (par exemple une division par 0), l'interpréteur stoppe alors l'exécution du programme et affiche la nature de l'erreur. Les types d'erreurs sont classés suivant une certaine hiérarchie consultable ici : http://docs.python.org/py3k/library/exceptions.html

Il est possible de produire une erreur personnalisée dans un programme grâce à l'instruction `raise` suivie de l'une des exceptions intégrées à **Python**.
Supposons par exemple que l'on souhaite écrire une fonction qui renvoie le produit scalaire de deux vecteurs si les deux vecteurs ont même taille, et lève une exception dans le cas contraire :

dotprod.py

```
def dotprod(u, v):
    if len(u) != len(v):
        raise IndexError("Les vecteurs n'ont pas la même taille !")
    return sum(x*y for x, y in zip(u, v))
print(dotprod([3, 9], [5, -1])); print(dotprod([3, 9, 0], [5, -1]))
```

```
6
Traceback (most recent call last):
  File "<stdin>", line 7, in <module>
  File "<stdin>", line 3, in dotprod
IndexError: Les vecteurs n'ont pas la même taille !
```

Lorsqu'une exception est interceptée, le programme est interrompu et l'interpréteur remonte en sens inverse toutes les couches du code précédemment traversées. Pour stopper cette remontée, on peut utiliser la directive `try...except`. En effet, si au cours de l'exécution de la clause `try` une erreur est interceptée, alors l'interpréteur passe à l'exécution de(s) clause(s) `except` avant de continuer le programme normalement. Plus précisément, l'interpréteur cherche à savoir si l'exception interceptée correspond à l'une des erreurs listées dans les clauses `except`.

```
try:
    x = a + 1/a
except ZeroDivisionError:
    print("Division par zéro !")
except NameError:
    print("La variable 'a' n'est pas définie !")
else:
    print(x)
```

Il est possible d'utiliser le mot-clé `except` sans préciser le type d'exception à intercepter, mais cette pratique est à déconseiller absolument. En effet, il se pourrait alors qu'on masque ainsi une exception qui ne corresponde pas à celle que l'on attendait, ce qui pourrait entraîner une erreur inextricable.
Signalons que le bloc `try` supporte l'utilisation d'un bloc `else` après le dernier bloc `except` et d'un bloc `finally` :

- le bloc `else` est exécuté si aucune exception n'a été interceptée dans le bloc `try` ;
- le bloc `finally` est exécuté dans tous les cas.

16 Compléments sur les fonctions

Pour définir la liste des paramètres formels d'une fonction, **Python** offre une syntaxe très riche qui permet d'économiser souvent l'inclusion d'instructions conditionnelles au sein du

programme.
Tout d'abord, on peut préciser des *valeurs par défaut* pour les paramètres formels :

```
def zeta(p, N=100):
    return sum(1/i**p for i in range(1, N))
print(zeta(2))          # p, N = 2, 100
print(zeta(2, 10**5))   # p, N = 2, 100000
print(zeta(3))          # p, N = 3, 100
```

Il est également possible d'utiliser des *paramètres implicites*, ce qui permet à l'utilisateur de fournir autant de paramètres qu'il souhaite. L'interpréteur place alors les valeurs des paramètres dans un dictionnaire :

```
def affiche(**arguments):
    for clé, valeur in arguments.items():
        print("arguments[{}] = {}".format(clé, valeur), end=' ; ')
    print('\narguments =', arguments)
affiche(arg1 = 'mot1', arg2 = '10', x = 2)
```

```
arguments[arg1] = mot1 ; arguments[arg2] = 10 ; arguments[x] = 2 ;
arguments = {'arg1': 'mot1', 'arg2': '10', 'x': 2}
```

Enfin, il est possible d'utiliser des *paramètres arbitraires*. Ces paramètres sont semblables aux paramètres implicites ; la différence tient au fait qu'ils ne sont pas nommés. L'interpréteur les place dans un t-uplet.

```
def produit(*facteurs):
    p = 1
    for i in facteurs: p *= i
    print("Le produit des facteurs {} est {}.".format(facteurs, p))
produit(2, 3, 4, 5)
```

```
Le produit des facteurs (2, 3, 4, 5) est 120.
```

Lorsqu'on combine des paramètres avec des valeurs par défaut, des paramètres implicites et arbitraires, on doit respecter scrupuleusement l'ordre suivant :

```
def fonction(x, y, z, *args, **keywords, a=1, b=2, c=3)
```

Signalons qu'il est possible de définir des fonctions anonymes à l'aide de la directive `lambda`. Voici par exemple une fonction `composition` qui prend en argument deux fonctions f et g et qui renvoie la *fonction* composée $f \circ g$:

```
def composition(f, g):
    return lambda x: f(g(x))
```

Terminons cette section en mentionnant que l'on peut munir une fonction d'une documentation appelée en anglais « *docstring* » : elle doit être placée à la ligne qui suit le mot-clé `def`, et être saisie entre triples guillemets anglais. La première phrase se doit d'expliquer de manière concise à quoi sert la fonction. Pour une fonction un peu développée, on peut même inclure après une ligne blanche des exemples d'utilisation qui peuvent ensuite servir de tests unitaires [18].

18. Un test unitaire est un procédé permettant de s'assurer du fonctionnement correct d'une portion d'un programme. On écrit un test pour confronter une réalisation à sa spécification.

composition.py

```
def composition(f, g):
    """Renvoie la composée de deux fonctions f et g.

    >>> from math import sin, cos
    >>> f = composition(cos, sin)
    >>> g = composition(sin, cos)
    >>> [f(x) - g(x) for x in range(0, 3)]
    [0.1585290151921035, 0.15197148686933137, 1.018539435968748]
    """
    return lambda x: f(g(x))

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Pour accéder à la documentation de la fonction, on utilise `help` :

```
>>> help(composition)
```

Pour lancer les tests, on exécute le script avec l'option `-v` (mode « bavard ») :

```
$ python3 mafonction.py -v
```

17 Notions sur les classes

En **Python**, tous les objets manipulés sont des objets au sens de la *programmation orientée objet*, c'est-à-dire des entités regroupant :

- des données, appelés *attributs* ou *variables d'instance* de l'objet ;
- des fonctions, appelées *méthodes* de l'objet.

Par exemple, les entiers, les flottants, les listes, les t-uplets, les chaînes de caractères sont des objets. Parmi les attributs d'un objet de la classe liste, on peut citer ses éléments ; parmi ses méthodes, on peut citer les fonctions `append()`, `extend()`, etc.
Seules les méthodes sont habilitées à manipuler les données d'un objet, ce que l'on traduit en disant que les données de l'objet sont *encapsulées* dans l'objet.
Pour connaître la liste des attributs et méthodes d'un objet, on invoque la primitive `dir` avec comme argument le nom de l'objet.
Une *classe* est une description donnant naissance à différents objets disposant tous de la même structure de données et des mêmes méthodes. En un sens, on pourrait dire que les mots « classe » et « type » sont synonymes.
À côté des nombreuses classes définies par défaut par **Python** (par exemple `int`, `float`, `list`, `tuple`, `str`,...), il est possible de définir de nouvelles classes. Dans cette section, nous présentons quelques ébauches de classes pour programmer quelques objets mathématiques. Nous verrons dans les chapitres suivants des exemples plus développés de définition de classes. Enfin le chapitre 6 donnera un exemple de projet abouti utilisant de manière intensive les principes de la programmation orientée objet.

Nous allons partir d'un exemple concret, à savoir la notion de polynôme. La manière la plus naturelle de représenter un polynôme $p = a_0 + a_1 X + \cdots + a_n X^n$ est d'utiliser une liste contenant ses coefficients $[a_0, a_1, \ldots, a_n]$. Si l'on dispose alors de deux listes `p1` et `p2` représentant deux polynômes, on souhaiterait pouvoir les additionner en utilisant l'opérateur +. Mais en l'état, la liste renvoyée serait le résultat de la concaténation des deux listes. De plus, on souhaiterait pouvoir afficher le polynôme de manière conventionnelle. C'est ici que la définition d'une classe nous permettra de définir de nouvelles fonctions (les méthodes) propres aux polynômes.
Pour créer une classe, on utilise le mot-clé `class` :

```
class Polynome(object):
```

Ensuite, il faut définir le *constructeur* de la classe, c'est-à-dire la méthode qui nous permettra de créer (ou d'*instancier*) un objet de la classe polynôme. Le constructeur a toujours pour nom `__init__()` et prend en argument au moins un paramètre appelé `self`. Ce paramètre `self` fera toujours référence à l'objet instancié (à savoir ici un polynôme).

Polynomes.py

```
class Polynome(object):
    def __init__(self, coefficients):
        self.coeffs = coefficients
```

Pour créer un objet de la classe polynôme, il nous suffira alors d'utiliser la syntaxe suivante :

```
>>> p = Polynome([3, -2, 1])  # représente le polynôme 3-2.X+X^2
```

On va munir à présent les objets `Polynome` d'une méthode renvoyant le degré du polynôme :

Polynomes.py

```
    def deg(self):
        n = len(self.coeffs)
        for i, c in enumerate(reversed(self.coeffs)):
            if c != 0:
                return n-1-i
        return -1
```

Pour appliquer cette méthode à un polynôme, on préfixera le nom de l'objet au nom de la méthode :

```
>>> p.deg()
```

On souhaiterait à présent définir l'addition de deux polynômes. Une première façon de faire consisterait à définir une méthode `ajoute()`. Pour ajouter deux polynômes, on écrirait alors `p1.ajoute(p2)`. L'idéal serait de pouvoir écrire plus simplement `p1+p2`. Pour cela, on va redéfinir (ou plus exactement *surcharger*) la méthode `__add__` définie par défaut à chaque définition de classe (mais dont la définition par défaut ne rend pas cette méthode opérationnelle en général).

Polynomes.py

```
    def __add__(self, other):
        if self.deg() < other.deg():
            self, other = other, self
        tmp = other.coeffs + [0]*(self.deg() - other.deg())
        return Polynome([x+y for x, y in zip(self.coeffs, tmp)])
```

Ainsi, les deux syntaxes suivantes seront possibles :

```
>>> p1 + p2    # équivaut à  p1.__add__(p2)
<Polynomes.Polynome object at 0xb7261e6c>
```

Le résultat semble décevant et ce, même si on utilise la syntaxe `print(p1+p2)`. En fait, au moment de la définition de la classe [19], chaque objet possède par défaut non seulement une méthode `__add__()`, mais aussi une méthode `__str__()` qui est à surcharger par la suite pour personnaliser le type d'affichage d'un objet de la classe, ce que nous pourrions faire ainsi :

Polynomes.py

```
    def str_monome(self, i, c):
        coeffs = '{}'.format(c) if c >= 0 else '({})'.format(c)
        indet = ('.X^{}'.format(i) if i > 1
                 else ('.X' if i == 1 else ''))
        return ''.join([coeffs, indet])

    def __str__(self):
        chaine = ' + '.join(self.str_monome(i, c)
                 for i, c in enumerate(self.coeffs) if c !=0)
        chaine = chaine.replace(' 1.', ' ')
        return chaine if chaine != '' else '0'
```

Pour améliorer l'affichage d'un polynôme, on a défini une fonction `str_monome()` qui affiche un monôme (par exemple, sous la forme `(-2).X^3`) en rajoutant des parenthèses autour des coefficients négatifs. Bien que cette fonction n'utilise pas le paramètre `self`, il est impératif que ce paramètre figure (et figure en premier) dans la liste des paramètres formels de la fonction `str_monome()`. C'est ce qui permettra d'appeler cette fonction dans une autre méthode de la classe, à savoir ici, la méthode `__str__()`.

```
>>> print(Polynome([3, -2, 1]))  # rép.: 3 + (-2).X + X^2
```

De même, on pourrait programmer l'évaluation d'un polynôme en un point (en suivant la méthode de HORNER) :

Polynomes.py

```
    def __call__(self, x):
        somme = 0
        for c in reversed(self.coeffs):
            somme = c + x*somme
        return somme
```

Pour l'évaluation, on utiliserait alors la syntaxe :

```
>>> p(1.2) # équivalente à p.__call__(1.2)
```

Le produit pourrait se programmer en définissant au préalable une fonction qui calcule le produit d'un polynôme par un monôme de la forme cX^i :

19. La liste des méthodes fournies par défaut au moment de la définition d'une classe se trouve ici : http://docs.python.org/py3k/reference/datamodel.html

Polynomes.py

```
    def mul_monome(self, i, c):
        return Polynome([0]*i + [c * x for x in self.coeffs])

    def __mul__(self, other):
        tmp = Polynome([0])
        for i, c in enumerate(other.coeffs):
            tmp += self.mul_monome(i, c)
        return tmp
```

L'encapsulation représente un intérêt certain de la notion de classe, mais ce n'est pas le seul. Le principal avantage de la notion de classe, est que l'on peut définir des classes dérivées d'une classe qui hériteront des méthodes de la classe parente. Ceci permet d'éviter de répéter des portions de code semblables. Pour définir une classe dérivée d'une classe parente, on utilise la syntaxe :

```
class Dérivée(Parente):
```

Considérons un exemple. Pour représenter une fraction rationnelle, on pourrait définir une classe à partir de rien et programmer des méthodes similaires à celles définies pour les polynômes. Mais il est bien plus avantageux de se servir des méthodes programmées dans la classe des polynômes pour les utiliser au sein de la classe des fractions rationnelles. Nous allons donc définir la classe des fractions rationnelles en la faisant dériver de la classe des polynômes.

Polynomes.py

```
class FracRationnelle(Polynome):
    def __init__(self, numerateur, denominateur):
        self.numer = numerateur
        self.denom = denominateur

    def deg(self):
        return self.numer.deg() - self.denom.deg()

    def __call__(self, x):
        return self.numer.__call__(x) / self.denom.__call__(x)

    def __str__(self):
        return ("({}) / ({})".format(self.numer, self.denom))

    def __add__(self, other):
        numer = self.numer * other.denom + self.denom * other.numer
        denom = self.denom * other.denom
        return FracRationnelle(numer, denom)

    def __mul__(self, other):
        numer = self.numer * other.numer
        denom = self.denom * other.denom
        return FracRationnelle(numer, denom)
```

Voici quelques exemples d'utilisation de cette classe :

```
p1, p2, p3 = Polynome([1]), Polynome([-1, 1]), Polynome([1, 1])
r1, r2 = FracRationnelle(p1, p2), FracRationnelle(p1, p3)
print(r1, r2, r1 + r2, r1 * r2, sep=' ; ')
print(r1(-1.3))
```

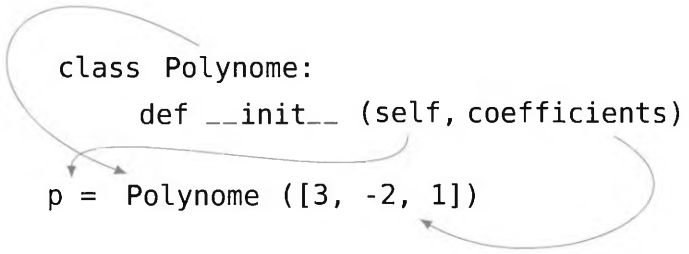
On pourrait encore améliorer la définition de notre classe de fractions rationnelles, en définissant une classe de rationnels ; puis en faisant hériter la classe des fractions rationnelles non seulement de la classe des polynômes, mais aussi de la classe des rationnels. On parle dans ce cas d'héritage multiple.

Polynomes.py

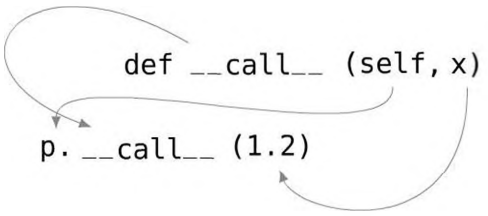
```
class Rationnel(object):
    def __init__(self, num, den):
        self.numer = num
        self.denom = den

    def __str__(self):
        return ("({}) / ({})".format(self.numer, self.denom))

    def __add__(self, other):
        denom = self.denom * other.denom
        numer = self.numer * other.denom + other.numer * self.denom
        return Rationnel(numer, denom)

    def __mul__(self, other):
        numer = self.numer * other.numer
        denom = self.denom * other.denom
        return Rationnel(numer, denom)

class FracRationnelle(Rationnel, Polynome):
    def __init__(self, numerateur, denominateur):
        self.numer = numerateur
        self.denom = denominateur

    def deg(self):
        return self.numer.deg() - self.denom.deg()

    def __call__(self, x):
        return self.numer.__call__(x) / self.denom.__call__(x)

    def __add__(self, other):
        tmp = (Rationnel(self.numer, self.denom)
               + Rationnel(other.numer, other.denom))
        return FracRationnelle(tmp.numer, tmp.denom)

    def __mul__(self, other):
        tmp = (Rationnel(self.numer, self.denom)
               * Rationnel(other.numer, other.denom))
        return FracRationnelle(tmp.numer, tmp.denom)
```

On remarque dans ce cas qu'il est inutile de redéfinir une méthode `__str__()` pour la classe `FracRationnelle`, et que l'écriture des méthodes `__add__()` et `__mul__()` se trouve simplifiée.
Ces programmations de classes de polynômes, de nombres rationnels et de fractions rationnelles ne sont que des esquisses et sont loin d'être achevées. Le but dans cette section était simplement d'illustrer les notions de classe, de méthode, de surcharge des opérateurs, d'héritage. Le chapitre 6 présentera des classes pleinement opérationnelles pour les polynômes, les rationnels et les fractions rationnelles. Avant cela, d'autres exemples de leur utilisation seront abordés. C'est au fil des exemples que leur manipulation deviendra de plus en plus naturelle.
En guise de conclusion, rassemblons quelques remarques au vu des méthodes programmées précédemment.

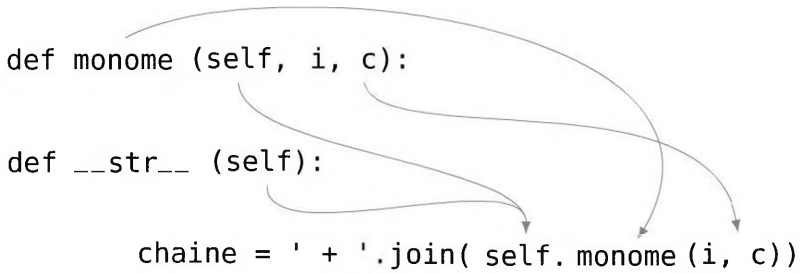
- Le paramètre `self` représente une instance quelconque (c'est-à-dire un objet) de la classe.



- Le paramètre `self` est toujours le premier paramètre formel d'une méthode (de la classe) au moment de sa définition. En revanche, il n'apparaît plus dans la liste des paramètres d'une méthode (de la classe) au moment de son appel. En effet, il est alors préfixé au nom de la méthode appelée.



- Pour accéder à une autre méthode ou à un attribut à l'intérieur d'une méthode de la classe, il faut toujours préfixer le nom du paramètre `self` au nom de la méthode.



18 Exercices d'entraînement

Les corrigés sont disponibles sur le site dunod.com à partir de la page d'accueil de l'ouvrage.

Exercice 12.
Donner au moins trois façons d'écrire une fonction permettant de tester si une chaîne de caractères est un palindrome.

Exercice 13.
La constante de Champernowne est un nombre réel de partie entière nulle et dont le développement décimal est obtenu en écrivant les uns à la suite des autres tous les entiers naturels :

$$0.1\,2\,3\,4\,5\,6\,7\,8\,9\,10\,11\,12\,13\,14\,15\,16\,17\,18\,19\,20\,21\,22\,23\,24\,25\,26\,27\,28\,29\,30\ldots$$

Il a été démontré que ce nombre est transcendant, normal et universel.
Écrire une procédure qui renvoie les n premières décimales de ce nombre.

Exercice 14.

a) On note C_n le nombre de chiffres qui interviennent dans l'écriture décimale de l'entier n. Afficher les 8 premières valeurs de la somme partielle de la série $\sum_{n\geq 1} \frac{C_n}{n(n+1)}$.

b) Si $n \in \mathbb{N}^*$ ne contient pas de 9 dans son écriture en base 10, on pose $u_n = \frac{1}{n^2}$; sinon on pose $u_n = 0$. Donner une fonction qui approche les sommes partielles de la série $\sum_{n\geq 1} u_n$.

Solution.
a) On peut par exemple utiliser la primitive `str` pour convertir un entier en chaîne de caractères, puis en calculer la longueur.

```
def s(n):
    return sum(len(str(k))/(k*(k+1)) for k in range(1, n))

for i in range(1, 8):
    print(s(10**i), end=', ')
```

```
0.9, 1.08, 1.107, 1.1106, 1.11105, 1.111104, 1.1111103,
```

On conjecture donc que la série converge et que sa somme vaut $\frac{10}{9}$, ce qui peut se démontrer par une sommation par paquets.
b) Même principe :

```
def s(n):
    return sum(1/k**2 for k in range(1, n) if '9' not in str(k))
```

On conjecture donc que la série converge et que sa somme est proche de 1.6239249

Exercice 15. (Approximation de la fonction cosinus)
En utilisant l'inégalité de Taylor-Lagrange appliquée à la fonction cosinus, à l'ordre $2n$, on peut montrer que

$$\forall x \in \mathbb{R} \quad \forall n \in \mathbb{N} \qquad \left| \cos x - \sum_{k=0}^{n} (-1)^k \frac{x^{2k}}{(2k)!} \right| \leq \frac{|x|^{2n+1}}{(2n+1)!}$$

Écrire une procédure `cosinus(x, epsilon)` qui prend en argument un nombre `x` et une précision `epsilon`, et qui renvoie la valeur approchée du cosinus de `x` à `epsilon` près.
On remarquera qu'il est inutile d'utiliser une fonction factorielle.
À titre de vérification, voici ce que doit donner le calcul de cos(1) :

```
print(cosinus(1, 1e-14))                # rép.: 0.5403023058681398
```

Exercice 16. (Triplets pythagoriciens)
On appelle *triplet pythagoricien* un triplet d'entiers naturels (x, y, z) non nuls vérifiant la relation $x^2 + y^2 = z^2$.

a) Écrire une fonction qui prend en argument un entier n et qui renvoie la liste (sans doublon) des tous les triplets pythagoriciens dont les composantes sont strictement inférieures à n.

b) Combien y a-t-il de triplets pythagoriciens jusqu'à 100 ?

Exercice 17.
Chercher dans la documentation de **Python** l'utilité de la fonction `join`. À l'aide de cette fonction, créer des chaînes de caractères de longueur arbitraire. Comment afficher les égalités suivantes portant sur la somme des carrés des premiers entiers ?

```
1^2 = 1
1^2 +  2^2 = 5
1^2 +  2^2 +  3^2 = 14
1^2 +  2^2 +  3^2 +  4^2 = 30
1^2 +  2^2 +  3^2 +  4^2 +  5^2 = 55
1^2 +  2^2 +  3^2 +  4^2 +  5^2 +  6^2 = 91
1^2 +  2^2 +  3^2 +  4^2 +  5^2 +  6^2 +  7^2 = 140
1^2 +  2^2 +  3^2 +  4^2 +  5^2 +  6^2 +  7^2 +  8^2 = 204
1^2 +  2^2 +  3^2 +  4^2 +  5^2 +  6^2 +  7^2 +  8^2 +  9^2 = 285
1^2 +  2^2 +  3^2 +  4^2 +  5^2 +  6^2 +  7^2 +  8^2 +  9^2 + 10^2 = 385
1^2 +  2^2 +  3^2 +  4^2 +  5^2 +  6^2 +  7^2 +  8^2 +  9^2 + 10^2 + 11^2 = 506
```

Exercice 18. (Nombres parfaits)
Un nombre est dit *parfait* lorsque il est égal à la somme de ses diviseurs propres. Un diviseur propre est un diviseur autre que le nombre lui-même. Le premier nombre parfait est 6. En effet 1, 2 et 3 sont les diviseurs propres de 6 et $6 = 1 + 2 + 3$.

a) Écrire une fonction booléenne `parfait(n)` qui teste si un entier est parfait.

b) Écrire une fonction `liste_parfaits(n)` qui renvoie la liste des nombres parfaits strictement inférieurs `n`. Quels sont les nombres parfaits inférieurs à 10 000 ?

c) Écrire une fonction `somme(n)` qui, si `n` est un nombre parfait, affiche l'égalité justifiant cette propriété. Par exemple, `somme(6)` devra afficher `6 = 1 + 2 + 3`.

Exercice 19. (Crible d'Ératosthène)
Écrire une fonction récursive `crible(liste)` qui prend en argumant une liste et y élimine récursivement tous les multiples de ses éléments en parcourant la liste de gauche à droite.
Appeler la fonction `crible(range(2,n))` pour obtenir alors la liste des nombres premiers strictement inférieurs à `n`.

2 Modules

Sommaire

1 Structure d'un module

En programmation, il est fréquent de sauvegarder dans un fichier du code pour pouvoir le réutiliser dans des contextes variés. Un tel fichier est appelé un *module*. On donne généralement à son nom l'extension « .py ».
Supposons que l'on écrive une fonction pour afficher une liste assez longue en alignant les éléments de la liste les uns au-dessus des autres [1]. Une fois la fonction écrite, on la place dans un fichier comme ci-dessous :

Lprint.py

```
"""
   Module pour afficher une liste longue.
"""
def lprint(liste, largeur, forme):
    n = len(liste)
```

1. Noter qu'une fonction semblable est fournie avec le module `pprint` de la librairie standard : http://docs.python.org/py3k/library/pprint.html

```
    lignes = int(n / largeur)
    print('[', end='')
    for i in range(n):
        s = '{:' + forme + '}'
        if i != n-1:
            print(s.format(liste[i]), end=',')
        else:
            print(s.format(liste[i]), end=' ]')
        if i != 0 and i % largeur == largeur - 1:
            print('\n ', end='')
    print()
```

La directive `import` permet alors de charger le contenu du module de la manière suivante :

```
>>> from Lprint import * # importe TOUS les éléments d'un module
>>> liste = list(range(30))
>>> lprint(liste, 12, '3d')
[  0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
  12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
  24, 25, 26, 27, 28, 29 ]
```

Au moment de la première importation du module `Lprint.py`, l'interpréteur génère un fichier `Lprint.pyc` qui optimise ensuite sa mise en œuvre.
Si l'on ne souhaite pas encombrer l'espace des noms en important un module, il est recommandé de l'importer sans utiliser le mot-clé `from` : dans ce cas, pour appeler une fonction du module importé, il suffira de faire précéder le nom de la fonction du nom du module en accolant les deux par un point.

```
>>> import Lprint
>>> liste = list(range(50))
>>> Lprint.lprint(liste, 12, '3d')
```

Dans ce cas, l'utilisation d'un alias plus court à taper peut se révéler avantageuse :

```
>>> import Lprint as lp
>>> liste = list(range(50))
>>> lp.lprint(liste, 12, '3d')
```

Pour finir, mentionnons deux astuces concernant l'écriture de modules. Lorsqu'on met au point un module, il est intéressant d'y inclure des tests pour en vérifier son bon fonctionnement. L'embarras, c'est qu'au moment de l'importation du module, ces tests seront également effectués, ce qui n'est pas souhaitable en général. Il suffit alors de faire précéder les tests de l'instruction `if __name__ == "__main__":`. Par exemple dans le module précédent, on pourrait ajouter en fin de module les lignes suivantes :

Lprint.py

```
if __name__ == "__main__":
    l = list(range(100))
    print(l)
    lprint(l, 12, '3d')
```

Un problème se présente lorsqu'on a écrit un module que l'on souhaite utiliser dans des répertoires autres sans avoir à le copier en plusieurs endroits. En effet, si l'on importe un module

personnel qui n'est pas dans le répertoire où se trouve le fichier à exécuter, un message d'erreur est renvoyé disant que ce module est inconnu. Dans ce cas, le plus simple est de modifier la variable `sys.path` qui contient la liste des chemins où l'interpréteur recherche les modules, en écrivant au début du fichier dans lequel on souhaite importer le module :

```
import sys
sys.path.append('/home/user/programmation/python/mes_modules/')
```

2 Quelques modules « Batteries included »

Le langage **Python** offre par défaut une bibliothèque de plus de deux cents modules [2] qui évite d'avoir à réinventer la roue dès que l'on souhaite écrire un programme. Ces modules couvrent des domaines très divers : mathématiques (fonctions mathématiques usuelles, calculs sur les réels), administration système, programmation réseau, manipulation de fichiers, etc. Nous en présenterons ici seulement quelques-uns, à savoir ceux dont nous nous servirons dans la suite de notre ouvrage.

2.1 Le module `math`

Le module `math` permet d'importer les fonctions et constantes mathématiques usuelles.

commande **Python**	constante / fonction mathématique
`pi, e, exp(x), log(x), log(x,a)`	π, e= exp(1), $\exp(x)$, $\ln(x)$, $\log_a(x)$
`pow(x,y), floor(x), abs(x), factorial(n)`	x^y, $\lfloor x \rfloor$, $\lvert x \rvert$, $n!$
`sin(x), cos(x), tan(x), asin(x),...`	fonctions trigonométriques
`sinh(x), cosh(x), tanh(x), asinh(x),...`	fonctions hyperboliques

Comme mentionné précédemment, on dispose de plusieurs syntaxes pour importer un module :

```
>>> from math import *
>>> cos(pi / 4.0)
0.70710678118654757
```

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
```

2.2 Le module `random`

Le module `random` propose diverses fonctions permettant de générer des nombres (pseudo-)aléatoires qui suivent différentes distributions mathématiques.
Il apparaît assez difficile d'écrire un algorithme qui soit réellement non-déterministe (c'est-à-dire qui produise un résultat totalement imprévisible). Il existe cependant des techniques mathématiques permettant de simuler plus ou moins bien l'effet du hasard.

Voici quelques fonctions fournies par ce module :

2. La liste complète des modules de la bibliothèque standard est accessible à l'adresse : http://docs.python.org/py3k/library/index.html. À cela s'ajoute les 16627 modules développés (à ce jour) par la communauté **Python** : http://pypi.python.org/pypi

`random.randrange(p, n, h)`	choisit un entier aléatoirement dans `range(p,n,h)`
`random.randint(a, b)`	choisit un entier aléatoirement dans $[\![a,b]\!]$
`random.choice(seq)`	choisit un entier aléatoirement dans la séquence `seq`
`random.random()`	renvoie un décimal aléatoire dans $[0,1[$
`random.uniform(a, b)`	choisit un décimal aléatoirement dans $[a,b]$

Exercice 20. Les méthodes de simulation de MONTE-CARLO permettent de calculer une valeur numérique en utilisant des procédés aléatoires. Elles sont particulièrement utilisées pour calculer des intégrales en dimensions plus grandes que 1 (en particulier, pour le calcul de surfaces et de volumes).
Par exemple, le calcul approché d'une intégrale d'une fonction à 1 variable par cette méthode repose sur l'approximation :

$$\int_a^b f(t)\,\mathrm{d}t \approx \frac{b-a}{n}\sum_{i=1}^{n} f(x_i)$$

où $i \mapsto x_i$ désigne une variable aléatoire suivant la loi uniforme sur l'intervalle $[a,b]$.
Écrire une fonction renvoyant une approximation du nombre

$$\pi = 4\int_0^1 \sqrt{1-t^2}\,\mathrm{d}t$$

en utilisant la méthode de MONTE-CARLO.

Solution.

montecarlo.py

```
import random, math

def montecarlo(f, a, b, n):
    somme = 0
    for i in range(n):
        x = random.uniform(a, b)
        somme += f(x)
    return somme * (b-a) / n

def f(x):
    return math.sqrt(1 - x**2)

print('-'*63)
print('{0:>10s} | {1:^12s} | {2:^14s} | {3:^15s} |'.format('n',
        'approximation', 'erreur absolue', 'erreur relative'))
print('-'*63)
for i in range(0, 10, 2):
    n = 10**i
    approx = 4*montecarlo(f, 0, 1, n)
    erreur = math.pi - approx
    print('{0:10d} | {1: 12.10f} | {2: 14.10f} | {3: 15.10f} |'
            .format(n, approx, erreur, abs(erreur / math.pi)))
print('-'*63)
```

```
-------------------------------------------------------------
         n | approximation | erreur absolue | erreur relative |
-------------------------------------------------------------
         1 |  3.2073235561 |  -0.0657309025 |    0.0209227961 |
       100 |  3.1653208532 |  -0.0237281996 |    0.0075529205 |
     10000 |  3.1463012568 |  -0.0047086032 |    0.0014987949 |
   1000000 |  3.1414691559 |   0.0001234977 |    0.0000393105 |
 100000000 |  3.1418489337 |  -0.0002562801 |    0.0000815765 |
-------------------------------------------------------------
```

Cette méthode de calcul approché d'intégrale se révèle assez peu efficace (et pas seulement sur cet exemple) en dimension 1 ; il faut un grand nombre de points pour obtenir quelques décimales significatives. Il est possible de prouver qu'avec la méthode d'approximation de Monte Carlo, l'erreur est dominée par $\frac{1}{n^{1/2}}$, alors que dans la méthode des trapèzes l'erreur est dominée par $\frac{1}{n^2}$. On comparera les résultats obtenus avec ceux obtenus à la section 3.
Pour une version vectorisée de l'algorithme de Monte Carlo, on pourra consulter [Lan09].

Exercice 21. Comme nous l'avons dit, la méthode de MONTE-CARLO peut également être utilisée pour le calcul numérique d'intégrales multiples. Prenons le cas d'une intégrale double

$$\mathrm{I} = \iint_{\mathrm{D}} f(x,y)\,\mathrm{d}x\,\mathrm{d}y$$

Pour estimer I, on commence par choisir un rectangle R qui contient D. On a alors

$$\mathrm{I} = \iint_{\mathrm{D}} f(x,y)\,\mathrm{d}x\,\mathrm{d}y = \iint_{\mathrm{R}} f(x,y)\chi_{\mathrm{D}}(x,y)\,\mathrm{d}x\,\mathrm{d}y$$

où χ_{D} est la fonction caractéristique de D.
On génère ensuite une suite de points (x_i, y_i) aléatoires suivant une loi uniforme dans le rectangle D. L'intégrale I est alors approchée par la quantité

$$\mathrm{I} \approx \frac{\mathcal{A}(\mathrm{R})}{n^2} \sum_{i=1}^{n} f(x_i,y_i)\chi_{\mathrm{D}}(x_i,y_i)$$

où $\mathcal{A}(\mathrm{R})$ désigne l'aire du rectangle R.
Soit $\mathrm{D} = \left\{(x,y) \in \mathbb{R}^2 \,\middle|\, \frac{x^2}{a^2} + \frac{y^2}{b^2} \leq 1\right\}$ et F, F′ les foyers de l'ellipse frontière.
Écrire une procédure permettant de calculer numériquement l'intégrale

$$\mathrm{I}(a,b) = \iint_{\mathrm{D}} (\mathrm{MF} + \mathrm{MF}')\,\mathrm{d}x\,\mathrm{d}y \quad \text{où} \quad \mathrm{M}(x,y) \in \mathrm{D}$$

Solution. Un calcul exact donne comme valeur $\mathrm{I}(a,b) = \frac{2\pi b}{3}\left(3a^2 - b^2\right)$, ce qui va nous permettre de tester la qualité de l'approximation. Pour les calculs, on utilise la fonction `hypot` du module `math` qui calcule la norme euclidienne d'un couple `(x, y)`.

montecarlo2.py

```
import random

from math import sqrt, hypot, pi

def f(x, y, a, b):
    c = sqrt(a**2-b**2)
    return hypot(x-c, y) + hypot(x+c, y)
```

```
def montecarlo2(a, b, n):
    xrandom = [random.uniform(-a, a) for i in range(n)]
    yrandom = [random.uniform(-b, b) for i in range(n)]

    somme = sum(f(x, y, a, b) for x in xrandom
            for y in yrandom if hypot(x/a, y/b) <= 1)

    return somme * (4 * a * b) / n**2

print('-'*63)
print('{0:>7s} | {1:^15s} | {2:^15s} | {3:^15s} |'.format('n',
        'approximation', 'erreur absolue', 'erreur relative'))
print('-'*63)

for i in range(1, 5):
    a, b, n = 2, 1, 10**i
    approx = montecarlo2(a, b, n)
    exacte = 2*pi*b*(3*a**2-b**2)/3
    erreur = exacte - approx
    print('{0:7d} | {1: 15.10f} | {2:^ 15.3e} | {3:^ 15.3e} |'
            .format(n, approx, erreur, abs(erreur / exacte)))
print('-'*63)
```

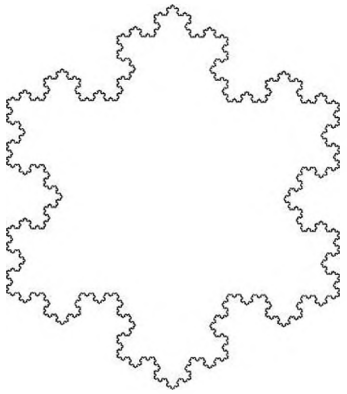
```
---------------------------------------------------------------
      n |  approximation  | erreur absolue  | erreur relative |
---------------------------------------------------------------
     10 |   21.2978747041 |    1.740e+00    |    7.555e-02    |
    100 |   22.6145697956 |    4.238e-01    |    1.839e-02    |
   1000 |   22.9467190111 |    9.163e-02    |    3.977e-03    |
  10000 |   23.2042742508 |   -1.659e-01    |    7.202e-03    |
---------------------------------------------------------------
```

2.3 Le module `turtle`

Le module `turtle` permet de réaliser des « graphiques tortue », c'est-à-dire des dessins géométriques correspondant à la piste laissée derrière elle par une petite « tortue » virtuelle, dont on contrôle les déplacements sur l'écran de l'ordinateur à l'aide d'instructions simples.
Les principales fonctions mises à votre disposition dans le module `turtle` sont les suivantes :

`reset()`	On efface tout et on recommence
`goto(x, y)`	Aller au point de coordonnées `(x, y)`
`forward(d), backward(d)`	Avancer, reculer d'une distance `d`
`up(), down()`	Relever, abaisser le crayon
`left(alpha), right(alpha)`	Tourner à gauche, à droite d'un angle `alpha` (en degrés)
`color(couleur), width(l)`	Détermine la couleur, l'épaisseur du tracé
`fill(1)`	Remplir un contour fermé

Exercice 22. (Flocon de VON KOCH)



L'algorithme de VON KOCH consiste, à partir d'un segment donné, à le diviser en 3 segments de même longueur et à remplacer le segment central par les 2 côtés d'un triangle équilatéral construit extérieurement à partir de ce segment central.
Le flocon de VON KOCH est obtenu en appliquant cet algorithme aux côtés d'un triangle équilatéral.

a) Écrire une fonction récursive `von_koch(l, n)` qui applique `n` fois l'algorithme de VON KOCH à un segment de longueur donnée ; cette fonction exécutera donc la suite d'instructions suivantes :

i) Tracer `von_koch(l/3, n-1)`
ii) Tourner à gauche de 60 degrés
iii) Tracer `von_koch(l/3, n-1)`
iv) Tourner à droite de 120 degrés
v) Tracer `von_koch(l/3, n-1)`
vi) Tourner à gauche de 60 degrés
vii) Tracer `von_koch(l/3, n-1)`

b) Écrire une fonction `flocon(l, n)` qui dessine un flocon de von Koch.

Solution.

vonKoch.py

```
import turtle
turtle.pen(speed = 0) # vitesse maximale

def von_koch(longueur, n):
    if n == 1:
        turtle.forward(longueur)
    else:
        d = longueur / 3.
        von_koch(d, n - 1); turtle.left(60)
        von_koch(d, n - 1); turtle.right(120)
        von_koch(d, n - 1); turtle.left(60)
        von_koch(d, n - 1)

def flocon(longueur, n):
    turtle.up()
    turtle.goto(- longueur / 2, longueur / 3) # on se place en haut, à gauche
    turtle.down()
    for i in range(3):
        von_koch(longueur, n); turtle.right(120)

flocon(300, 6)
```

```
# Pour exporter le graphique au format EPS :
from tkinter import *
import os
turtle.ht()
ts = turtle.getscreen()
ts.getcanvas().postscript(file="{0}.eps".format("von_koch"))
turtle.mainloop()
```

Exercice 23. (La machine de GALTON) La planche de GALTON réalise l'expérience aléatoire suivante. Des clous sont disposés en pyramide, de sorte qu'en lâchant une bille sur le clou supérieur, celle-ci tombe à gauche ou à droite, pour encore rebondir de la même façon sur le clou suivant, jusqu'à finir sa chute dans un réservoir. On laisse choir une collection de billes et on observe la répartition obtenue en fin d'expérience dans les différents réservoirs.

Le principe retenu pour la simulation est basé sur le traitement des listes. Si nous prenons une pyramide de base n clous, nous disposons de $n+1$ réservoirs pour recueillir toutes les chutes (et donc toutes les trajectoires) possibles. Notre réservoir est la liste des entiers de 0 à n. À chaque clou, une bille possède la même probabilité de partir à gauche ou à droite : si elle part à gauche, elle ne pourra plus atteindre le dernier réservoir à droite, qui est donc éliminé de la liste... En fin de chute, la liste est un singleton dont le numéro servira d'indice pour incrémenter le compteur du réservoir en question.

La machine de GALTON est un outil permettant de simuler une loi binomiale, et ainsi observer sa convergence vers une loi normale.

Solution.

galton.py

```
from random import *
from turtle import *

def chute(niveaux):
    c = ""
    for n in range(niveaux):
        if random() < 0.5:
            c = c + 'G'
        else:
            c = c + 'D'
    return c

def tirage(niveaux):
    cibles = [n for n in range(niveaux + 1)]
    c = chute(niveaux)
    for n in c:
        if n == 'G':
```

```
            cibles.pop()
        else:
            cibles.pop(0)
    return cibles[0]

def simulation(billes, niveaux):
    print("simulation de", billes, "sur un crible de", niveaux, "niveaux :")
    tas = [0] * (niveaux + 1)
    for b in range(billes):
        tas[tirage(niveaux)] += 1
        print(tas)
    print()

def rectangle(largeur, hauteur):
    couleur = (random(), random(), random())
    color(couleur)
    begin_fill()
    for n in range(2):
        forward(largeur); left(90)
        forward(hauteur); left(90)
    end_fill()

def diagramme(billes, niveaux):
    ### simulation ###
    tas = [0] * (niveaux + 1)
    for b in range(billes):
        tas[tirage(niveaux)] += 1
    print(tas)
    ### dessin ###
    largeur = 50
    hauteur_max = 0
    billes_max = 0
    for n in tas:
        if n > hauteur_max:
            hauteur_max = n
            billes_max = n
    ### centrage et mise à l'échelle ###
    up()
    goto( -largeur * (niveaux + 1)/2, -float(billes_max)/billes * hauteur_max/2)
    down()
    speed("fastest")

    for n in tas:
        rectangle(largeur, float(n)/billes * hauteur_max)
        forward(largeur)
    up()

if __name__ == "__main__":
    diagramme(5000, 10)
    mainloop()
```

2.4 Le module `time`

Du module `time` nous n'utiliserons que la fonction `clock` qui renvoie une durée de temps en secondes (dont la signification exacte dépend du système d'exploitation) et qui permet donc par soustraction de mesurer des durées d'exécution.
Par exemple, on cherche à créer la liste des valeurs de la fonction sinus pour les n premiers entiers, avec n assez grand. Le programme suivant (inspiré d'un exemple de [Zia09]) permet de comparer (en terme de temps d'exécution) trois manières différentes de créer une telle liste.

Benchmark.py

```
from time import clock
from math import sin

def f1(n):
    liste = []
    for i in range(n):
        liste += [sin(i)]
    return liste

def f2(n):
    liste = []
    for i in range(n):
        liste.append(sin(i))
    return liste

def f3(n):
    return [sin(i) for i in range(n)]

def duree(fonction, n=10000000):
    debut = clock()
    fonction(n)
    fin = clock()
    return fin - debut

print(' {:-<25}>  {:f} s'.format('utilisation de +=', duree(f1)))
print(' {:-<25}>  {:f} s'.format('utilisation de append', duree(f2)))
print(' {:-<25}>  {:f} s'.format('list-comprehension', duree(f3)))
```

```
utilisation de += ------->  6.480000 s
utilisation de append --->  4.600000 s
list-comprehension ------>  3.550000 s
```

On en conclut que l'utilisation des « list-comprehension » est à privilégier. Remarquer également que l'utilisation de la méthode append est plus efficace que l'instruction `liste += [x]`.

2.5 Le module `decimal`

Revenons quelques instants sur l'exercice 4, et essayons d'échanger le contenu de deux variables comme suit.

```
>>> x, y = 1/3, 2/3
>>> x, y
(0.3333333333333333, 0.6666666666666666)
>>> x = x + y; y = x - y; x = x - y
>>> x, y
(0.6666666666666666, 0.33333333333333337)
```

On remarque que le contenu de y semble légèrement différent du x de départ. Que s'est-il passé ? Tout simplement les calculs effectués ne sont pas exacts et sont entachés d'erreurs d'arrondis.
Autres exemples encore plus surprenants :

```
>>> 1.1 + 2.2
3.3000000000000003
>>> 0.1 + 0.1 + 0.1 - 0.3
5.551115123125783e-17
```

On rappelle que tout nombre réel possède un développement décimal soit fini soit illimité ; et cette remarque fournit en fait un moyen efficace de définir l'ensemble des nombres réels comme l'ensemble des nombres possédant un développement décimal fini ou illimité (on pourra se reporter au chapitre 38 de [CBP10] pour une approche, élémentaire et accessible à un collégien, de cette définition).
Parmi les nombres réels, on peut alors distinguer les rationnels (dont le développement décimal est soit fini, soit illimité et périodique à partir d'un certain rang) des irrationnels (dont le développement décimal est illimité et non périodique).
Il est aisé de concevoir qu'il n'est pas possible pour un ordinateur de représenter de manière exacte un développement décimal illimité. Mais même la représentation des développements décimaux finis n'est pas toujours possible. En effet, un ordinateur stocke les nombres non pas en base 10, mais en base 2. Or un nombre rationnel peut tout à fait posséder un développement décimal fini et un développement binaire illimité ! C'est le cas des décimaux 1.1 et 2.2 mis en jeu dans l'exemple précédent :

base 10	base 2
1.1	01.00011001100110011001100110011...
2.2	10.00110011001100110011001100110...
3.3	11.01001100110011001100110011001...

Contrairement au cas des nombres entiers qui sont représentés en **Python** de manière exacte [3], les nombres réels sont représentés de manière approchée par des nombres décimaux, comportant 15 chiffres significatifs.
Si l'on souhaite travailler avec des nombres flottants en modifiant le nombre de chiffres significatifs, on peut utiliser le module `decimal`. Un objet du type `Decimal` est créé en utilisant la fonction `decimal.Decimal()`. Cette fonction peut prendre en argument un entier ou une chaîne de caractères — mais pas un objet du type `float` puisque ceux-ci sont représentés en mémoire de manière approchée, contrairement aux objets du type `Decimal`.
Reprenons les deux derniers calculs en important ce module :

3. En **Python** 3, les types `int` et `long` ont été fusionnés et donc la taille d'un entier n'est limitée que par la mémoire allouée par l'ordinateur à l'interpréteur **Python**.

```
>>> from decimal import *
>>> Decimal('1.1') + Decimal('2.2')
Decimal('3.3')
>>> Decimal('0.1')+Decimal('0.1')+Decimal('0.1')-Decimal('0.3')
Decimal('0.0')
```

Par défaut, le module `decimal` travaille avec une précision de 28 décimales ; mais il est possible de modifier cette précision :

```
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999999,
Emax=999999999, capitals=1, flags=[],
traps=[Overflow, DivisionByZero, InvalidOperation])
>>> getcontext().prec = 318       # Fixer une nouvelle précision
```

Illustrons de nouveau les problèmes d'arrondis en partant de l'identité remarquable suivante, que l'on trouve énoncée (en terme d'aires de rectangle) dans la proposition 5 du livre II des éléments d'EUCLIDE :

$$xy = \left(\frac{x+y}{2}\right)^2 - \left(\frac{x-y}{2}\right)^2$$

Dans le programme suivant, on compare les deux membres de cette égalité pour des nombres x et y devenant de plus en plus grands.

euclide.py

```
from decimal import *

def prod(x, y): return x*y

def diff(x, y): return ((x+y)/2)**2 - ((x-y)/2)**2

def euclide(a, b):
    c, d = Decimal(str(a)), Decimal(str(b))
    print('-'*53)
    print('{:<25s} | {:<25s}'.format('type float', 'type Decimal'))
    print('-'*53)
    for n in range(6):
        u, v = prod(a, b), diff(a, b)
        w, x = prod(c, d), diff(c, d)
        print('{:>25g} | {:>25g}'.format(u - v, w - x))
        a, b = u, a + 1
        c, d = w, c + 1
    print('-'*53)

a = 6553.99
b = a + 1

print('{:*^53}'.format(' getcontext().prec = 28 '))
print(); euclide(a, b); print()
getcontext().prec = 128
print('{:*^53}'.format(' getcontext().prec = 128 '))
euclide(a, b)
```

```
************** getcontext().prec = 28 ***************
-----------------------------------------------------
type float                | type Decimal
-----------------------------------------------------
                        0 |                       0.0000
               -0.0585938 |                        0e-10
              1.66707e+06 |                  0.00000573
             -4.79826e+21 |                6.782540e+9
              1.04665e+44 |          3.35746553058e+33
              1.40437e+80 |  2.177355930228656542e+71
-----------------------------------------------------

************** getcontext().prec = 128 **************
-----------------------------------------------------
type float                | type Decimal
-----------------------------------------------------
                        0 |                       0.0000
               -0.0585938 |                        0e-10
              1.66707e+06 |                        0e-14
             -4.79826e+21 |                        0e-22
              1.04665e+44 |                        0e-34
              1.40437e+80 |         1.842018954699e-30
-----------------------------------------------------
```

On constate que la divergence des calculs est spectaculaire !

2.6 Le module `fractions`

Ce module permet de manipuler des nombres rationnels. L'exemple suivant se passe de commentaires.

```
>>> 1/3 + 2/5
0.7333333333333334
>>> from fractions import Fraction
>>> Fraction(1, 3) + Fraction(2, 5)
Fraction(11, 15)
>>> Fraction(5, 15)
Fraction(1, 3)
```

2.7 Le module sys

Le module `sys` contient des fonctions et des variables spécifiques au système, ou plus exactement à l'interpréteur lui-même. Ce module est particulièrement intéressant pour récupérer les arguments passés à un script **Python** lorsque celui-ci est appelé en ligne de commande. La variable `sys.argv` de ce module contient la liste représentant tous les arguments de la ligne de commande, y compris le nom du script lui-même qui en est le premier élément.

arguments.py

```
import sys
print(sys.argv)
```

Terminal
```
$ python3 arguments.py voici un exemple 1 2 3
['arguments.py', 'voici', 'un', 'exemple', '1', '2', '3']
```

Par exemple, on peut écrire un petit programme qui s'utilise en ligne de commande et calcule la moyenne arithmétique des nombres passés en arguments du programme.

moyenne.py
```
import sys
print(sum(eval(x) for x in sys.argv[1:]) / len(sys.argv[1:]))
```

Terminal
```
$ chmod u+x moyenne.py
$ ./moyenne.py 4 0.2 -3 8 -5
0.84
```

3 Lire et écrire dans un fichier

En **Python**, l'accès à un fichier est assuré par l'intermédiaire d'un objet particulier appelé objet-fichier et créé à l'aide de la primitive `open(nom, mode)` comme ceci :

```
>>> objet = open('mon_fichier.txt', 'w')
```

Une fois l'objet-fichier créé, il est doté de méthodes spécifiques, qui permettent notamment de lire et écrire dans le fichier, suivant le mode d'ouverture qui a été précisée dans le paramètre `mode`. Ce paramètre de type « chaîne de caractères » peut prendre plusieurs valeurs :

- `r` : ouverture pour lecture seule ;
- `w` : ouverture pour écriture : si le fichier n'existe pas, il est créé, sinon son contenu est écrasé ;
- `a` : ouverture pour ajout : si le fichier n'existe pas, il est créé, sinon l'écriture s'effectue à la suite du contenu déjà existant ;
- `+` : ouverture pour lecture *et* écriture.

Une fois la lecture ou l'écriture dans le fichier terminée, on referme le fichier à l'aide de l'instruction `close`.
Voici un exemple d'ouverture de fichier avec affichage d'un message d'erreur si le fichier n'existe pas.

```
nom = 'mon_fichier.txt'
objet = open(nom, 'r')
if objet:
    objet.close()
else:
    print("Erreur le fichier {} n'existe pas".format(nom))
```

Pour écrire dans un fichier, on peut utiliser l'instruction `write` qui écrit une chaîne de caractères ou l'instruction `writelines` qui écrit une liste de chaînes de caractères en les concaténant.

Pour lire l'intégralité du contenu d'un fichier, on dispose de l'instruction `read` ; si l'on souhaite récupérer une liste contenant l'ensemble des lignes du fichiers, on utilise l'instruction `readlines`.
Ces fonctions sont illustrées dans le script suivant :

manip_fichier.py

```
# Création d'un nouveau fichier et écriture
nom = 'mon_fichier.txt'
fichier = open(nom, 'w')
for i in range(1, 4):
    fichier.write('Ceci est la ligne {}\n'.format(i))
fichier.close()

# Lecture
fichier = open(nom, 'r')
liste = fichier.read()
print(liste)
fichier.close()

# Ajout en fin de fichier
fichier = open(nom, 'a')
for i in range(4, 6):
    fichier.write('Ceci est la ligne {}\n'.format(i))
fichier.close()

# Variante pour la lecture
fichier = open(nom, 'r')
liste = fichier.readlines()
print(liste)
fichier.close()

# Variante pour la lecture
fichier = open(nom, 'r')
for ligne in fichier:
    print(ligne, end='')
fichier.close()
```

L'exécution du script précédent donne ceci :

```
Ceci est la ligne 1
Ceci est la ligne 2
Ceci est la ligne 3

['Ceci est la ligne 1\n', 'Ceci est la ligne 2\n', 'Ceci est la ligne 3\n', 'Ceci
    est la ligne 4\n', 'Ceci est la ligne 5\n']
Ceci est la ligne 1
Ceci est la ligne 2
Ceci est la ligne 3
Ceci est la ligne 4
Ceci est la ligne 5
```

Lorsqu'on manipule un fichier, il est recommandé d'utiliser le mot-clé `with` qui a l'avantage de fermer proprement le fichier une fois le bloc d'instructions correspondant exécuté, et cela même si une exception a été atteinte dans ce dernier. De plus, il permet une syntaxe plus concise :

```
>>> with open('mon_fichier', 'r') as f:
...     lire = f.read()
>>> f.closed
True
```

Exercice 24. Écrire une fonction qui renvoie le nombre de lignes d'un fichier.

Solution.

```
def nbre_lignes(nom):
    with open(nom, 'r') as f:
        n = len(f.readlines())
    return n
```

Exercice 25. Écrire une fonction qui renvoie la liste de tous les mots d'un fichier. On utilisera la méthode `split` qui s'applique à une chaîne de caractères et dont on cherchera la fonction dans la documentation officielle.

Solution.

```
def extraction(nom):
    liste = []
    with open(nom, 'r') as f:
        for ligne in f:
            for mot in ligne.split():
                liste.append(mot)
    return(liste)
```

Signalons que **Python** dispose de très nombreux modules permettant d'appeler le système d'exploitation pour manipuler les fichiers et répertoires, notamment le module `os`, dont la présentation dépasse le cadre de cet ouvrage[4].

```
>>> from os.path import *
>>> print(exists('mon_fichier.txt'))
True
>>> print(abspath('mon_fichier.txt'))
/home/utilisateur1/mon_fichier.txt
>>> from os import system  # pour exécuter une commande externe
>>> system('cat mon_fichier.txt')
Ceci est la ligne 1
Ceci est la ligne 2
Ceci est la ligne 3
Ceci est la ligne 4
Ceci est la ligne 5
0
```

4. Pour une introduction concise et rapide aux commandes usuelles du `bash`, on pourra consulter [Gra06].

Exercice 26. L'extension `tikz` du langage de composition LaTeX ne permettant pas pour l'instant un tracé aisé des courbes en polaires, écrire un script qui effectue les opérations suivants :

- calcul des points de la courbe d'équation polaire $\rho = 1 + \frac{1}{3} \cdot \cos\left(\frac{20 \cdot \theta}{19}\right)$;
- écriture de ces points dans un fichier (esclave) d'extension « `.table` » ;
- écriture d'un fichier (maître) LaTeX contenant le préambule et l'environnement `tikz` et appelant le fichier esclave ;
- compilation du document maître à l'aide du programme `rubber`[5].

Solution.

courbetikz.py

```
from math import *
N, a, b = 500, 0, 1 + floor(38 * pi)

def rho(theta):
    return 1 + cos(theta*20/19) / 3

def echelle(t, xmin=a, xmax=b, numpoints=N):
    return xmin + t/numpoints * (xmax - xmin)

s = ("\\documentclass{article}\n\n"
     "\\usepackage[mathletters]{ucs}\n"
     "\\usepackage[utf8x]{inputenc}\n"
     "\\usepackage[T1]{fontenc}\n"
     "\\usepackage{xcolor}\n"
     "\\usepackage{tikz}\n\n"
     "\\begin{document}\n"
     "\\begin{tikzpicture}[scale=2]\n"
     "\\clip (-1.7,-1.7) rectangle (1.7,1.7);\n"
     "\\draw[->] (-1.5,0) -- (1.5,0) node[right] {$x$};\n"
     "\\draw[->] (0,-1.5) -- (0,1.5) node[above] {$y$};\n"
     "\\draw[color=blue] plot[smooth] file {courbe.table};\n"
     "\\end{tikzpicture}\n"
     "\\end{document}\n")

with open("courbe.tex", 'w') as f:
    f.writelines(s)

with open("courbe.table", 'w') as f:
    for i in range(N):
        theta = echelle(i)
        r = rho(theta)
        x, y = r * cos(theta), r * sin(theta)
        f.write("{:8.5f}    {:8.5f}\n".format(x, y))

import os
os.system("rubber -d {0}.tex && evince {0}.pdf &".format('courbe'))
```

5. cf. https://launchpad.net/rubber

4 Manipulation de fichiers CSV

Le format « Comma-separated values » (CSV) est un format informatique ouvert représentant des données tabulaires sous forme de « valeurs séparées par des virgules » ; c'est le format le plus couramment utilisé pour importer ou exporter des données d'une feuille de calcul d'un tableur.
Un fichier CSV est un fichier texte, dans lequel chaque ligne correspond à une rangée du tableau ; les cellules d'une même rangée sont séparées par une virgule.
À partir d'un tableur, il suffit de sauvegarder une feuille de calcul en précisant le format CSV pour fabriquer un tel fichier.
Pour éviter les désagréments liés à l'absence de standardisation du séparateur décimal, il est fortement conseillé de paramétrer son tableur pour choisir le point (et non la virgule) comme séparateur décimal. Si vous utilisez `OpenOffice.org Calc` ou `LibreOffice Calc`, allez dans le menu Outils > Options > Paramètres linguistiques > Langue > Paramètre linguistique et choisissez une combinaison langue/pays utilisant le point comme séparateur décimal : « anglais (tous pays) » ou mieux « français (Suisse) ».
La figure suivante montre un exemple de feuille de calcul ouverte dans `OpenOffice` :

Le fichier CSV correspondant sera

```
"Noms","maths","physique","chimie"
"Achille",12,14,15
"Ajax",17,11,9
"Alceste",15,15,16
"Alcibiade",11,13,12
"Aspasie",19,15,18
"Berenice",14,17,17
```

Pour ouvrir un fichier CSV avec **Python**, on importe le module `csv` et on utilise la syntaxe suivante :

lireCSV.py

```
import csv
with open('test.csv', newline='') as f:
    lire = csv.reader(f)
    for ligne in lire:
        print(ligne)
```

```
['Noms', 'maths', 'physique', 'chimie']
['Achille', '12', '14', '15']
['Ajax', '17', '11', '9']
['Alceste', '15', '15', '16']
['Alcibiade', '11', '13', '12']
['Aspasie', '19', '15', '18']
['Berenice', '14', '17', '17']
```

Pour écrire dans un fichier, on utilise l'instruction open avec l'option `'w'` pour écrire ou `'a'` pour ajouter du contenu.

ecrireCSV.py

```
table = [['Noms', 'maths', 'physique', 'chimie'],
         ['Achille', '12', '14', '15'], ['Ajax', '17', '11', '9'],
         ['Alceste', '15', '15', '16'], ['Alcibiade', '11', '13', '12'],
         ['Aspasie', '19', '15', '18']]

import csv
with open('test.csv', 'w', newline='') as f:
    ecrire = csv.writer(f)
    for ligne in table:
        ecrire.writerow(ligne)

with open('test.csv', 'a', newline='') as f:
    ecrire = csv.writer(f)
    ecrire.writerow(['Berenice', '14', '17', '17'])
```

Il faut bien prendre garde au fait qu'avec l'option `'w'`, la méthode `writer` écrase le fichier s'il existe déjà.

Écrivons à présent un script qui ajoute au fichier `test.csv` les moyennes de chaque élève et les moyennes de la classe pour chaque matière.

moyennesCSV.py

```
import csv
with open('test.csv', newline='') as f:
    lignes = [ligne for ligne in csv.reader(f)]

def moy(liste):
    liste = [eval(x) for x in liste]
    return str(round(sum(liste)/len(liste), 1))

lignes[0].append('moyennes')
for ligne in lignes[1:]:
    ligne.append(moy(ligne[1:]))
```

```
classe = [moy([lig[i] for lig in lignes[1:]]) for i in range(1, 5)]
classe[0:0] = ['moyennes']
lignes.append(classe)

with open('test.csv', 'w', newline='') as f:
    ecrire = csv.writer(f)
    for ligne in lignes:
        ecrire.writerow(ligne)
```

Le résultat est alors le suivant :

5 Comment générer des graphiques ?

Représenter des fonctions ou des courbes paramétrées est un besoin constant en mathématiques. Pour ce faire, plusieurs méthodes se présentent à nous. Parmi les modules fournis par défaut dans **Python**, on pourrait utiliser `turtle` ou `Tkinter`. Le premier a l'inconvénient d'être assez rudimentaire et le second peut demander beaucoup d'efforts pour un résultat correct. Parmi les modules de tierces parties, celui qui est certainement le plus adapté est `matplotlib`, dont nous parlerons brièvement dans la prochaine section. Néanmoins sa grande puissance a un revers : sa prise en main n'est pas chose aisée.
À titre pédagogique, nous proposons donc ici une autre approche. **Python** est connu pour être une bonne « colle » entre différents langages ou logiciels or, aujourd'hui, une bonne maîtrise de l'outil informatique nécessite de pouvoir faire interagir différents langages ou logiciels entre eux. Nous allons donc montrer ici comment générer des graphiques dans le langage de description de page `PostScript`. Un intérêt évident de cette méthode est qu'il nous permettra non seulement de visualiser une courbe, mais bien plus, le tracé en sera obtenu au format PostScript encapsulé. La conversion au format PDF peut se faire ensuite, par exemple, avec

le logiciel payant Acrobat Distiller, ou bien avec le logiciel gratuit ghostscript (via l'utilitaire `eps2pdf`). Sous l'un de ces deux formats, il sera facile d'intégrer ce tracé dans un document LaTeX par exemple.

Voici un module simple pour créer un fichier EPS à partir d'une liste de points à tracer.

PostScript0.py

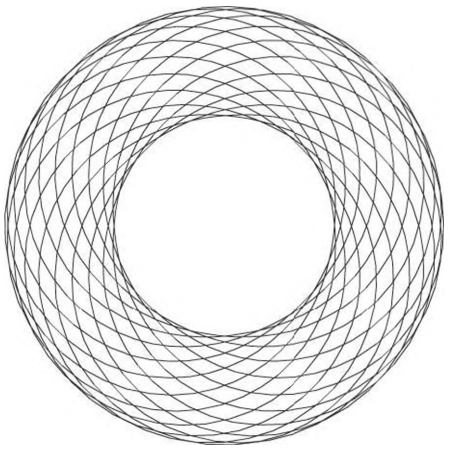
```
"""
   Module de tracé d'une liste de points dans un fichier PostScript
"""
import os

def nrange(a, b, numpoints):
    """Renvoie une subdivision de [a, b] à N+1 points."""
    pas = (b - a) / numpoints
    return (a + i * pas for i in range(numpoints + 1))

def preambule(nomFichier, boite, zoom, delta):
    """ Écrit le préambule du fichier EPS."""
    cadre = [x * zoom * delta for x in boite]
    s_debut = ("%!PS-Adobe-2.0 EPSF-2.0\n"
    "%%BoundingBox: {0[0]:.1f} {0[1]:.1f} {0[2]:.1f} {0[3]:.1f}\n"
    "{1} {1} scale\n").format(cadre, zoom)
    with open(nomFichier + ".eps", 'w') as f:
        f.write(s_debut)

def fin(nomFichier):
    """ Clôture le fichier EPS."""
    s_fin = "\nshowpage\n"
    with open(nomFichier + ".eps", 'a') as f:
        f.write(s_fin)

def ajouteCourbe(nomFichier, liste, boite, zoom, epaisseurTrait, rgb):
    """Ajoute une courbe donnée sous forme de liste."""
    with open(nomFichier + ".eps", 'a') as f:
        f.write("\nnewpath\n")
        for i, point in enumerate(liste):
            if i == 0:
                f.write("    {0[0]: .4f}  {0[1]: .4f}   ".format(point))
                f.write("moveto\n")
            elif (boite[0] <= point[0] <= boite[2]
                    and boite[1] <= point[1] <= boite[3]):
                f.write("    {0[0]: .4f}  {0[1]: .4f}   ".format(point))
                f.write("lineto\n")
        f.write("{1} {0} div setlinewidth\n"
                "{2[0]} {2[1]} {2[2]} setrgbcolor\n"
                "stroke\n".format(zoom, epaisseurTrait, rgb))

def affiche(nomFichier):
    """Affiche le graphique via ghostview."""
    os.system("gv {0}.eps &".format(nomFichier))
```

```
if __name__ == "__main__":
    from math import pi, cos, sin, floor

    N, a, b = 1000, 0, 1 + floor(38 * pi)
    nomFichier = "polar"
    zoom, epaisseurTrait, rgb = 100, 0.4, (0, 0, 1)
    boite = [-1.5, -1.5, 1.5, 1.5] # xmin , ymin, xmax, ymax

    # Fonction définissant la courbe polaire
    def f(theta):
        return 1 + cos(theta*20/19) / 3

    # Liste de points de la courbe
    liste = ([f(theta) * cos(theta), f(theta) * sin(theta)]
            for theta in nrange(a, b, N))

    # Création du fichier EPS
    preambule(nomFichier, boite, zoom, 1.1)
    ajouteCourbe(nomFichier, liste, boite, zoom, epaisseurTrait, rgb)
    fin(nomFichier)
    affiche(nomFichier)
```

L'exemple présenté en fin de module reprend le problème du tracé de la courbe d'équation polaire :

$$\rho = 1 + \frac{1}{3} \cdot \cos\left(\frac{20 \cdot \theta}{19}\right)$$

Une fois créée la liste de points à tracer, on utilise les fonctions `preambule`, `ajouteCourbe` et `fin` pour obtenir la figure ci-contre.



PostScript0.py

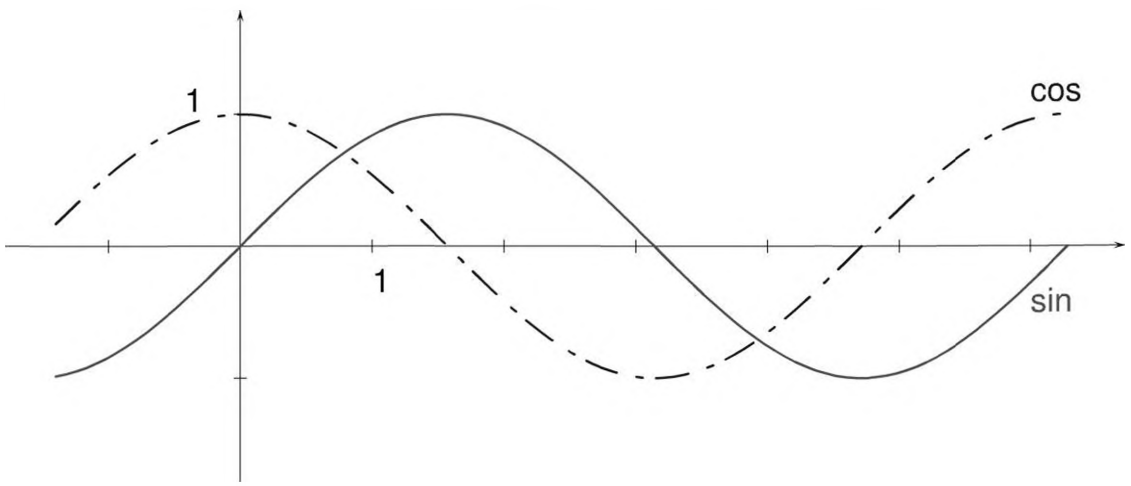
```
from math import *
import PostScript0 as ps

N, a, b = 1000, 0, 1 + floor(38 * pi)
nomFichier, zoom, epaisseurTrait, rgb = "polar", 100, 0.4, (0, 0, 1)
boite = [-1.5, -1.5, 1.5, 1.5] # xmin , ymin, xmax, ymax

# Fonction définissant la courbe polaire
def f(theta): return 1 + cos(theta*20/19) / 3
```

```
# Liste de points de la courbe
liste = ([f(theta) * cos(theta), f(theta) * sin(theta)]
        for theta in ps.nrange(a, b, N))

# Création du fichier EPS
ps.preambule(nomFichier, boite, zoom, 1.1)
ps.ajouteCourbe(nomFichier, liste, boite, zoom, epaisseurTrait, rgb)
ps.fin(nomFichier)
ps.affiche(nomFichier)
```

Dans l'archive disponible sur la Toile, le lecteur trouvera un module `PostScript.py` amélioré qu'il suffira d'importer pour pouvoir obtenir les graphiques présents dans cet ouvrage.
Pour une utilisation simplifiée, on peut utiliser directement la fonction `plot` qui prend en argument le nom du fichier PostScript à générer, une liste `boite=[xMin, yMin, xMax, yMax]`, le facteur d'agrandissement `zoom`, la liste des courbes à tracer, chaque courbe étant donnée sous forme d'une liste contenant la liste des points, la couleur et éventuellement l'épaisseur de trait. Du texte peut également être placé sur le graphique. Enfin, des arguments optionnels permettent de modifier le paramétrage par défaut, notamment du tracé des axes.
Voici un exemple d'utilisation :

PostScript.py

```
from math import *
from PostScript import plot, nrange

# Paramètres à fixer
numpoints, a, b, boite = 100, - 1.4 , 2 * pi, [-1.7, -1.7, 6.4, 1.7]
nomFichier, zoom, rouge, bleu = "sin_cos", 100, (1, 0, 0), (0, 0, 1)

# Liste de points de la courbe
liste_s = [[x, sin(x)] for x in nrange(a, b, numpoints)]
liste_c = [[x, cos(x)] for x in nrange(a, b, numpoints)]

plot(nomFichier, boite, zoom,
        [[liste_s, bleu], [liste_c, rouge]],
        ['sin', [6, -0.5], bleu], ['cos', [6, 1.1], rouge])
```



6 Un coup d'œil vers le module Matplotlib

Pour utiliser le module de tierce partie `Matplotlib`, il faut au préalable l'installer, ainsi que le module `numpy`[6]. Une fois ces modules installés, on importe le module `mumpy` pour pouvoir utiliser le type `array`. Un objet du type `array` est un tableau dont les éléments sont tous du même type, contrairement à ce qui se passe dans le cas des objets de type `list`. On peut se servir d'un `array` unidimensionnel pour représenter un vecteur mathématique, et d'un `array` multidimensionnel pour représenter une matrice mathématique.
Voici un exemple de création de tableau à une dimension dont on déclare que les quatre composantes sont des flottants.

```
>>> import numpy as np
>>> x = np.array([0, 1, 4, 9], float)
>>> x          # rép.: array([ 0.,  1.,  4.,  9.])
>>> type(x)    # rép.: <class 'numpy.ndarray'>
```

Pour créer des tableaux du type `array`, le module `numpy` possède plusieurs primitives, dont `linspace`. Cette dernière permet de stocker dans un tableau les points d'une subdivision régulière d'intervalle. Voici, par exemple, une subdivision de l'intervalle $[-1,1]$ comprenant 5 points :

```
>>> np.linspace(-1, 1, 5)           # <--- (min, max, nbre de pts)
array([-1. , -0.5,  0. ,  0.5,  1. ])
```

Soit à tracer la fonction sinus cardinal $f : x \mapsto \frac{\sin x}{x}$ sur l'intervalle $[-15,15]$. Pour tracer cette fonction, la primitive de `matplotlib` aura besoin de deux tableaux, le premier contenant les abscisses des points à tracer, le second leurs ordonnées.
Nous connaissons déjà le moyen de créer le tableau des abscisses. Pour créer le tableau des ordonnées, on pourrait procéder comme suit :

```
>>> x = np.linspace(-1, 1, 4)                    # les abscisses
>>> x
array([-1.        , -0.33333333,  0.33333333,  1.        ])
>>> import math                                  # pour la fonction sinus !
>>> y = np.array([math.sin(i) / i for i in x])   # les ordonnées
>>> y
array([ 0.84147098,  0.98158409,  0.98158409,  0.84147098])
```

Mais il y a bien plus simple : en effet, les objets de type `array` supportent la vectorisation. Il existe dans le module `numpy` des primitives permettant d'appliquer en un clin d'œil une fonction mathématique usuelle à tous les éléments d'un tableau !

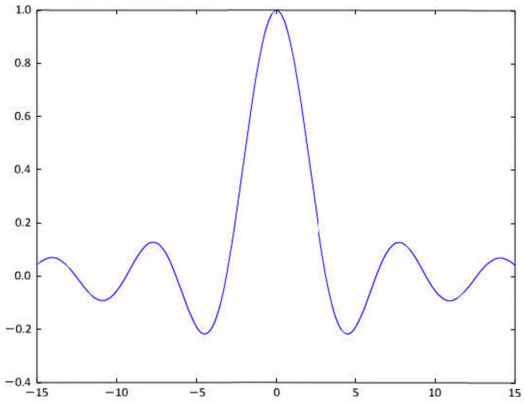
```
>>> x = np.linspace(-1, 1, 4)                    # les abscisses
>>> y = np.sin(x) / x                            # les ordonnées
>>> y
array([ 0.84147098,  0.98158409,  0.98158409,  0.84147098])
```

À présent, tout est en place pour utiliser `matplotlib`. On commence par saisir la commande `import matplotlib.pyplot` qui importe une collection de primitives permettant d'utiliser `matplotlib` avec une syntaxe proche de celle de MATLAB. On crée ensuite l'objet graphique correspondant au graphe de notre fonction et il ne nous reste plus qu'à l'exhiber :

6. Consulter http://www.scipy.org/install.html et http://matplotlib.org/users/installing.html.

sinusc.py

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-15, 15, 150)
y = np.sin(x) / x

plt.plot(x, y)
plt.show()
```

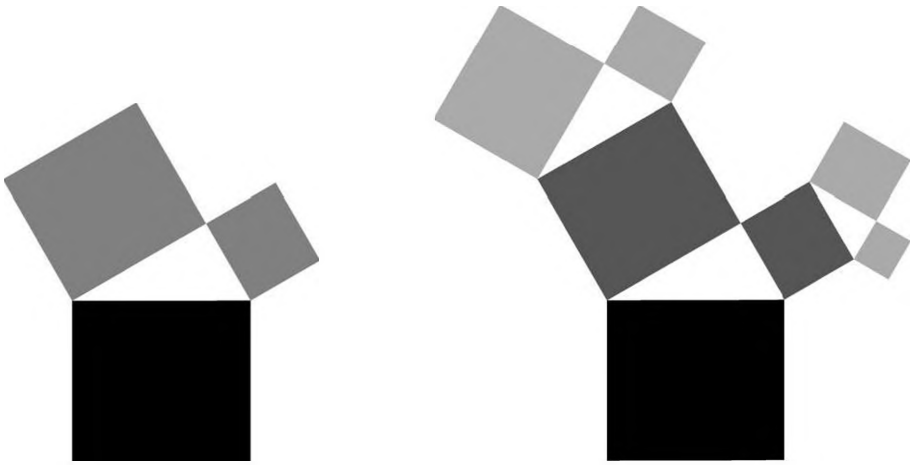


7 Exercices d'entraînement

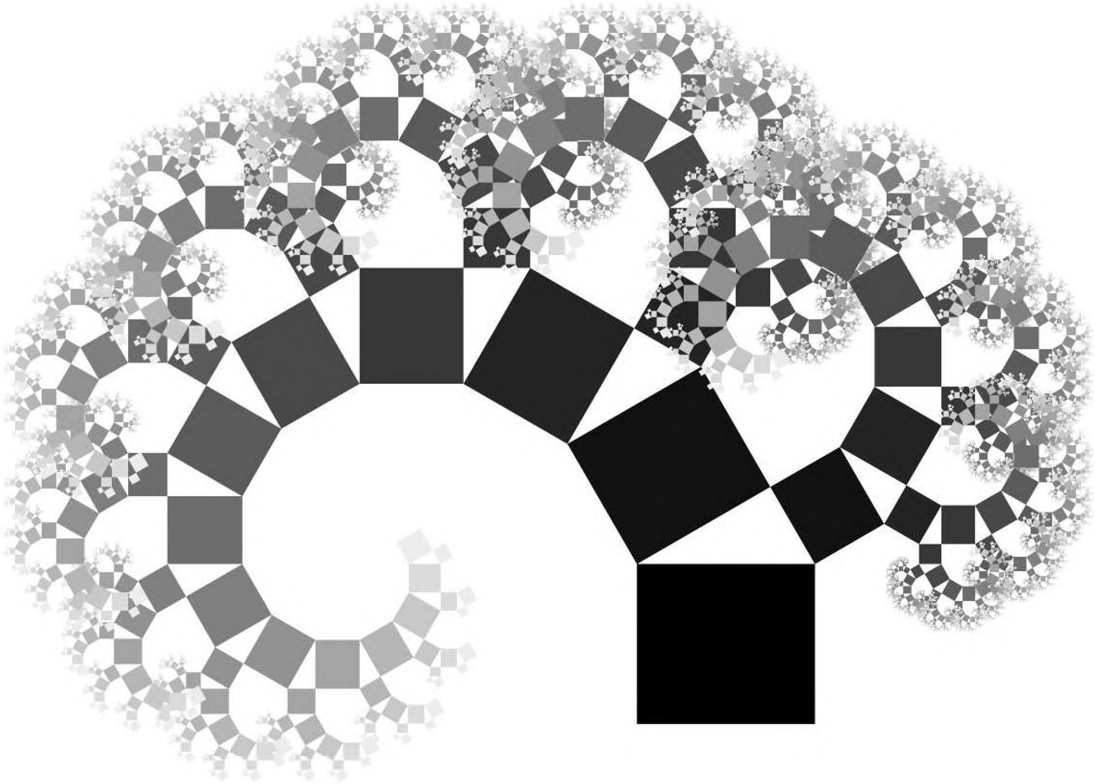
Les corrigés sont disponibles sur le site dunod.com à partir de la page d'accueil de l'ouvrage.

Exercice 27. En utilisant la primitive `walk` du module `os` et la méthode `endswith` s'appliquant à une chaîne de caractères, écrire un programme qui parcourt l'arborescence d'un répertoire et affiche le nom de tous les scripts **Python** rencontrés, précédé du chemin absolu. Si aucun répertoire n'est fourni en argument, le programme explorera le répertoire courant.

Exercice 28. (Arbre de Pythagore)
La construction de l'arbre de Pythagore débute avec un simple carré. On trace alors les côtés d'un triangle rectangle ayant pour hypoténuse le côté supérieur du carré. Sur les bords de ce triangle, on construit deux carrés. Sur les côtés supérieurs de ces carrés, on construit deux nouveaux triangles homothétiques au premier. La procédure est appliquée récursivement à chaque carré, jusqu'à l'infini. Les dessins ci-dessous illustrent les deux premières itérations de la construction.



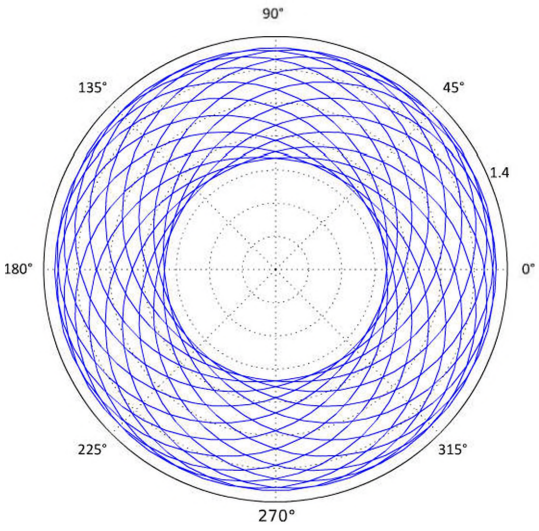
Écrire un programme qui dessine l'arbre de Pythagore au bout d'un nombre donné d'itérations. On pourra utiliser le module Postscript.py pour créer le graphique. Au bout de 14 itérations, on obtient :

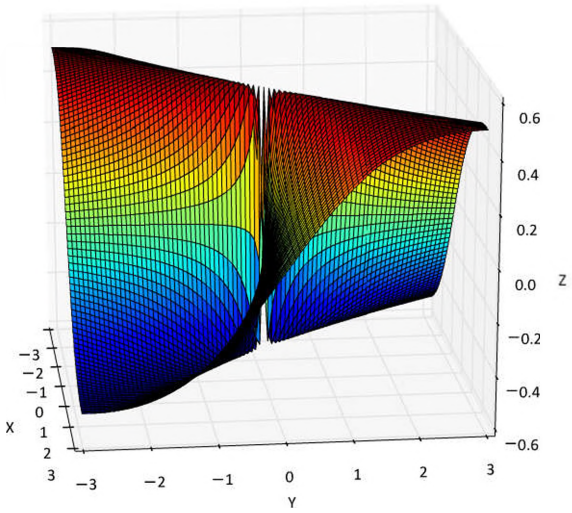


Exercice 29. (Utilisation de `matplotlib`)

a) En s'aidant de l'aide en ligne de `matplotlib`, tracer la rosace rencontrée à l'exercice 26.

b) Obtenir la surface représentative de la fonction : $f\colon (x_1, x_2) \in \mathbb{R}^2 \setminus \{(0,0)\} \longmapsto \frac{x_1 x_2}{x_1^2 + x_2^2}$





3

Thèmes mathématiques

Sommaire

1 Matrices

1.1 Les matrices avec NumPy et SciPy

Le module `scipy` propose tous les outils nécessaires au calcul matriciel en utilisant les tableaux de `numpy` : les opérations arithmétiques basiques ainsi qu'une soixantaine de méthodes couvrant les besoins standards et au-delà.

```
>>> import numpy as np
>>> from scipy import linalg
>>> A = np.array([(1,2),(3,4)])
>>> A
array([[1, 2],
       [3, 4]])
>>> 3 * A   # chaque terme est multiplié par 3
array([[ 3,  6],
       [ 9, 12]])
>>> A + 10  # chaque terme est additionné à 10
array([[11, 12],
       [13, 14]])
>>> A.T     # transposée
array([[1, 3],
       [2, 4]])
```

Attention, l'opérateur $*$ effectue le produit terme à terme. Pour multiplier deux matrices au sens usuel, il faut utiliser `dot` :

```
>>> A * A
array([[ 1,  4],
       [ 9, 16]])
```

```
>>> A ** 2
array([[ 1,  4],
       [ 9, 16]])
```

```
>>> A.dot(A)
array([[ 7, 10],
       [15, 22]])
```

On peut calculer le déterminant et l'inverse d'une matrice facilement :

```
>>> 1/A # c'est l'inverse terme à terme
array([[ 1.        ,  0.5       ],
       [ 0.33333333,  0.25      ]])
>>> linalg.det(A)
-2.0
```

```
>>> iA = linalg.inv(A)
>>> iA
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

mais...

```
>>> A.dot(iA)
array([[  1.00000000e+00,   0.00000000e+00],
       [  8.88178420e-16,   1.00000000e+00]])
```

...compte-tenu des erreurs d'arrondis dont nous reparlerons plus tard (cf sections 2.4 page 132 et 1 page 180). On peut cependant y remédier avec la fonction `allclose` :

```
>>> help(np.allclose)
Help on function allclose in module numpy.core.numeric:
allclose(a, b, rtol=1e-05, atol=1e-08)
    Returns True if two arrays are element-wise equal within a tolerance.
```

Alors, sachant que `np.eye(n)` renvoie la matrice identité de taille n :

```
>>> np.allclose(np.dot(A, iA), np.eye(2))
True
```

De nombreux exemples et une présentation quasi exhaustive des possibilités de ces deux modules peuvent être consultés sur https://scipy-lectures.github.io/intro/numpy/numpy.html et https://scipy-lectures.github.io/intro/scipy.html#linear-algebra-operations-scipy-linalg.

1.2 Fabriquons nos outils

Utiliser `numpy` et `scipy` comme une boîte noire peut être intéressant dans certains contextes, notamment pour l'ingénieur (sauf peut-être si c'est un ingénieur informaticien...). Nous nous intéressons plutôt dans cet ouvrage à programmer nos propres outils pour réfléchir aux mathématiques sous-jacentes. Nous allons donc perdre en efficacité de temps de calcul mais gagner en compréhension des notions abordées et des difficultés à les traduire en programme.

1.2.1 Création d'une classe « Matrice »

Dans cette section, nous créerons nos matrices en donnant leur dimension et la fonction définissant leurs coefficients en fonction de leurs indices. Comme Python est un langage objet, autant créer une classe qui débutera ainsi et que nous complèterons au fur et à mesure :

matrice.py

```
class Mat:
    """ une matrice sous la forme Mat([nb lignes, nb cols], fonction(i,j))"""

    def __init__(self, dim, f):
        self.F = f   # fonction (i,j) -> coeff en position i,j
        self.D = dim # liste [nb de lignes, nb de cols]
```

Par exemple, $\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$ sera creée avec :

```
>>> M = Mat([3,2],lambda i,j : 2*i + (j + 1))
```

On a quand même l'habitude de rentrer une matrice comme une liste de lignes. On va donc créer une fonction qui permet de convertir une liste de lignes en matrice :

matrice.py

```
def list2mat(mat):
    r,c = len(mat),len(mat[0])
    return Mat([r,c], lambda i,j : mat[i][j])
```

La matrice précédente aurait donc pu être définie par :

```
>>> M = list2mat([[1,2],[3,4],[5,6]])
```

Attention ! On rappelle qu'en **Python**, le premier élément d'une liste est indexé par 0.

```
>>> M.D # dimension de la matrice
[3, 2]
```

```
>>> M.F(0,1) # coef. à la position 0,1
2
```

Exercice 30. Écrivez une fonction qui renvoie une matrice nulle de taille $n \times m$ puis une autre qui renvoie la matrice identité de taille n.

Solution.

```
def zeros(n,m):
    return Mat([n,m], lambda i,j : 0)

def unite(n):
    return Mat([n,n], lambda i,j: 1 if i == j else 0)
```

Nous rajoutons quelques méthodes habituellement définies pour les objets structurés de ce type :

matrice.py

```
    def __getitem__(self,cle):
        """ permet d'obtenir Mij avec M[i,j] """
        return self.F(*cle)

    def __iter__(self):
        """ pour itérer sur la liste des coefficients donnés par colonnes """
        [r,c] = self.D
        for j in range(c):
            for i in range(r):
                yield self.F(i,j)
```

L'emploi du mot-clé `yield` au lieu de `return` permet de ne *retourner* `self.F(i,j)` que lorsque cela est demandé. Ainsi la liste de tous les coefficients n'est pas créée en entier systématiquement ce qui économise la mémoire.
Nous pouvons ainsi créer une méthode qui va permettre un joli affichage :

matrice.py

```
    def __str__(self):
        """ joli affichage d'une matrice"""
        [r,c],f = self.D, self.F
        lmax = len(str(max(iter(self)))) + 1
        s = '\n'.join( (' '.join('{0:>.{l}G}'.format(f(i,j),l=lmax) for j in
            range(c))) for i in range(r))
        return s

    def __repr__(self):
        """ représentation dans le REPL"""
        return str(self)
```

La méthode `join` est à employer systématiquement pour la concaténation des chaînes de caractères : elle est beaucoup plus efficace que son pendant sous forme de boucle...

```
>>> M
1 2
3 4
5 6
```

Exercice 31. Créez deux méthodes qui créent un itérateur respectivement sur les lignes et sur les colonnes. Par exemple, la première colonne de M sera donnée par :

```
>>> [i for i in M.col(0)]
[1, 3, 5]
```

Solution.

matrice.py

```
    def ligne(self,i):
        for j in range(self.D[1]):
            yield self.F(i,j)
```

matrice.py

```
    def col(self,j):
        for i in range(self.D[0]):
            yield self.F(i,j)
```

1.2.2 Transposée d'une matrice

La transposée d'une matrice $(M_{ij})_{\substack{1\leqslant i\leqslant n\\1\leqslant j\leqslant m}}$ est égale à la matrice $(M_{ji})_{\substack{1\leqslant i\leqslant n\\1\leqslant j\leqslant m}}$. Attention aux indices et aux nombre de lignes et de colonnes !

Exercice 32. Créez une méthode qui renvoie la transposée d'une matrice.

```
>>> M.transpose()
1 3 5
2 4 6
```

Solution.

matrice.py

```
    def transpose(self):
        """ transposée d'une matrice """
        [r,c] = self.D
        return Mat([c,r], lambda i,j : self.F(j,i))
```

1.2.3 Somme de matrices

On voudrait créer une méthode qui prend comme arguments deux matrices et renvoie leur somme. Il faudra vérifier que les matrices à additionner sont de bonnes tailles : on utilisera la fonction `assert` qui est suivie d'une condition à vérifier et d'un message à affichher en cas de problème.
Pour pouvoir utiliser le symbole + par la suite, on va nommer notre méthode `__add__` :

matrice.py

```
    def __add__(self,other):
        """ somme de deux matrices : utilisation du symbole +"""
        assert self.D == other.D, "tailles incompatibles"
        return Mat(self.D, lambda i,j : self.F(i,j) + other.F(i,j))
```

Par exemple, avec la matrice M précédemment introduite :

```
>>> M + M
  2   4
  6   8
 10  12
```

On définit de même `__neg__` pour l'opposé et `__sub__` pour la soustraction :

matrice.py

```
    def __neg__(self):
        """ opposé d'une matrice : utilisation du symbole -"""
        return Mat(self.D,lambda i,j : - self.F(i,j))

    def __sub__(self,other):
        """ différence de deux matrices : utilisation dyu symbole -"""
        return self + (-other)
```

1.2.4 Produit par un scalaire

On voudrait obtenir la matrice $k \cdot \mathrm{M}$ à partir d'une matrice M et d'un scalaire k.

```
>>> A = Mat([2,3], lambda i,j : 3*i+j)
>>> A
 0  1  2
 3  4  5
```

```
>>> A.prod_par_scal(5)
  0   5  10
 15  20  25
```

Exercice 33. Écrivez une implémentation de la méthode `prod_par_scal`

Solution.

mattrice.py

```
    def prod_par_scal(self,k):
        """ renvoie le produit d'une matrice par un scalaire"""
        return Mat(self.D, lambda i,j: k*self.F(i,j))
```

1.2.5 Produit de matrices

Soit $\mathrm{A} = (a_{ij})_{\substack{1 \leqslant i \leqslant n \\ 1 \leqslant i \leqslant m}}$ et $\mathrm{B} = (b_{ij})_{\substack{1 \leqslant i \leqslant m \\ 1 \leqslant i \leqslant p}}$. Alors $\mathrm{A} \times \mathrm{B} = \mathrm{C}$ avec $\mathrm{C} = (c_{ij})_{\substack{1 \leqslant i \leqslant n \\ 1 \leqslant i \leqslant p}}$ et

$$\forall (i,j) \in \mathbb{N}_n^* \times \mathbb{N}_p^*, \quad c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$$

C'est l'algorithme habituel des trois boucles imbriquées.

Nous allons l'abstraire d'un petit niveau : chaque coefficient c_{ij} est en fait égal au produit scalaire de la ligne i de A et de la colonne j de B.

Commençons donc par construire une fonction qui calcule le produit scalaire de deux itérables. Pour cela, nous allons utiliser une fonction extrêmement importante qui vient du monde de la programmation fonctionelle : `map`.

`map(fonc, iterable(s))` renvoie un itérateur de type `map` mais où chaque élément sera remplacé par son image par la fonction (si c'est une fonction d'une variable) ou par l'image combinée des itérables si c'est une fonction de plusieurs variables. Par exemple :

```
>>> map(lambda x: x*x, [1,2,3,4])
<map at 0x7fe8307f2ef0>
```

```
>>> [i for i in m]
[1 4 9 16]
```

arghhh : l'objet crée est de type `map` et on a son adresse mais on ne sait pas ce qu'il y a dedans...

```
>>> m = map(lambda x: x*x, [1,2,3,4])
>>> [i for i in m]
[1 4 9 16]
```

Mais atention ! m a maintenant été « consommé ». Si on en redemande :

```
>>> [i for i in m]
[]
```

Si on veut en garder une trace on peut utiliser par exemple `list` :

```
>>> m = map(lambda x: x*x, [1,2,3,4])
>>> l = list(m)
```

```
>>> l
[1 4 9 16]
```

Avec une fonction de deux variables et deux itérables :

```
>>> m = map(lambda x, y: x + y, [1,2,3,4], [-1,-2,-3,-4])
>>> list(m)
[0, 0, 0, 0]
```

On peut aller plus vite en utilisant les opérateurs arithmétiques écrits en notation préfixée dans la bibliothèque `operator` :

```
>>> from operator import mul
>>> m = map(mul, [1,2,3,4], [-1,-2,-3,-4])
>>> list(m)
[-1, -4, -9, -16]
```

Pour notre produit scalaire, nous utilisons également la classique fonction `sum` :

matrice.py

```
def prod_scal(it1,it2):
    return sum(map(mul, it1, it2))
```

Par exemple :

```
>>> prod_scal([1,2,3],[4,5,6])
32
```

Exercice 34. Utilisez `prod_scal` pour écrire en une ligne la matrice produit de deux matrices données en arguments.
Vous créerez ainsi une méthode `prod_mat(self,other)` en ajoutant à la ligne précédente une ligne vérifiant que les formats sont corrects.

Solution.

matrice.py

```
    def prod_mat(self,other):
        """ renvoie le produit de deux matrices"""
        [rs,cs],[ro,co] = self.D,other.D
        assert cs == ro, "tailles incompatibles"
        return Mat([rs,co], lambda i,j : prod_scal(self.ligne(i),other.col(j)))
```

Pour utiliser le symbole $*$ pour le produit d'une matrice par une autre matrice ou un scalaire, on crée dans notre classe une méthode `__mul__`. On utilisera la fonction `type` qui teste le type d'un objet.

matrice.py

```
    def __mul__(self,other):
        """ produit d'une matrice par un scalaire ou une matrice : utilisation
            du symbole *"""
        if Mat == type(other):
            return self.prod_mat(other)
        else:
            return self.prod_par_scal(other)
```

1.2.6 Comparaison de plusieurs produits matriciels

Notre produit n'est pas si mal : il s'écrit très simplement et est assez efficace. Comparons-le avec d'autres implémentations.

Avec les outils standards de Python D'abord une implémentation classique en introduisant une liste Python et des matrices du type `array` de `numpy`. On travaillera par exemple avec des tableaux d'entiers (l'argument `dtype = int`) et on utilisera les commandes `shape` et `zeros` de `numpy`.

```
import numpy as np

def pymatmatprod(A, B):
    ''' Multiplication matricielle avec les outils Python usuels'''

    ra, ca = A.shape
    rb, cb = B.shape
    assert ca == rb, "Tailles incompatibles"
    C = np.zeros((ra, cb),dtype = int)
    for i in range(ra):
        Ai,Ci = A[i],C[i]
        for j in range(cb):
            for k in range(ca):
                Ci[j] += Ai[k] * B[k, j]
    return C
```

Avec Cython `Cython` (http://cython.org/) est un langage dont la syntaxe est très proche de celle de Python mais c'est un langage qui permet de générer des exécutables compilés et d'utiliser un style de programmation proche du C. Cela permet d'optimiser grandement Python. Le logiciel de calcul formel Sage http://www.sagemath.org/ est principalement écrit en Cython pour gagner en efficacité.
Voyons un exemple en œuvre. L'usage conjoint de Python et Cython est grandement facilité par le travail dans l'environnement iPython.

iPython

```
In [1]: %load_ext cythonmagic

In [2]:
%%cython
cimport cython
import numpy as np
cimport numpy as np

cdef long[:, :] matprodC( long[:, :]  A, long[:, :]  B):
    '''
    multiplication matricielle via cython et les array de numpy
    '''
    cdef:
        int i, j, k
        int ra = A.shape[0]
        int ca = A.shape[1]
        int rb = B.shape[0]
        int cb = B.shape[1]
        long [:, :] C
        long [:] Ai, Ci

    assert ca == rb, 'Tailles non compatibles'

    C = np.zeros((ra, cb),dtype=int)
    for i in range(ra):
        Ai,Ci = A[i],C[i]
        for j in range(cb):
            for k in range(ca):
                Ci[j] = Ci[j] + Ai[k] * B[k, j]
    return C

def matprod(long [:, :]  A, B):
    return matprodC(A, B)
```

On remarque qu'écrire en Cython revient un peu à écrire en Python mais en déclarant des variables comme en C.
iPython permet également de comparer facilement les temps de calcul avec la commande magique `%timeit`.
On commence par créer une matrice de taille 200×200 puis on va demander son coefficient [100,100] :

iPython

```
In [3]: A = np.ones((200,200),dtype = int)
```

On compare alors le produit via Python, Cython et la multiplication de numpy :

iPython

```
In [4]: %timeit matprod(A,A)[100,100]
100 loops, best of 3: 16.3 ms per loop

In [5]: %timeit pymatmatprod(A,A)[100][100]
1 loops, best of 3: 8.35 s per loop

In [6]: %timeit np.dot(A,A)[100,100]
100 loops, best of 3: 8.94 ms per loop
```

Notre fonction en Cython est 500 fois plus rapide!!! Elle est très légèrement plus lente que le produit de numpy.
Et notre implémentation personnelle avec la classe Mat ?

iPython

```
In [7]: A = Mat([200,200],lambda i,j:1)

In [8]: %timeit (A*A)[100,100]
10000 loops, best of 3: 79.6 µs per loop
```

Trop fort! Nous battons tout le monde à plate couture! Nous sommes 100 fois plus rapides que numpy!!! ...et nous avons bien le bon résultat :

iPython

```
In [9]: (A*A)[100,100]
Out[9]: 200
```

Bon, ça va pour un seul produit. Pour calculer une puissance 200, ce sera moins magique...

1.2.7 Puissances d'une matrice

Par exemple, avec $J = \left(\begin{smallmatrix} 1 & 1 \\ 1 & 1 \end{smallmatrix}\right)$:

```
>>> J = Mat([2,2], lambda i,j : 1)
>>> J ** 5
 16  16
 16  16
```

En effet, $J^n = 2^{n-1}J$ est un résultat classique. On utilise un algorithme d'exponentiation rapide. Par exemple, en voici une version récursive rapide à écrire qui se nomme __pow__ afin d'utiliser le symbole ** :

matrice.py

```
    def __pow__(self,n):
        r = self.D[0]
        def pui(m,k,acc):
            if k == 0:
                return acc
            return pui((m*m),k//2,acc if k % 2 == 0 else (m*acc))
        return pui(self,n,unite(r))
```

Exercice 35. Déterminez une version impérative de la méthode précédente.

Solution.

```
def __pow__(self,n):
        r   = self.D[0]
        k   = n
        m   = self
        acc = unite(r)
        while k > 0:
            if k % 2 == 1:
                acc *= m
            m *= m
            k //= 2
        return acc
```

1.2.8 Trace d'une matrice

Tout est dans le titre. Nous vous rappelons que la trace d'une matrice est égale à la somme de ses éléments diagonaux.

matrice.py

```
def trace(self):
        r,c = self.D
        assert r == c, "La matrice doit être carrée"
        return sum(self[i,i] for i in range(r))
```

1.3 Inverse d'une matrice par la méthode de Gauss-Jordan

1.3.1 Opérations élémentaires

Pour effectuer une combinaison linéaire des lignes, on va créer une fonction :
`comb_ligne(ki,kj,M,i,j)`
qui renverra la matrice construite à partir de M en remplaçant la ligne L_j par $k_j \times L_j + k_i \times L_i$.

matrice.py

```
    def comb_lignes(self,ki,kj,i,j):
        """Li <- ki*Li + kj * Lj"""
        f = self.F
        g = lambda r,c : ki*f(i,c) + kj*f(j,c) if r == i else f(r,c)
        return Mat(self.D,g)
```

On crée également une fonction `mult_ligne(k,M,j)` : qui renverra la matrice construite à partir de M en remplaçant la ligne L_j par $k \times L_j$.

matrice.py

```
    def mult_ligne(self,k,i):
        """Li <- k*Li"""
        f = self.F
        g = lambda r,c : k*f(i,c) if r == i else f(r,c)
        return Mat(self.D,g)
```

1.3.2 Calcul de l'inverse étape par étape

On commence par créer une fonction qui renvoie une matrice où se juxtaposent la matrice initiale et la matrice identité :

matrice.py

```
    def cat_carre_droite(self):
        """
        Colle l'identité à droite pour la méthode de GJ

        """
        [lig, col], f = self.D, self.F
        g = lambda r,c: 1 if c == col + r else 0 if c >= col else f(r,c)
        return Mat([lig,2*col],g)
```

Créons une matrice :

```
>>> M = list2mat([[1,0,1],[0,1,1],[1,1,0]])
```

$$M = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

Complétons-la par la matrice identité de même taille :

```
>>> T = M.cat_carre_droite()
>>> T
 1  0  1  1  0  0
 0  1  1  0  1  0
 1  1  0  0  0  1
```

On effectue $L_3 \leftarrow L_3 - L_1$:

```
>>> T = T.comb_lignes(1,-1,2,0)
>>> T
 1  0  1  1  0  0
 0  1  1  0  1  0
 0  1 -1 -1  0  1
```

puis $L_3 \leftarrow L_3 - L_2$:

```
>>> T = T.comb_lignes(1,-1,2,1)
>>> T
 1  0  1  1  0  0
 0  1  1  0  1  0
 0  0 -2 -1 -1  1
```

puis $L_3 \leftarrow \frac{1}{2}L_3$:

```
>>> T = T.mult_ligne(-0.5,2)
>>> T
 1  0  1  1   0   0
 0  1  1  0   1   0
 0  0  1  0.5 0.5 -0.5
```

On effectue $L_1 \leftarrow L_1 - L_3$:

```
>>> T = T = T.comb_lignes(1,-1,1,2)
>>> T
 1  0  1  1   0   0
 0  1  0 -0.5 0.5 0.5
 0  0  1  0.5 0.5 -0.5
```

et enfin $L_2 \leftarrow L_2 - L_3$:

```
>>> T = T.comb_lignes(1,-1,0,2)
>>> T
   1    0    0  0.5 -0.5  0.5
   0    1    0 -0.5  0.5  0.5
  -0   -0    1  0.5  0.5 -0.5
```

Il ne reste plus qu'à extraire la moitié droite du tableau qui sera l'inverse cherchée à l'aide d'une petite fonction :

matrice.py

```
    def extrait_carre_droite(self):
```

```
        """
        Extrait le carré de droite d'un tableau de GJ
        """
        r,f = self.D[0],self.F
        return Mat([r,r], lambda i,j:f(i,j + r) )
```

alors :

```
>>> iM = T.extrait_carre_droite()
>>> iM
 0.5 -0.5  0.5
-0.5  0.5  0.5
 0.5  0.5 -0.5
```

On vérifie que c'est bien l'inverse de M :

```
>>> iM * M
   1    0    0
   0    1    0
   0    0    1
```

1.3.3 Réduction sous-diagonale d'une matrice

On pourrait calculer l'inverse d'une matrice en généralisant la méthode précédente mais cela s'avèrerait beaucoup trop gourmand en temps.
Nous allons dans un premier temps trigonaliser la matrice en effectuant des opérations élémentaires.
Pour éviter de faire trop de transformations, nous allons petit à petit réduire la taille de la matrice à trigonaliser en ne considérant que le carré inférieur droit situé sous le pivot courant et mémoriser chaque ligne obtenue dans une matrice.
Cela permet également de généraliser l'emploi de la méthode de GAUSS à des matrices non carrées pour la résolution de systèmes par exemple. Il faut donc faire attention maintenant à utiliser le minimum entre le nombre de lignes et le nombre de colonnes de la matrice.
On place des « mouchards » pour comprendre l'évolution du code.

matrice.py

```
    def triangle(self):
        """ renvoie la triangulation d'une matrice de haut en bas """
        [r,c] = self.D
        m     = min(r,c)
        S     = self
        T     = zeros(r,c)
        while m > 0:
            NoLigne = 0
            while S[NoLigne, 0] == 0 and (NoLigne < m - 1):
                    NoLigne += 1
            if S[NoLigne, 0] != 0:
                pivot = S[NoLigne,0]
                for k in range(1,m):
                    if S[k,0] != 0:
                        S = S.comb_lignes(pivot, -S[k,0],k,0)
                        print("pivot = "+str(pivot))
                        print("S dans for :")
                        print(S)
            T = T.remplace_ligned(r - m,S.F)
            print("Évolution de T :")
```

```
            print(T)
            S = S.dswap(NoLigne)
            m -= 1
        return T
```

Par exemple :

```
In [1]: M
Out[1]:
 1  2  3
 4  5  6
 7  8  8
In [2]: M.triangle()
pivot = 1
S dans for :
 1  2  3
 0 -3 -6
 7  8  8
pivot = 1
S dans for :
 1  2  3
 0 -3 -6
 0 -6 -13
```

```
Évolution de T :
 1  2  3
 0  0  0
 0  0  0
pivot = -3
S dans for :
-3 -6
 0  3
Évolution de T :
 1  2  3
 0 -3 -6
 0  0  0
Évolution de T :
 1  2  3
 0 -3 -6
 0  0  3
```

Nous avons besoin de deux fonctions intermédiaires, une remplissant le tableau T et l'autre réduisant S.

matrice.py

```
    def decoupe(self,i):
        """
        Fonction interne à triangle qui retire la 1ère ligne et la 1ère colonne

        """
        [lig, col], f = self.D, self.F
        return Mat([lig-1,col-1],lambda r,c : f(r+1,c+1))

    def remplace_ligned(self,i,g):
        """
        Fonction interne à triangle qui remplace dans la ligne i
        les coefficients à partir de la colonne i par ceux du tableau S
        """
        [lig, col], f = self.D, self.F
        h = lambda r,c: g(r-i,c-i) if r == i and c >= i else  f(r,c)
        return Mat([lig,col],h)
```

Exercice 36. Déterminez une méthode qui calcule rapidement le rang d'une matrice.

Solution.

```
def rang(self):
        r,c = self.D
        T = self.triangle()
        return len([T[i,i] for i in range(r) if T[i,i] != 0])
```

1.3.4 Calcul du déterminant

On triangularise la matrice et le déterminant est égal au produit des éléments diagonaux à un détail près : il ne faut pas oublier de tenir compte de la parité du nombre d'échanges de lignes ainsi que des multiplications des lignes modifiées par combinaisons.

matrice.py

```
    def det(self):
        """ renvoie le déterminant de self par Gauss-Jordan """
        [r,c] = self.D
        assert r == c, "Matrice non carrée"
        S,i,d = self,1,1
        while r > 0:
            NoLigne = 0
            while S[NoLigne, 0] == 0 and (NoLigne < r - 1):
                    NoLigne += 1
            if S[NoLigne, 0] == 0:
                return 0
            pivot = S[NoLigne,0]
            S = S.swap(0,NoLigne)
            i *= (-1) ** (NoLigne != 0)
            for k in range(1,r):
                if S[k,0] != 0:
                    S = S.comb_lignes(pivot, -S[k,0],k,0)
                    i *=  pivot
            S = S.decoupe()
            r -= 1
            d *= pivot
        return (d/i)
```

C'est assez efficace pour du Python basique : 555 ms pour un déterminant d'une matrice de taille 200.

```
In [1]: M = unite(200)
In [2]: %timeit M.det()
1 loops, best of 3: 555 ms per loop
```

1.3.5 Calcul de l'inverse d'une matrice

La trigonalisation est assez rapide du fait de la réduction progressive de S. L'idée est alors de trigonaliser de bas en haut puis de haut en bas pour arriver à une matrice diagonale ce qui sera plus efficace qu'un traitement de tout le tableau en continu.
On peut utiliser la fonction `det` qui est rapide et permet de ne pas prendre trop de précautions ensuite sachant que la matrice est inversible.

matrice.py

```
def inverse(self):
    return
        self.cat_carre_droite().triangle().diago_triangle().extrait_carre_droite()
```

Le tout est de construire cette méthode `diago_triangle` qui va trigonaliser le triangle en le parcourant de bas en haut.

matrice.py

```
      def diago_triangle(self):
        [r,c] = self.D
        assert c == 2*r, "Le tableau doit être un rectangle L x (2L)"
        m     = r - 1
        S     = self
        T     = zeros(r,c)
        while m >= 0:
            pivot = S[m,m]
            assert pivot !=0, "matrice non inversible"
            for k in range(m-1,-1,-1):
                if S[k,m] != 0:
                    S = S.comb_lignes(pivot, -S[k,m],k,m)
            T = T.remplace_ligneg(m,S.F)
            S = S.decoupe_bas()
            m -= 1
        for k in range(r):
            T = T.mult_ligne(1/T[k,r-1],k)
        return T
```

On transforme légèrement les fonctions intermédiaires précedentes pour les adapter au nouveau parcours :

matrice.py

```
    def decoupe_bas(self):
        """
        Fonction interne à diago_triangle qui retire la dernière ligne et la
        colonne de même numéro de S

        """
        [lig, col], f = self.D, self.F
        #g = lambda r,c: f(lig-1,c) if r == lig else f(i,c) if r == lig-1 else
            f(r,c)
        g = lambda r,c: f(r,c) if c < lig - 1 else f(r,c+1)
        return Mat([lig-1,col-1],lambda r,c : g(r,c))

    def remplace_ligneg(self,i,g):
        """
        Fonction interne à diago_triangle qui remplace dans la ligne i
        les coefficients à partir de la colonne i par ceux du tableau S
        """
        [lig, col], f = self.D, self.F
        h = lambda r,c: g(r,c - (lig - 1) + i) if r == i and c >= i else  f(r,c)
        return Mat([lig,col],h)
```

Exercice 37. (L'élimination Gaussienne mise en défaut) On considère la matrice

$$A = \begin{pmatrix} 10^{-20} & 0 & 1 \\ 1 & 10^{20} & 1 \\ 0 & 1 & -1 \end{pmatrix}$$

Quel est son rang ? Quel est son déterminant ? Utilisez les fonctions précédentes puis du papier et un crayon...

Solution.
Avec nos fonctions :

```
In [1]: M = list2mat([[1e-20,0,1],[1,1e20,1],[0,1,-1]])
```

```
In [2]: M.triangle()
Out[2]:
1E-20   0    1
   0    1   -1
   0    0    0
```

```
In [3]: M.rang()
Out[3]: 2

In [4]: M.det()
Out[4]: 0
```

Pourtant en effectuant la trigonalisation à la main on trouve : $\begin{pmatrix} 10^{-20} & 0 & 1 \\ 0 & -1 & 1-10^{-20} \\ 0 & 0 & -10^{-20} \end{pmatrix}$.
La matrice est donc de rang 3 et son déterminant vaut 10^{-40}.
Le problème :

```
In [5]: 1 - 1e-20 == 1
Out[5]: True
```

La représentation des flottants en précision standard sur 64 bits entraîne l'élimination de 10^{-20} devant 1 (voir 2.4 page 132).

1.4 Chiffrement de Hill

Nous allons à présent utiliser nos outils de calcul matriciel en cryptographie. En 1929, Lester HILL (1891 - 1961) proposa une méthode de chiffrement par blocs de lettres en travaillant avec des matrices à coefficients dans $\mathbb{Z}/26\mathbb{Z}$ (combien y a-t-il de lettres dans notre alphabet ?). Par la suite nous noterons $\mathbb{Z}_n$ l'ensemble $\mathbb{Z}/n\mathbb{Z}$. Cela a ouvert le domaine de la cryptographie algébrique et de nos jours, c'est l'AES (*Advanced Encrypting Standard*), né en 2000, qui assure la majorité de la sécurité informatique.
Nous allons explorer cette méthode en travaillant sur tous les caractères affichables en ASCII.

1.4.1 Codage

Chaque caractère est associé à un codage binaire, en général sur 8 bits (un octet) ce qui donne $2^8 = 256$ caractères possibles.
En fait, un américain a mis au point en 1961 l'ASCII (on prononce « aski ») pour « American Standard Code for Information Interchange » qui codait 128 caractères sur 7 bits ce qui est suffisant pour les américains qui n'ont pas de lettres accentuées.

Il a ensuite fallu étendre ce code pour les langues comportant des caractères spéciaux et là, les normes diffèrent, ce qui crée des problèmes de compatibilité.
Depuis quelques années, le codage *Unicode* tente de remédier à ce problème en codant tous les caractères existants.
La plupart des langages de programmation, et bien sûr **Python**, ont une commande qui associe le code ASCII (éventuellement étendu) à un caractère. On pourra donc l'utiliser pour résoudre notre problème. Il s'agit de la commande `ord(caratère)` qui renvoie le code ASCII du caractère `caractère`.

Caractère		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/	0	1	2
Code	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
Caractère	3	4	5	6	7	8	9	:	;	<	=	>	?	@	A	B	C	D	E
Code	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69
Caractère	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Code	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88
Caractère	Y	Z	[	\	]	^	_	‘	a	b	c	d	e	f	g	h	i	j	k
Code	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107
Caractère	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~
Code	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126

TABLE 3.1 – Table des caractères ASCII affichables

Par exemple :

```
>>> ord('@')
64
```

Mais attention ! Cela ne fonctionne par pour les chaînes de caractères :

```
>>> ord('AB')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: ord() expected a character, but string of length 2 found
```

Il faudra donc considérer les caractères d'une chaîne un par un :

```
def ascii(chaine):
    return [ord(c)-ord(' ') for c in chaine]
```

Le `-ord(' ')` permet d'obtenir des caractères codés entre 0 et 94.

```
>>> ascii("3h de retard !!#@!")
[19, 72, 0, 68, 69, 0, 82, 69, 84, 65, 82, 68, 0, 1, 1, 3, 32, 1]
```

Tant qu'on y est, créons une fonction `deascii` qui effectuera l'opération inverse. On utilisera alors la commande `chr` qui renvoie le caractère associé à un code ASCII. On utilise `join` pour concaténer plus vite des chaînes.

```
def deascii(chaine):
    return ''.join(chr(x + ord((' '))) for x in chaine)
```

Par exemple :

```
>>> deascii([19, 72, 0, 68, 69, 0, 82, 69, 84, 65, 82, 68, 0, 1, 1, 3, 32, 1])
'3h de retard !!#@!'
```

Choisissons à présent un message à coder, assez long :

Arghh !! Ce 1000 pattes aurait 314 pattes ??

```
>>> m = "Arghh !! Ce 1000 pattes aurait 314 pattes ??"
```

Le principe consiste alors à regrouper les nombres par paquets de longueur constante p et donc à construire à partir de la liste `L` une matrice de p lignes complétée éventuellement de zéros.
Nous allons pour cela créer une fonction `to_Hill(liste,p)` qui transforme une liste en une matrice de p colonnes en complétant par des zéros si nécessaire :

matrice.py

```
def to_Hill(m,p):
    cs = ascii(m)
    while len(cs) % p > 0 :
       cs += [0]
    g = lambda i,j: cs[(i % p) + p*j]
    return Mat([p,len(cs) // p], g)
```

Le message en clair devient une matrice. Choisissons par exemple un 3-chiffrement de Hill ; la matrice `clair` aura donc 3 lignes :

```
>>> clair = to_Hill(m,3)
>>> clair
 33  72   1  35  17  16  65  69  65  65   0  20  65  69  31
 82  72   1  69  16   0  84  83  85  73  19   0  84  83  31
 71   0   0   0  16  80  84   0  82  84  17  80  84   0   0
```

Il faut choisir à présent une clé de chiffrement : ce sera une matrice carrée de taille 3, à coefficients dans $\mathbb{Z}_{95}$ et inversible dans $\mathbb{Z}_{95}$. Nous allons vérifier que la matrice $\begin{pmatrix} 1 & 23 & 29 \\ 0 & 31 & 1 \\ 47 & 3 & 1 \end{pmatrix}$ convient.
Nous voudrions utiliser `inverse` mais on y effectue une division réelle ce qui n'est pas autorisée dans $\mathbb{Z}_{95}$. Nous allons donc un peu transformer `inverse` pour donner la méthode d'inversion des nombres comme argument. Seule la dernière ligne de `diago_triangle` change en fait :

matrice.py

```
    def diago_triangle(self,inv):
        .
        .
            T = T.mult_ligne(inv(T[k,r-1]),k)
        return T
```

puis :

matrice.py

```
def inverse(self,inv):
  return
      self.cat_carre_droite().triangle().diago_triangle(inv).extrait_carre_droite()
```

Oui mais quel est l'inverse d'un nombre dans $\mathbb{Z}_{95}$?
C'est un problème d'arithmétique qui se résout à l'aide de l'algorithme d'Euclide étendu :
Petit rappel sur cet algorithme qui permet de prouver le théorème de Bézout et d'obtenir des coefficients u et v vérifiant $au + bv = a \wedge b$:

k	u_k	v_k	r_k	q_k
0	1	0	$r_0 = a$	/
1	0	1	$r_1 = b$	q_1
2	$u_0 - u_1 q_1$	$v_0 - v_1 q_1$	$r_2 = r_0 - r_1 q_1$	q_2
3	$u_1 - u_2 q_2$	$v_1 - v_2 q_2$	$r_3 = r_1 - r_2 q_2$	q_3
$\vdots$			$\vdots$	$\vdots$
$p-1$			$r_{p-1} = a \wedge b$	q_{p-1}
p			$r_p = 0$	

La démonstration de l'algorithme découle de la relation :

$$u_k - (u_{k+1} \times q_{k+1}) = u_{k+2}$$

et de la relation similaire pour les v_k et les r_k.
Par exemple, pour 19 et 15 :

k	u_k	v_k	r_k	q_k	
0	1	0	19	/	L_0
1	0	1	15	1	L_1
2	1	−1	4	3	$L_2 \leftarrow L_0 - 1 \times L_1$
3	−3	4	3	1	$L_3 \leftarrow L_1 - 3 \times L_2$
4	4	−5	1	3	$L_4 \leftarrow L_2 - 1 \times L_3$
5			0		$L_5 \leftarrow L_3 - 3 \times L_4$

C'est-à-dire $4 \times 19 - 5 \times 15 = 1$ et $19 \wedge 15 = 1$.
On va utiliser les tableaux de `numpy` qui permettent d'effectuer des opérations vectorielles sur les listes :

matrice.py

```
import numpy as np

def bezout(a,b) :
    la  = np.array([1, 0, a])
    lb  = np.array([0, 1, b])
    while lb[2] != 0 :
        q      = la[2] // lb[2]
        la, lb = lb, la - q*lb
    return la
```

Soit $\dot{a}$ la classe de l'entier a modulo n. Démontrons que $\dot{a}$ est inversible dans $\mathbb{Z}_n$ si et seulement si a et n sont premiers entre eux :
$\dot{a} \times \dot{b} = \dot{1} \Leftrightarrow ab \equiv 1[n] \Leftrightarrow \exists k \in \mathbb{Z}\ ab = 1 + kn \Leftrightarrow \exists k \in \mathbb{Z}\ ba - kn = 1 \Leftrightarrow a \wedge n = 1$, la dernière équivalence étant obtenue grâce au théorème de Bézout.

Comme 11 et 13 sont premiers, chaque élément non nul de $\mathbb{Z}_{11}$ et $\mathbb{Z}_{13}$ est inversible. Mais par exemple 6 n'est pas premier avec 9, donc $\dot{6}$ n'est pas inversible dans $\mathbb{Z}_9$, c'est pourquoi la résolution de l'équation $6x \equiv b[9]$ pose problème.
Nous allons donc utiliser la fonction `bezout` créée précédemment pour déterminer l'inverse d'un nombre dans $\mathbb{Z}_n$:

matrice.py

```
def inv_mod(p,n):
    B = bezout(p,n)
    assert B[2] == 1, str(p) + ' non inversible modulo ' + str(n)
    return B[0] % n
```

On peut alors obtenir directement l'inverse de la matrice `cle` :

```
>>> cle = list2mat([[1,23,29],[0,31,1],[47,3,1]])
>>> inv_cle = cle.inverse(lambda x: inv_mod(x, 95))
>>> inv_cle
-1.82136E+09 -4.16311E+09  5.69826E+10
      23312      -675552         -496
     -30597        22638          651
```

Oups ! Il serait plus lisible d'obtenir les représentants canoniques de $\mathbb{Z}_{95}$. Il suffit donc de calculer le reste de chaque coefficient dans la division par 95 donc d'appliquer la fonction `lambda coeff : coeff % 95`.
Comme ce genre d'action est standard, on créer une méthode `map` dans notre classe `Mat` (on parle alors de *foncteur* en programmation fonctionnelle qui rejoint la notion homonyme en théorie des catégories).

matrice.py

```
    def map(self,f):
        return Mat(self.D, lambda i,j: f(self.F(i,j)))
```

Alors :

```
>>> inv_cle = cle.inverse(lambda x: inv_mod(x, 95)).map(lambda coeff : coeff % 95)
>>> inv_cle
 18  14  34
 37  88  74
 88  28  81
```

On vérifie :

```
>>> (cle * inv_cle).map(lambda coeff : coeff % 95)
 1  0  0
 0  1  0
 0  0  1
```

Nous pouvons multiplier `cle` par `clair` pour coder notre message. Pour le décoder, il suffira de multiplier le message codé par `inv_cle`.
La matrice `cle×clair` est une matrice de 3 lignes. Il faudra la transformer ensuite en liste pour lui appliquer `decode`. On utilise la fonction `iter` définie dans notre classe `Mat`.

```
>>> crypte = deascii(iter((cle * clair).map(lambda n : n % 95)))
```

```
>>> crypte
"sP_2Ou8?R'QOyE(Xph_<bn(h<Yc cAkDj\\pf_<bn(ho+>"
```

Si on le compare au message initial :
`"Arghh !! Ce 1000 pattes aurait 314 pattes ??"`
on remarque que le premier « pattes » est codé par `h_<be(` alors que le second est codé par `pf_<bn` : ce n'est donc pas un codage lettre par lettre et il est donc apparemment plus compliqué que le chiffrement de César.

1.4.2 Décodage

Celui qui reçoit ce message (l'éternel Bob...) possède la clé de décodage qui est `inv_cle`.

```
>>> decrypte = deascii(iter((inv_cle * (to_Hill(crypte,3))).map(lambda n: n%95)))
>>> decrypte
'Arghh !! Ce 1000 pattes aurait 314 pattes ?? '
```

Ça marche ! On remarque les deux espaces (neutres) rajoutés à la fin pour obtenir une chaîne de longueur un multiple de 3.

1.5 Chaînes de Markov

1.5.1 Qu'est-ce que c'est ?

Андрей Андреевич МАРКОВ (1856 - 1922), grand mathématicien russe, initiateur des probabilités modernes, grand amateur de poésie, n'hésita pourtant pas à s'impliquer dans la tumultueuse histoire de son pays. Il n'eut pas peur d'affirmer publiquement son anti-tsarisme et fut renvoyé de l'Université. Il fut réhabilité après la Révolution.
Nous allons illustrer de manière élémentaire quelques résultats de ses travaux sur les processus aléatoires (ou stochastiques), c'est-à-dire l'évolution dans le temps d'une variable aléatoire appelée *chaînes de* MARKOV dont les applications sont très nombreuses, notamment en informatique (reconnaissance de la langue d'un texte, techniques de compression, l'algorithme PageRank de Google, files d'attente, bioinformatique,...)
D'un point de vue plus probabiliste, les chaînes de MARKOV permettent d'étudier des variables aléatoires non indépendantes ce qui est très délicat à faire habituellement.
La particularité des chaînes de MARKOV est de représenter des processus où la probabilité permettant de passer à un instant donné d'un état présent à un état futur ne dépend du passé qu'à travers le présent soit : « la probabilité du futur sachant présent et passé est égale à la probabilité du futur sachant le présent ».
Nous allons présenter des exemples en utilisant des outils d'algèbre linéaire.

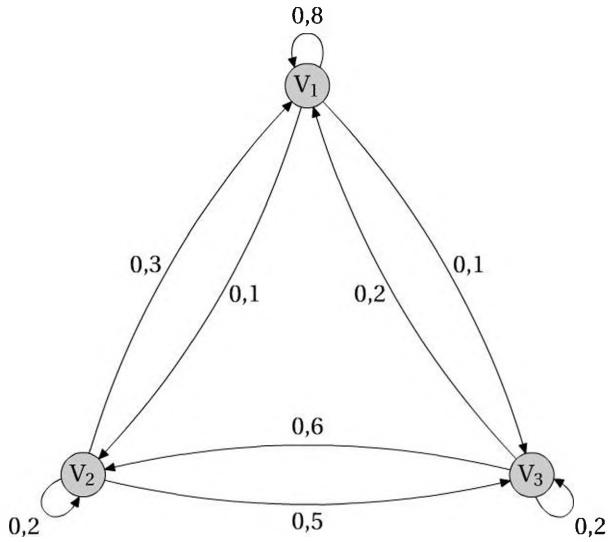
1.5.2 Markov et la mafia

Zlot, Brzxxz et Morzgniouf sont trois villes situées respectivement en Syldavie, Bordurie et Bouzoukstan. Des trafiquants de photos dédicacées du groupe ABBA prennent leur marchandise le matin dans n'importe laquelle de ces villes pour l'apporter le soir dans n'importe quelle autre. On notera pour simplifier V_1, V_2 et V_3 ces villes et p_{ij} la probabilité qu'une marchandise prise le matin dans la ville V_i soit rendue le soir dans la ville V_j. La matrice $P = (p_{ij})_{1\leqslant i\leqslant 3//1\leqslant j\leqslant 3}$ est appelée *matrice de transition* de la chaîne de MARKOV.

On s'attend donc que la somme des composantes de chaque vecteur colonne soit égal à 1.
Supposons que P soit connue et vaille :

$$P = \begin{pmatrix} 0,8 & 0,3 & 0,2 \\ 0,1 & 0,2 & 0,6 \\ 0,1 & 0,5 & 0,2 \end{pmatrix}$$

Les trafiquants se promenant de ville en ville, il peut être utile de visualiser leurs déplacements par le *diagramme de transition* suivant :



On notera $x_i^{(k)}$ la proportion de trafiquants qui se trouvent au matin du jour k dans la ville V_i.
En probabilités, on appelle *vecteur d'état* tout élément $(x_1, ..., x_n)$ de $\mathbb{R}^n$ tel que $x_1 + \cdots + x_n = 1$.
Ainsi, $x^{(k)} = \left(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}\right)$ est un vecteur d'état.
On montre que les vecteurs d'état de la chaîne sont liés par la relation :

$$\forall k \in [\![1, n]\!], \quad x^{(k)} = P \cdot x^{(k-1)}$$

et donc par une récurrence immédiate :

$$\forall k \in [\![1, n]\!], \quad x^{(k)} = P^k \cdot x^{(0)}$$

Supposons que le chef de la mafia locale dispose de 1000 trafiquants qui partent tous le matin du jour 0 de la ville de Zlot. Quelle sera la proportion de trafiquants dans chacune des villes au bout d'une semaine ? d'un an ?
Il s'agit donc de calculer des puissances successives de P. Nous allons utiliser les outils de la section précédente.
On obtient les proportions au bout d'une semaine en calculant $P^7 \cdot x^{(0)}$ avec $x^{(0)} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$:

```
>>> P = list2mat([[0.8,0.3,0.2],[0.1,0.2,0.6],[0.1,0.5,0.2]])
>>> X0 = Mat([3,1],lambda i,j : 1 if i == 0 else 0)
>>> P**7 * X0
           0.563887
           0.226069
           0.210044
```

i.e. des proportions respectives dans chacune des villes d'environ 56,4 %, 22,6 % et 21 %.
Au bout d'un mois, les proportions ne changent guère :

```
>>> P**30 * X0
          0.557377
          0.229508
          0.213115
```

On observe en fait que cette matrice « à l'infini » a trois colonnes identiques :

```
>>> P**30
          0.557377          0.557377          0.557377
          0.229508          0.229508          0.229508
          0.213115          0.213115          0.213115
```

On retrouve le vecteur précédent sur chaque colonne qui est en fait un vecteur propre pour la valeur propre 1 :

```
>>> Xs = P**30 * X0
>>> D = P*Xs - Xs
>>> D
           -2.79954E-09
            1.47778E-09
            1.32177E-09
```

1.5.3 Littérature aléatoire

On peut produire des textes (ou de la musique...) aléatoirement à l'aide de chaînes de Markov. Par exemple, connaissant un couple de lettres consécutives, on peut calculer la probabilité qu'il soit suivi par telle lettre en faisant un calcul de fréquence sur un très grand texte.
On va en profiter pour voir quelques outils de manipulation de texte, de récupération de textes en ligne.
On crée une classe pour être plus Pythonesque...

markov.py

```
from codecs import open
from urllib.request import urlretrieve
from random import choice
from itertools import chain


class Markov(object):

    def __init__(self,fichier):
        self.dicAssoc = {}
        self.fichier = fichier
        self.lettres = self.fichierVersLettres()
        self.baseDeTuplets()
        fichier.close()

    def fichierVersLettres(self):
        """ Renvoie les lettres du texte sous forme d'une liste """
```

```
        data = self.fichier.read() # le fichier entier de type string
        lettres = map(lambda lettre: ','.join(lettre),data) # on sépare chaque
            caractère du texte par une virgule
        return list(chain.from_iterable(lettres)) # on en fait une liste

    def tuplets(self):
        """ Crée un générateur de triplets de lettres successives """
        for i in range(len(self.lettres) - 2):
            yield (self.lettres[i], self.lettres[i+1], self.lettres[i+2])

    def baseDeTuplets(self):
        """Crée un dictionnaire dont les clés sont des couples de lettres
            successives et la valeur la liste des successeurs observés. """
        for l1,l2,l3 in self.tuplets():
            key = l1,l2
            if key in self.dicAssoc:
                self.dicAssoc[key].append(l3)
            else:
                self.dicAssoc[key] = [l3]

    def genereMarkovText(self, nbLettres = 100):
        """ Génère un texte selon la distribution du dictionnaire d'association
            """
        (l1,l2) = choice(list(self.dicAssoc.keys())) # on choisit un couple de
            lettres au hasard dans le dico
        gen_lettres = "" # on initialise le texte généré
        for i in range(nbLettres):
            gen_lettres += l1 # on écrit l1 dans letexte
            l1,l2 = l2,choice(self.dicAssoc[(l1,l2)]) # on avance d'un cran
        print(gen_lettres)
```

On peut récupérer des textes en ligne dont les droits sont libres :

```
# Notre-Dame de Paris
urlretrieve('http://www.gutenberg.org/cache/epub/19657/pg19657.txt','ndp.txt')
# Gone with the wind in english
urlretrieve('http://gutenberg.net.au/ebooks02/0200161.txt','autant.txt')
```

Les textes sont maintenant enregistrés au format txt dans le répertoire courant.
Il ne reste plus qu'à en faire des objets de type Markov :

```
ndp    = Markov(open('ndp.txt','r'))
autant = Markov(open('autant.txt','r'))
```

On peut alors générer du Victor HUGO aléatoire avec la commande :

```
>>> ndp.genereMarkovText(300)
```

> Ochez-votre proique fitu sais homme pemes tous de Mai sourgeaunes mit à tê-tandantres te, Domment de joul voint, priacur ? que de souvraîne Rymbots plus, per sée lir rour, étit l'égayeur. À nomme, ille périm Chât dest ret her, un dic s'étalla criait fles ble. Quas à saupi coeinit dirraldandre faistais

ou du Margarett MITCHELL aléatoire :

> Vot, Mamill befugh town sher to hanicand She mong for sestraid and he sair ton's whis As bey youts, ass welp gionto the carme : He depe ? Atled I die theard the wraing. Eurgo anked, streft te shou de up sware shand mor wharder lank.
>
> I wit. Hoplencithe ons. Don switay : Toddly yours. Dearry bee

On peut obtenir quelque chose de plus lisible en ne combinant plus les lettres mais les mots du texte.
Il suffit de faire deux légères modifications. D'une part la premiére fonction doit renvoyer la liste des mots ce que l'on peut obtenir avec la méthode `split` :

```
def fichierVersLettres(self):
        data = self.fichier.read()
        return data.split()
```

D'autre part, dans la dernière fonction, on introduit une espace entre les mots :

```
gen_lettres += ' ' + l1
```

Cela devient presque de la Littérature...

> pressées et les bourgeois de Paris jusqu'aux genoux.–Ceci s'est passé il n'y aurait point eu de l'argent, j'ai fait pour des bedeaux ! Et les escabeaux du recteur ! À bas ! reprit le compère Jacques se récria. « Tout de suite, Sire ! Le bailliage aura vingt fois l'épaisse porte sur laquelle l'ombre des chapiteaux saxons, d'autres montagnes que les vandales avaient produit, les académies l'ont tué. Aux siècles, aux révolutions qui dévastent du moins le reste de l'année d'échancrer de leurs jupes de dessous, plus précieuses encore que je vous dis que non ! » La bonne dame, fort entêtée de sa tête.

En anglais :

> friends came to town to see Mammy or Dilcey or even affection ? Or have I ever been before. Public feeling was back in no time. They are–right pretty, he conceded. I'll hang them for that, Pork. I've got to have a governess and I am annoyed at myself to death, and the women with broods of tow- haired silent children spent the evening at her own age. He always set a store by that niece of yours, isn't he ?

1.6 Manipulation d'images

Dans cette section, des mathématiques concrètes vont nous permettre de réduire, agrandir, assombrir, éclaircir, compresser, bruiter, quantifier,... une photo. Pour cela, il existe des méthodes provenant de la théorie du signal et des mathématiques continues. Nous nous pencherons plutôt sur des méthodes plus légères basées sur l'algèbre linéaire et l'analyse matricielle. Une image sera pour nous une matrice carrée de taille 2^9 à coefficients dans $[\![0, 2^9 - 1]\!]$. Cela manque de charme ? C'est sans compter sur Lena qui depuis quarante ans rend les maths sexy (la population hantant les laboratoires mathématiques et surtout informatiques est plutôt masculine...).

Nous étudierons pour cela la « Décomposition en Valeurs Singulières » qui apparaît de nos jours comme un couteau suisse des problèmes linéaires : en traitement de l'image et de tout signal en général, en reconnaissance des formes, en robotique, en statistique, en étude du langage naturel, en géologie, en météorologie, en dynamique des structures, etc.
Dans quel domaine travaille-t-on alors : algèbre linéaire, analyse, probabilités, topologie,... ? Un peu de tout cela et d'autres choses encore : cela s'appelle...La mathématique.

1.6.1 Lena

Nous allons travailler avec des images qui sont des matrices de niveaux de gris. Notre belle amie Léna sera représentée par une matrice carrée de taille 2^9 ce qui permet de reproduire Léna à l'aide de $2^{18} = 262\,144$ pixels. Léna prend alors beaucoup de place. Nous allons tenter de compresser la pauvre Léna sans pour cela qu'elle ne perde sa qualité graphique. Une des méthodes les plus abordables est d'utiliser la décomposition d'une matrice en valeurs singulières.
C'est un sujet extrêmement riche qui a de nombreuses applications. L'algorithme que nous utiliserons (mais que nous ne détaillerons pas) a été mis au point par deux très éminents chercheurs en 1965 (Gene GOLUB, états-unien et William KAHAN, canadien, père de la norme IEEE-754). Il s'agit donc de mathématiques assez récentes, au moins en comparaison avec votre programme...
La petite histoire dit que des chercheurs américains de l'University of Southern California étaient pressés de trouver une image de taille 2^{18} pixels pour leur conférence. Passe alors un de leurs collègues avec, en bon informaticien, le dernier Playboy sous le bras. Ils décidèrent alors d'utiliser le poster central de la Playmate comme support...
La photo originale est ici : http://www.lenna.org/full/len_full.html mais nous n'utiliserons que la partie scannée par les chercheurs, de taille 5.12in × 5.12in...

1.6.2 Environnement de travail

Nous travaillerons dans IPython avec l'option `pylab` qui charge les bibliothèques nécessaires à la manipulation d'images (en fait, pylab charge les bibliothèques pour se retrouver dans un environnement proche de celui de logiciels comme MatLab ou Scilab).

```
$ ipython --pylab
```

Les images sont alors sous forme d'`array` de la bibliothèque `numpy`.
Nous allons créer quelques outils pour continuer à travailler malgré tout avec notre classe `Mat`.

matrice.py

```
def array2mat(tab):
    """ convertit un array de numpy en objet de type Mat"""
    dim = list(tab.shape)
    return Mat(dim,  lambda i,j : tab[i,j])

def mat2array(mat):
    """ convertir un objet de type Mat en array de numpy """
    return np.fromfunction(mat.F,tuple(mat.D),dtype = int)

def montre(mat):
```

```
    """ permet de visualiser les images depuis un terminal """
    return imshow(mat2array(mat), cmap = cm.gray)
```

Python contient un tableau carré de taille 512 contenant les niveaux de gris représentant Lena. Il suffit de charger les bonnes bibliothèques :

```
from scipy import misc

# lena sous forme d'un objet de type Mat
matlena = array2mat(misc.lena())
```

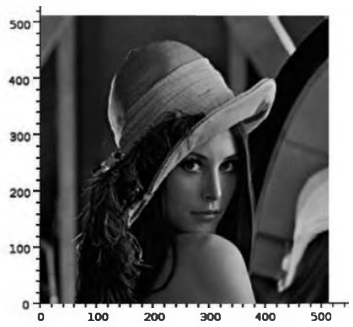
Lena est bien un carré de 512×512 pixels :

```
In [9]: matlena.D
Out[9]: [512, 512]
```

On peut voir Lena :

```
In [3]: montre(matlena)
Out[3]: <matplotlib.image.AxesImage at 0x7f06c6d0c7b8>
```

et on obtient :



On peut préférer travailler sur une photo de format jpg (qui sera à l'envers) ou png (qui sera à l'endroit). Python la transforme en matrice avec la fonction `imread`. La matrice obtenue est en RVB : chaque coefficient est un vecteur (R,V,B). On la convertit en niveau de gris par exemple en calculant la moyenne des trois couleurs.
On récupère d'abord une image :

```
In [4]: from urllib.request import urlretrieve
In [5]: urlretrieve('addresse de la photo en ligne','photo.png')
Out[5]: ('photo.png', <http.client.HTTPMessage at 0x7f3aeeb5fa20>)
```

et on la transforme en matrice :

matrice.py

```
lena_face_couleur = imread('lena_face.jpeg')

def rvb_to_gray(m):
    r,c = m.shape
    return(np.array([[sum(m[i,j]) / 3 for j in range(c)] for i in range(r)]))

lena_face_array = rvb_to_gray(lena_face_couleur)
lena_face       = array2mat(lena_face_array)
```

En exclusivité mondiale, voici Lena de face :

```
In [3]: montre(lena_face)
Out[3]: <matplotlib.image.AxesImage at 0x7f6d8c68cc88>
```

1.6.3 Manipulations basiques

Exercice 38. Comment obtenir les deux images de gauche sachant que l'image originale est à droite :

Solution.
Juste par de simples manipulations des coefficients :

matrice.py

```
def retourne(m):
    [r,c], f = m.D, m.F
    return Mat([r,c],lambda i,j: f(r - 1 - i, j))

def miroir(m):
    [r,c], f = m.D, m.F
    return Mat([r,c],lambda i,j: f(i,c - 1 - j))
```

Sachant que l'échelle de gris varie entre 0 et 255, comment obtenir les images suivantes :

On peut modifier le contraste en « mappant » par une fonction croissant plus rapidement ou plus lentement que la fonction identité sur [0;255] vers [0;255].
Par exemple, pour avoir une image plus foncée :

```
matlena.map(lambda x: int(x**3/255**2))
```

mais en fait Python normalise automatiquement donc ceci suffit :

```
matlena.map(lambda x: x**3)
```

et pour obtenir une image plus claire :

```
matlena.map(lambda x: x**.5)
```

Exercice 39. Comment obtenir l'image en négatif de Lena ?

Solution.
On prend le complémentaire par rapport à 255 :

```
matlena.map(lambda x: 255 - x)
```

1.6.4 Résolution

Une matrice de taille 2^9 contient 2^{18} entiers codés entre 0 et $2^8 - 1$ ce qui prend pas mal de place en mémoire. Peut-on être plus économique sans pour cela diminuer la qualité esthétique de la photo ?
La première idée est de prendre de plus gros pixels, c'est-à-dire une matrice plus petite : on extrait par exemple régulièrement un pixel sur k (en choisissant k parmi une puissance de 2 inférieure à 2^9).
Comment obtenir par exemple ces images ?



matrice.py

```
def resolution(k,mat):
    [r,c],f = mat.D,mat.F
    return Mat([r//k,c//k], lambda i,j : f(k*i,k*j))
```

Par exemple, pour réduire l'image à 128 × 128 de cette manière :

```
resolution(4,matlena)
```

1.6.5 Quantification

On peut également réduire le nombre de niveaux de gris en regroupant par exemple tous les niveaux entre 0 et 63 en un seul niveau 0, puis 64 à 127 en 64, 128 à 191 en 128, 192 à 256 en 192.
Par exemple, avec 4, 8 puis 16 niveaux :



Pour cela, on divise chaque coefficient de la matrice par une puissance de 2.

```
matlena.map(lambda x:  x // 2**4)
```

Mais une division d'un nombre par une puissance de 2 revient à faire un décalage de la chaîne de bits représentant ce nombre vers la droite Ce décalage vers la droite est effectué en Python par l'opérateur >>. L'image précedente peut donc être obtenue par :

```
matlena.map(lambda x: x >> 4)
```

1.6.6 Enlever le bruit

La transmission d'informations a toujours posé des problèmes : voleurs de grands chemins, poteaux télégraphiques sciés, attaques d'indiens, etc.
Claude Elwood SHANNON (1916 - 2001) est un mathématicien-inventeur-jongleur américain qui, suite à son article « *A mathematical theory of communications* » paru en 1948, est considéré comme le fondateur de la *théorie de l'information* qui est bien sûr une des bases de... l'*informatique*.
L'idée est d'étudier et de quantifier l'« information » émise et reçue : quelle est la compression maximale de données digitales ? Quel débit choisir pour transmettre un message dans un canal « bruité » ? Quel est le niveau de sûreté d'un chiffrement ?...
La théorie de l'information de SHANNON est fondée sur des modèles probabilistes : leur étude est donc un préalable à l'étude de problèmes de réseaux, d'intelligence artificielle, de systèmes complexes.
Par exemple, dans le schéma de communication présenté par SHANNON, la source et le destinataire d'une information étant séparés, des perturbations peuvent créer une différence entre le message émis et le message reçu. Ces perturbations (bruit de fond thermique ou accoustique, erreurs d'écriture ou de lecture, etc.) sont de nature *aléatoire* : il n'est pas possible de prévoir leur effet. De plus, le message source est par nature *imprévisible* du point de vue du destinataire (sinon, à quoi bon le transmettre).
SHANNON a également emprunté à la physique la notion d'entropie pour mesurer le désordre de cette transmission d'information, cette même notion d'entropie qui a inspiré notre héros national, Cédric VILLANI...
Mais revenons à Lena. Nous allons simuler une Lena bruitée. Pour cela, nous allons ajouter ajouter des pixels aléatoirement selon une loi de Bernoulli de paramètre p.

matrice.py

```
from numpy.random import randint
```

```
def ber(p):
    return 1 if random() < p else 0

def rand_image(dim,p):
    return Mat(dim, lambda i,j : randint(0,255)*ber(p))

def bruit(mat,p):
     return (mat + rand_image(mat.D,p))
```

Il existe de nombreuses méthodes, certaines très élaborées, permettant de minimiser ce bruit. Une des plus simples est de remplacer chaque pixel par la moyenne de lui-même et de ses 8 voisins directs, voire aussi ses 24 voisins directs et indirects, voire plus, ou de la médiane de ces séries de « cercles » concentriques de pixels.

matrice.py

```
def moyenne(mat,k):
    [li,co],f = mat.D, mat.F
    return Mat([li - k,co - k],
               lambda i,j : np.mean([f(r, c)
                                     for r in range(i - k, i + k + 1)
                                     for c in range(j - k, j + k + 1)]))
def mediane(mat,k):
    [li,co],f = mat.D, mat.F
    return Mat([li - k,co - k],
               lambda i,j : np.median([f(r, c)
                                       for r in range(i - k, i + k + 1)
                                       for c in range(j - k, j + k + 1)]))
```

Voici les résultats avec respectivement la moyenne sur des 9 pixels puis 25 pixels et à doite la médiane de carrés de 9 pixels. La médiane est plus efficace que la moyenne !

1.6.7 Compression par SVD

Un théorème démontré officiellement en 1936 par Carl ECKART et Gale YOUNG affirme alors que toute matrice rectangulaire A se décompose sous la forme :

$$A = U \times S \times {}^{t}V$$

avec U et V des matrices orthogonales et S une matrice nulle partout sauf sur sa diagonale principale qui contient les valeurs singulières de A rangées dans l'ordre décroissant. L'anglais

SYLVESTER s'y était déjà intéressé pour des matrices carrées réelles en 1889 mais ce résultat a semblé n'intéresser personne pendant des années.
Le court article de ECKART et YOUNG est disponible à l'adresse suivante :
http://projecteuclid.org/DPubS?service=UI&version=1.0&verb=Display&handle=euclid.bams/1183501633.
Ce qui est remarquable, c'est que n'importe quelle matrice admet une telle décomposition, alors que la décomposition en valeurs propres (la diagonalisation d'une matrice) n'est pas toujours possible. Et quand on dit n'importe quelle matrice, on ne rigole pas : toute matrice, même rectangulaire. C'est vraiment très puissant et à la fois simple mais cette décomposition n'a trouvé d'applications importantes qu'assez récemment ce qui explique peut-être qu'on en parle si peu en premier cycle.
Cependant, la décomposition en éléments singuliers utilise la décomposition en éléments propres.
Notons r le rang de A et U_i et V_i les vecteurs colonnes de U et V. La décomposition s'écrit :

$$A = \begin{pmatrix} U_1 & U_2 & \cdots & U_r & \cdots & U_m \end{pmatrix} \times \begin{pmatrix} \sigma_1 & & & & & \\ & \ddots & & & & \\ & & \sigma_r & & & \\ & & & \ddots & & \\ & & & & 0 & \end{pmatrix} \times \begin{pmatrix} {}^tV_1 \\ \vdots \\ {}^tV_r \\ \vdots \\ {}^tV_n \end{pmatrix}$$

ou formulé autrement :

$$\begin{aligned} A &= \sigma_1 \cdot U_1 \times {}^tV_1 + \sigma_2 \cdot U_2 \times {}^tV_2 + \cdots + \sigma_r \cdot U_r \times {}^tV_r + 0 \cdot U_{r+1} \times {}^tV_{r+1} + \cdots \\ &= \sigma_1 \cdot U_1 \times {}^tV_1 + \sigma_2 \cdot U_2 \times {}^tV_2 + \cdots + \sigma_r \cdot U_r \times {}^tV_r \end{aligned}$$

Il faut ensuite se souvenir que les σ_i sont classés dans l'ordre décroissant ce qui va avoir des applications importantes en informatique.
Dans de nombreux domaines, on doit travailler avec de grosses matrices. Il est alors très important de déterminer une matrice $\widetilde{A_s}$ de même taille qu'une matrice A mais de rang s inférieur au rang r de A et qui minimise l'erreur commise pour une certaine norme (qui est le plus souvent la norme de FROBENIUS, i.e. la racine carrée de la somme des carrés des coefficients).
Le théorème d'ECKART-YOUNG permet alors de dire que :

$$\min_{\mathrm{rg}(X) \leqslant s} \|A - X\|_F = \|A - \widetilde{A_s}\|_F = \sqrt{\sum_{j=s+1}^{r} \sigma_j^2(A)}$$

avec $\widetilde{A_s} = \sigma_1 \cdot U_1 \times {}^tV_1 + \sigma_2 \cdot U_2 \times {}^tV_2 + \cdots + \sigma_s \cdot U_s \times {}^tV_s$: on considère que les éléments diagonaux de A sont nuls en-dessous d'un certain seuil. Nous allons l'illustrer informatiquement à défaut de le démonter. On voit tout de suite des conséquences pratiques, que nous allons également illustrer informatiquement dans le cas de la compression des images.
La matrice donc image peut être décomposée sous la forme :

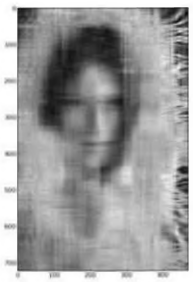
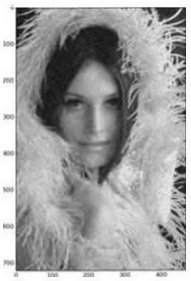
$$A = \sigma_1 \cdot U_1 \times {}^tV_1 + \sigma_2 \cdot U_2 \times {}^tV_2 + \cdots + \sigma_r \cdot U_r \times {}^tV_r$$

avec les σi de plus en plus petits. On peut donc approximer A en ne gardant que les premiers termes de la somme sachant que les derniers sont multipliés par des « petits » σ_i.

Le problème est d'obtenir cette décomposition. Python heureusement s'en charge grâce à la fonction `linalg.svd(mat)` qui renvoie la décomposition en valeurs singulières sous la forme de trois « array » de `numpy` U, S et ${}^{t}V$.
Par souci d'efficacité, nous travaillerons ici uniquement avec les « array » de `numpy`.

```
def c_svd(mat,k):
    U,s,tV = linalg.svd(mat)
    d = mat.shape
    C = np.zeros(d)
    for p in range(k):
        C += s[p]*(np.dot(U[:,p].reshape((d[0], 1)),tV[p,:].reshape((1,d[1]))))
    return C
```

Voici les trois niveaux de compression 10, 50 et 100 :

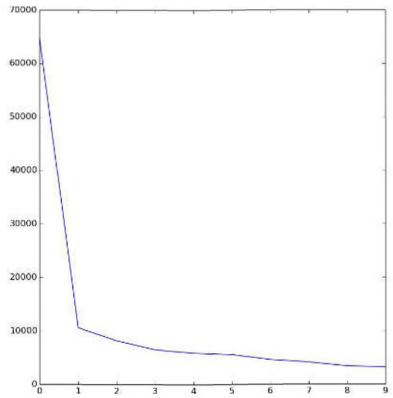



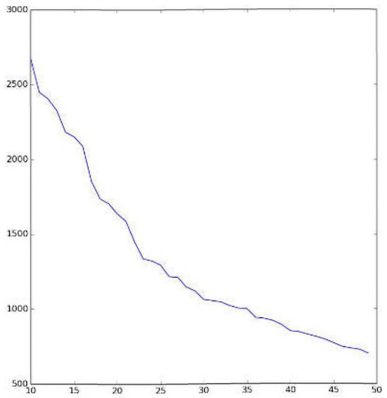

obtenus par :

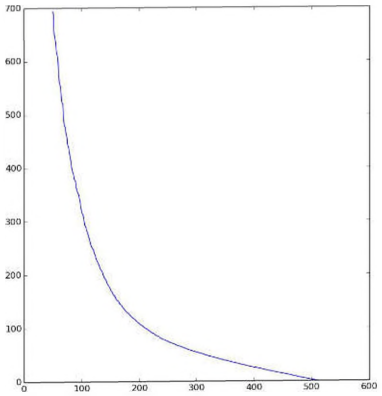
```
In [144]: imshow(c_svd(lena_face_array,100),cmap = cm.gray)
```

On peut observer la décroissance extrêmement rapide des valeurs singulières :

```
In [1]: U,s,tV = linalg.svd(misc.lena())
In [2]: x      = np.arange(len(s))
In [3]: plot(x[:10],s[:10])
In [4]: plot(x[10:50],s[10:50])
In [5]: plot(x[50:],s[50:])
```







On peut également faire une petite animation des images compressées :

matrice.py

```
def anim(im,n):
    for k in range(n):
        plt.imsave("lena_face_svd" + str(1000 + k), c_svd(lena_face_array,k),
            cmap = cm.pink)
```

Puis ensuite dans un terminal après installation de `imagemagick` :

matrice.py

```
import os

anim(150)
os.system("animate lena_face_svd*.png")
os.system("rm lena_face_svd*.png")
```

On obtient l'image animée dont on peut régler la vitesse avec les touches < et >. Une copie au format mpg est fournie dans l'archive.

1.7 Exercices d'entraînement

Les corrigés sont disponibles sur le site dunod.com à partir de la page d'accueil de l'ouvrage.

Exercice 40. (Algorithme de Strassen) En 1969 (*« Gaussian elimination is not optimal » in Numerische mathematik*), Konrad STRASSEN propose un algorithme pour calculer le produit de deux matrices carrées de taille n.
Le calcul habituel, pour le produit de deux matrices de taille 2 nécessite 8 multiplications et 4 additions.
Considérons à présent le produit de deux matrices de taille 2×2 :

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \times \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} w & x \\ y & z \end{pmatrix}$$

On note :

$$\begin{cases} p_1 = a(f-h),\ p_2 = (a+b)h,\ p_3 = (c+d)e,\ p_4 = d(g-e), \\ p_5 = (a+d)(e+h),\ p_6 = (b-d)(g+h),\ p_7 = (c-a)(e+f) \end{cases}$$

alors :

$$\begin{cases} w = p_4 + p_5 + p_6 - p_2 \\ x = p_1 + p_2 \\ y = p_3 + p_4 \\ z = p_1 + p_5 - p_3 + p_7 \end{cases}$$

Comparez le nombre d'opérations effectuées.
Considérez maintenant des matrices carrées de taille 2^t (si ce n'est pas le cas, que peut-on faire ?).

On peut découper nos matrices en quatre blocs de taille 2^{t-1} et appliquer récursivement notre algorithme : il faut quand même vérifier quelque chose pour passer des formules sur les coefficients aux formules sur les matrices : quoi donc ?
Comptez le nombre de multiplications et d'additions nécessaires.
Créez une fonction Python qui implémente cet algorithme.

Exercice 41. (Détection des bords) Pour sélectionner des objets sur une image, on peut essayer de détecter leurs bords, i.e. des zones de brusque changement de niveaux de gris.
On remplace alors un pixel par une mesure des écarts des voisins immédiats donnée par exemple par :

$$\sqrt{(m_{i,j+1} - m_{i,j-1})^2 + (m_{i+1,j} - m_{i-1,j})^2}$$

Écrivez une fonction qui trace le contour défini ci-dessus.

2 Les nombres : entre analyse et algèbre

2.1 Manipulation des entiers comme chaînes de bits

Nous allons essayer de nous rapprocher du cœur de la machine (autant que faire ce peut avec Python...) en définissant les opérations arithmétiques à partir de listes de bits.

2.1.1 Multiplication et complexité

Rappelons d'abord quelques définitions de notions largement utilisées en algorithmique.

Définition 3.1 (« Grand O »). Soit f et g deux fonctions de $\mathbb{N}$ dans $\mathbb{R}$. On dit que f est un « grand O » de g et on note $f = O(g)$ ou $f(n) = O(g(n))$ si, et seulement si, il existe une constante strictement positive C telle que $|f(n)| \leqslant C|g(n)|$ pour tout $n \in \mathbb{N}$.

Définition 3.2 (« Grand Oméga »). Soit f et g deux fonctions de $\mathbb{R}$ dans lui-même. On dit que f est un « grand Oméga » de g et on note $f = \Omega(g)$ ou $f(n) = \Omega(g(n))$ si, et seulement si, il existe une constante strictement positive C telle que $|f(n)| \geqslant C|g(n)|$ pour tout $n \in \mathbb{N}^*$.

Définition 3.3 (« Grand Théta »). $f = \Theta(g) \iff f = O(g) \ \wedge \ f = \Omega(g)$

Ce qui nous intéressera, c'est en fait le taux de croissance du temps d'exécution, c'est pourquoi ces notations sont très pratiques en algorithmique car elle font abstraction des temps d'accès mémoire, de la cadence du processeur, etc. qui dépendent de chaque machine et également des langages utilisés.
Vous savez additionner deux entiers de n chiffres : cela nécessite $\Theta(n)$ additions de nombres de un chiffre compte-tenu des re-tenues.
Multiplier un entier de n chiffres par un entier de 1 chiffre prend $\Theta(n)$ unités de temps selon le même principe.
En utilisant l'algorithme de l'école primaire pour multiplier deux nombres de n chiffres, on effectue n multiplications d'un nombre de n chiffres par un nombre de 1 chiffre puis une addition des n nombres obtenus. On obtient un temps de calcul en $\Theta(n^2)$.
On peut espérer faire mieux en appliquant la méthode *diviser pour régner.*

On coupe par exemple chaque nombre en deux parties de $m = \lfloor n/2 \rfloor$ chiffres :

$$xy = (10^m x_1 + x_2)(10^m y_1 + y_2) = 10^{2m} x_1 y_1 + 10^m (x_2 y_1 + x_1 y_2) + x_2 y_2$$

Fonction MUL(x: *e*ntier ,y: entier) : *entier*
Si n==1 **Alors**
  Retourner x.y
Sinon
  $m \leftarrow \lfloor n/2 \rfloor$
  $x_1, x_2 \leftarrow \lfloor x/10^m \rfloor, x \bmod 10^m$
  $y_1, y_2 \leftarrow \lfloor y/10^m \rfloor, y \bmod 10^m$
  $a, b \leftarrow$ MUL(x_1,y_1,m), MUL(x_2,y_1,m)
  $c, d \leftarrow$ MUL(x_1,y_2,m), MUL(x_2,y_2,m)
  Retourner $10^{2m} a + 10^m (b + c) + d$
FinSi

Les divisions et les multiplications par des puissances de 10 ne sont que des décalages effectués en temps constant.
L'addition finale est en $\Theta(n)$ donc le temps d'exécution est défini par :

$$T(n) = 4T(\lfloor n/2 \rfloor) + \Theta(n) \qquad T(1) = 1$$

Peut-on exprimez $T(n)$ explicitement en fonction de n sans récursion ?
Si nous avions une idée de la solution, nous pourrions la démontrer par récurrence.
Cherchons une idée donc... Ça serait plus simple si n était une puissance de 2. Voyons, posons $n = 2^k$ et $T(n) = T(2^k) = x_k$.
Alors la relation de récurrence devient :

$$x_k = 4x_{k-1} + \Theta(2^k) \qquad x_0 = 1$$

On obtient :

$$x_k = 4(4x_{k-2} + \Theta(2^{k-1})) + \Theta(2^k) = 4^k x_0 + \sum_{i=1}^{k} \Theta(2^k) = 4^k + k\Theta(2^k) = n^2 + \log n \Theta(n) = \Theta(n^2)$$

On montre alors par récurrence forte que ce résultat est vrai pour tout entier naturel non nul.
Bref, tout ça pour montrer que l'on n'a pas amélioré la situation...
Le grand Андрей Николаевич КОЛМОГОРОВ finit même par conjecturer dans les années 1950 que l'on ne pourra jamais trouver un algorithme de multiplication d'entiers en $o(n^2)$.
Lors d'un séminaire sur ce sujet en 1960, un jeune étudiant soviétique, Анатолий Алексеевич КАРАЦУБА propose cependant au Maître une solution plus simple et navrante de simplicité.
Il fait remarquer à КОЛМОГОРОВ que :

$$bc + ad = ac + bd - (a - b)(c - d)$$

Exercice 42. En quoi cela simplifie le problème ? Quelle est alors la nouvelle complexité ?

Solution. En ne considérant que les multiplications, on a maintenant 3 multiplications de nombres deux fois plus petits. On peut donc reprendre les calculs précédents et obtenir

$$x_k = 3^k + k\Theta(2^k) = 2^{k\log_2 3} + \log_2 n\Theta(n) = n^{\log_2 3} + \Theta(n\log_2 n) = \Theta(n^{\log_2 3})$$

ce qui est plus intéressant car $\log_2 3 \approx 1,58$

2.1.2 Classe de chaînes de bits

Nous allons manipuler les entiers signés comme des listes de bits de manière simple.
Par exemple, 13 sera représenté par `Bits([0,1,1,0,1])`, le premier bit représentant le signe : 0 pour + et 1 pour −.

Bits.py

```
class Bits(list) :
    """
    Crée des objets pour représenter des chaînes de bits correspondant à des
        entiers signés à partir de listes de 0 et 1.
    Par exemple 13 sera Bits([0,1,1,0,1])
    """

    def __init__(self,bits = []) :
        """ Une chaine de bits est construite à partir de la liste de ses bits"""
        self.bits = bits

    def __iter__(self) :
        """ On itère sur la liste des bits"""
        return iter(self.bits)

    def __len__(self) :
        """ Permet de donner le nombre de bits avec la méthode len"""
        return len(self.bits)

    def __getitem__(self,cle) :
        """ Pour obtenir le bit en position cle avec b[cle] ou une
            séquence de bits avec b[deb..fin]"""
        return self.bits[cle]

    def __reversed__(self) :
        """Pour renvoyer la chaîne de bits inversée"""
        return Bits(self.bits[::-1])

    def norm(self) :
        """Normalise l'écriture d'une chaîne de bits en supprimant les
        bits non significatifs à gauche"""
        if len(self) == 0:
            return self
        if len(self) == 1:
```

```
            return Bits(self.bits + self.bits)
        tete = self[0]
        qs   = self[1:]
        if qs[0] == tete:
            return Bits(qs).norm()
        return self

    def __str__(self) :
        """On utilise la fonction str pour afficher la liste des bits"""
        return str(self.bits)

    def __repr__(self) :
        """Une belle représentation dans l'interpréteur
           Bits([0,1,1,0,1]) est affiché 0:1101"""
        n = self.norm()
        return str(n[0]) + ':' + ''.join(str(b) for b in n.bits[1:])
```

2.1.3 Somme de deux chaînes de bits

On voudrait additionner facilement ces nombres. Une idée serait d'utiliser l'addition posée comme à l'école primaire. Par exemple pour calculer 5 + 9

```
    00101
+   01001
------------
    01110
```

On trouve bien 14.
Il faut faire attention à reporter la retenue.
Que se passe-t-il quand on additionne 3 bits ?

b_1	b_2	b_3	retenue	unité
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

On remarque que `u = b1 ^ (b2 ^ b3)` avec `^` l'opérateur du OU EXCLUSIF.
Quant à la retenue, elle vaut 1 si au moins deux des bits sont à 1. On va donc utiliser les opérateurs logiques ET et OU qui s'écrivent `&` et `|` en Python.

Bits.py

```
def __add__(self, other) :
        """
```

```
        Additionne deux entiers en base 2 sous forme de la liste de leurs bits
        """
        # Formate les listes pour qu'elles soient de même longueur
        long = max(len(self), len(other)) + 1
        s1,s2 = self.signe(), other.signe()
        op1 = Bits([s1 for i in range(len(self), long)] + self.bits)
        op2 = Bits([s2 for i in range(len(other), long)] + other.bits)
        # stocke la retenue de la précédente somme de 2 bits
        retenue = 0
        # On stocke la somme intermédiaire
        res = [];
        # on itère dans l'ordre inverse de la liste des bits des opérandes
        for i in range(1, long) :
            b1,b2 = op1[-i], op2[-i]
            # unité comme xor des deux bits et de la retenue
            res = [b1 ^ b2 ^ retenue] + res
            # retenue : vaut 1 si au moins 2 bits sont à 1
            retenue = (retenue & (b1 | b2)) | (b1 & b2)
        # on ajoute l'éventuelle dernière retenue à droite
        if s1 != s2:
            res = [retenue ^ s1 ^ s2] + res
        else:
            if retenue != 0:
                res = [s1, retenue] + res
            else:
                res = [s1] + res
        # on remet la liste dans le bon sens
        return Bits(res).norm()
```

Bon, essayons avec des nombres négatifs. Comment les représenter ? L'idée naïve est de mettre un 1 en premier puis de garder la même « valeur absolue ». Par exemple pour $5 + (-9)$:

```
    00101
+   11001
------------
    11110
```

Mais on trouve $5 + (-9) = -14$! Il faut trouver une autre idée sinon il faudrait créer une addition pour les positifs et une autre pour les nombres de signes différents ce qui ralentirait énormément le traitement des nombres sur machine.
Considérons `n=11110` et calculons $n+2$.

```
    11110
+   00010
------------
 (1)00000
```

Si l'on ne garde que le nombre de bits correspondant au plus long nombre, on obtient 0 (sur machine, on travaille sur 64 bits par exemple et tout ce qui dépasse cette taille n'est pas pris en compte).

On peut donc considérer que `11110` est l'opposé de `00010` en conservant le même algorithme d'addition.
En fait, ajouter `1` à `1111111` ou n'importe quelle autre chaîne uniquement composée de `1` donnera zéro.
Pour obtenir cette chaîne uniquement composée de `1`, on ajoute au nombre initial son *complément à 1*, c'est-à-dire qu'on remplace chaque bit par son complémentaire.
Par exemple, le complément à `1` de `01001` est `10110`. En rajoutant `1` à ce nombre, on obtient donc l'opposé du nombre initial.

$$01001 \xrightarrow{\text{complément à 1}} 10110 \xrightarrow{+1} 10111$$

Ainsi -9 s'écrit `10111` et cette fois :

```
    00101
+   10111
------------
    11100
```

Pour savoir le nombre que cette chaîne représente, on effectue les transformations réciproques dans l'ordre inverse :

$$11100 \xrightarrow{-1} 11011 \xrightarrow{\text{complément à 1}} 00100$$

On trouve 4 : en effet, $5 + (-9) = -4$!
Créons les méthodes associées :

Bits.py

```
    def map(self,fonc) :
        """Applique une fonction aux bits d'une chaîne"""
        return Bits(list(map(fonc,self.bits)))

    def compl(self) :
        """Renvoie le complément à 1 d'une chaîne"""
        return self.map(lambda x : 0 if x == 1 else 1)

    def __neg__(self) :
        """Opposé d'une chaîne en utilisant le symbole -"""
        bs = self.bits
        signe = bs[0]
        if signe == 1:
            return (self + Bits([1 for k in range(len(bs))])).compl()
        return self.compl() + Bits([0,1])

    def __sub__(self,other) :
        """Différence de deux chaînes avec -"""
        return self + (-other)

    def __abs__(self) :
        """Valeur absolue d'une chaîne"""
        return -self if self.signe() == 1 else self
```

2.1.4 Conversion en base 10

On rajoute quelques méthodes utiles :

Bits.py

```
    def mantisse(self) :
        """La liste des bits sans le bit de signe"""
        return self.norm()[1:]

    def __eq__(self,other) :
        """Permet de tester l'égalité de deux chaînes à partir de leur forme
            normalisée"""
        return self.norm().bits == other.norm().bits

    def signe(self) :
        """Renvoie le bit de signe"""
        return self[0]

    def pair(self) :
        """Teste la parité d'une chaîne"""
        return self[-1] == 0
```

On va maintenant passer de cette représentation en base 2 à la représentation en base 10 mais sans utiliser d'opérations arithmétiques.
Il existe sur tout langage des *opérations bits à bits* qui travaillent directement sur la représentation en base 2 des nombres. Nous avons déjà vu les opérateurs OU EXCLUSIF, ET, OU. Il existe également les opérateurs de *décalage* vers la gauche << ou la droite >> qui décale la chaîne de bits d'un certain nombres de bits d'un côté ou de l'autre. Ainsi on peut manipuler des entiers Python comme nos objets `Bits`.
Un décalage `<< n` correspondra donc à une multiplication par 2^n et un décalage `>> n` à une division par 2^n.

Bits.py

```
    def to_dec(self) :
        """Renvoie l'écriture décimale de la chaîne de bits"""
        ab = abs(self)
        cs = ab.mantisse()[::-1]
        if cs == []:
            return 0
        n   = 0
        dec = 0
        for c in cs:
            n += int(c) << dec
            dec += 1
        return n if self.signe() == 0 else -n
```

Par exemple :

```
In [29]: treize = Bits([0,1,1,0,1])
In [30]: dix = Bits([0,1,0,1,0])
```

```
In [31]: treize
Out[31]: 0:1101
```

```
In [32]: treize.to_dec()
Out[32]: 13
In [33]: dix - treize
```

```
Out[33]: 1:01
In [34]: (dix - treize).to_dec()
Out[34]: -3
```

2.1.5 Multiplication de l'école primaire

Nous allons commencer par la multiplication telle qu'elle est en général enseignée dans nos écoles primaires.
On prend les chiffres un par un d'un nombre, on les multiplie par l'autre nombre et on ajoute les résultats obtenus en « décalant ».

Bits.py

```
    def mul_primaire(self,other) :
        """Multiplication avec l'algo appris à l'école primaire"""
        s1, s2 = self.signe(), other.signe()
        sig = s1 ^ s2
        op1, op2 = abs(self)[1:], abs(other)[1:]
        m = Bits([0,0])
        dec = len(op1) - 1 # le décalage
        for n1 in op1:
            m1 = []
            for n2 in op2:
                m1.append(n2 * n1)
            m = m + Bits([0] + m1 + [0 for k in range(dec)])
            dec -= 1 # on va de gauche à droite
        if sig == 1:
            return -m
        return m
```

On peut alors définir cette multiplication comme produit canonique :

Bits.py

```
def __mul__(self,other) :
        """Définit la multplication de deux chaîne avec l'algo de l'école
            primaire"""
        return self.mul_primaire(other)
```

Par exemple :

```
In [38]: dix * -treize
Out[38]: 1:01111110
```

```
In [39]: (dix * -treize).to_dec()
Out[39]: -130
```

2.2 Les nombres rationnels

Nous utiliserons dans cette section la classe `rationnel` introduite page 252.

2.2.1 Les fractions continues

– Vous connaissez l'algorithme suivant :

$$\begin{aligned} 172 &= 3\times 51+19 \\ 51 &= 2\times 19+13 \\ 19 &= 1\times 13+6 \\ 13 &= 2\times 6+1 \\ 6 &= 6\times 1+0 \end{aligned}$$

– On peut donc facilement compléter la suite d'égalité suivante :

$$\frac{172}{51}=3+\frac{19}{51}=3+\frac{1}{\frac{51}{19}}=3+\frac{1}{2+\frac{13}{19}}=\ldots$$

Pratique du développement

– Quand tous les numérateurs sont égaux à 1, on dit qu'on a développé $\frac{172}{51}$ en fraction continue, et pour simplifier l'écriture on note :

$$\frac{172}{51}=[3,2,\ldots]$$

– Par exemple, on peut développer $\frac{453}{54}$ en fraction continue.
– Dans l'autre sens on peut écrire $[2,5,4]$ sous la forme d'une fraction irréductible.

Définissons une version récursive du calcul du développement en fractions continues d'un rationnel positif.

```
def fcr(F) :
    def aux(n,d,r,L) :
        if r == 0:
            L.append(n // d)
            return L
        else:
            L.append(n // d)
            return aux(d,r,d % r,L)
    return aux(F.num(), F.den(), F.num() % F.den(), [])
```

Pour reprendre notre exemple :

```
In [170]: fcr(rationnel(172,51))
Out[170]: [3, 2, 1, 2, 6]
```

On peut démontrer qu'un nombre est rationnel si, et seulement si, il admet un développement en fractions continues fini.
Voici une version impérative, simple traduction de la version récursive :

```
def fci(F) :
    n,d,r = F.num(), F.den(), F.num() % F.den()
    L = []
    while r > 0 :
```

```
            L.append(n // d)
            n, d, r = d, r, d % r
        return L + [n // d]
```

Nous allons à présent voir comment effectuer l'opération inverse, c'est-à-dire comment obtenir une fraction irréductible à partir de la liste du développement en fraction continue. L'écriture récursive est très rapide :

```
def fc2f(L) :
    if len(L) == 1:
        return rationnel(L[0],1)
    return rationnel(L[0]) + rationnel(1) / fc2f(L[1:])
```

sachant que `L[1:]` renvoie la liste `L` tronquée de son premier terme.

Par exemple :

```
In [176]: fc2f([3,2,1,2,6])
Out[176]: 172 / 51
```

Curiosité :

```
In [177]: fc2f([0,3,2,1,2,6])
Out[177]: 51 / 172
```

On peut essayer de démontrer ce résultat...

2.2.2 Les réduites d'une fraction continue

Une fraction continue, finie ou non, peut être tronquée après son k^{e} quotient :

$$q_0 + \cfrac{1}{q1 + \cfrac{1}{q_2 + \cfrac{1}{\cdots + \cfrac{1}{q_k}}}}$$

On appelle ce nombre rationnel la k^{e} réduite de la fraction continue considérée. Comme c'est un rationnel, il s'écrit sous forme irreductible $\frac{N_k}{D_k}$.
Pour découvrir une éventuelle relation de récurrence, on peut observer les premières réduites :

$$\begin{array}{ll} N_0 = q_0 & D_0 = 1 \\ N_1 = q_0 q_1 + 1 & D_1 = q_1 \\ N_2 = q_0 q_1 q_2 + q_0 + q_2 = q_2 N_1 + N_0 & D_2 = q_1 q_2 + 1 = q_2 D_1 + D_0 \end{array}$$

On peut montrer alors par récurrence (résultat dû à John WALLIS en 1655) que :

$$\begin{array}{rcl} N_k & = & q_k N_{k-1} + N_{k-2} \\ D_k & = & q_k D_{k-1} + D_{k-2} \end{array}$$

avec $N_0 = q_0$, $D_0 = 1$, $N_1 = q_0 q_1 + 1$ et $D_1 = q_1$.
Appelons **Python** à la rescousse :

```
def reduite(k,Q) :
    if k == 0:
        return frac(Q[0],1)
    elif k == 1:
        return frac(Q[1]*Q[0]+1,Q[1])
    else:
```

```
        return frac(Q[k]*(reduite(k-1,Q)).num +
               (reduite(k-2,Q)).num,Q[k]*(reduite(k-1,Q)).den +
               (reduite(k-2,Q)).den)
```

2.3 Les nombres réels

2.3.1 Approximation d'irrationnels à l'aide de rationnels

Algorithme de HÉRON Le mathématicien HÉRON d'Alexandrie n'avait pas attendu NEWTON et le calcul différentiel pour trouver une méthode permettant de déterminer une approximation de la racine carrée d'un nombre entier positif puisqu'il a vécu seize siècles avant Sir Isaac. Si x_n est une approximation strictement positive par défaut de $\sqrt{a}$, alors a/x_n est une approximation par excès de $\sqrt{a}$ et vice-versa.
La moyenne arithmétique de ces deux approximations est $\frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$ et constitue une meilleure approximation que les deux précédentes.
On peut montrer c'est une approximation par excès (en développant $\left(x_n - \sqrt{a}\right)^2$ par exemple).
En voici deux versions, une récursive et une itérative :

```
def heron_rec(a,fo,n) :
        if n == 0:
                return fo
        return heron_rec(a,(fo + rationnel(a) / fo) / rationnel(2), n - 1)

def heron_it(a,fo,n) :
        app = fo
        for k in range(n) :
                app = (app + rationnel(a) / app) / rationnel(2)
        return app
```

Ce qui donne par exemple :

```
In [1]: heron_rec(2,rationnel(1),6)
Out[1]: 1572584048032918633353217 / 1111984844349868137938112

In [2]: heron_it(2,rationnel(1),6)
Out[2]: 1572584048032918633353217 / 1111984844349868137938112
```

soit $\sqrt{2} \approx \frac{1\,572\,584\,048\,032\,918\,633\,353\,217}{1\,111\,984\,844\,349\,868\,137\,938\,112}$ en six étapes en partant de $\sqrt{2} \approx 1$.

Est-ce que la suite des valeurs calculées par la boucle converge vers $\sqrt{2}$? Nous aurons besoin de deux théorèmes que nous admettrons et dont nous ne donnerons que des versions simplifiées.

Théorème 3.1 (Théorème de la limite monotone)**.** Toute suite croissante majorée (ou décroissante minorée) converge.

Un théorème fondamental est celui du point fixe. Il faudrait plutôt parler des théorèmes du point fixe car il en existe de très nombreux avatars qui portent les noms de mathématiciens renommés : BANACH, BOREL, BROUWER, KAKUTANI, KLEENE,... Celui qui nous intéresserait le plus en informatique est celui de KNASTER-TARSKI. On peut en trouver une présentation

claire dans [Dow10]. Ils donnent tous des conditions d'existence de points fixes (des solutions de $f(x) = x$ pour une certaine fonction). Nous nous contenterons de la version light vue au lycée.

Théorème 3.2 (Un théorème light du point fixe)**.** Soit I un intervalle fermé de $\mathbb{R}$, soit f une fonction de I vers I et soit (r_n) une suite d'éléments de I telle que $r_{n+1} = f(r_n)$ pour tout entier naturel n. SI (r_n) est convergente ALORS sa limite est UN point fixe de f appartenant à I.

Cela va nous aider à étudier notre suite définie par $r_{n+1} = f(r_n) = \frac{r_n + \frac{2}{r_n}}{2}$ et $r_0 = 1$. On peut alors démontrer par récurrence que

a) pour tout entier naturel non nul n, on a $r_n \geqslant \sqrt{2}$;
b) la suite est décroissante ;
c) $\sqrt{2}$ est l'unique point fixe positif de f.

On peut alors conclure que notre algorithme va nous permettre d'approcher $\sqrt{2}$.

Quelle est la vitesse de convergence ? Ici, cela se traduit par « combien de décimales obtient-t-on en plus à chaque itération ? ». Introduisons la notion d'ordre d'une suite :

Définition 3.4 (Ordre d'une suite - Constante asymptotique d'erreur)**.** Soit (r_n) une suite convergeant vers ℓ. S'il existe un entier $k > 0$ tel que :

$$\lim_{n \to +\infty} \frac{|r_{n+1} - \ell|}{|r_n - \ell|^k} = \mathrm{C}$$

avec $\mathrm{C} \neq 0$ et $\mathrm{C} \neq +\infty$ alors on dit que (r_n) est d'ordre k et que C est la constante asymptotique d'erreur.

Déterminons l'ordre et la constante asymptotique d'erreur de la suite de Babylone donnant une approximation de $\sqrt{2}$. On démontre que

$$\left|r_{n+1} - \sqrt{2}\right| = \left|\frac{(r_n - \sqrt{2})^2}{2r_n}\right|$$

Ainsi la suite est d'ordre 2 et sa constante asymptotique est $\frac{1}{2\sqrt{2}}$. Ici, on peut même avoir une idée de la vitesse sans étude asymptotique.
En effet, on a $r_n \geqslant 1$ donc

$$\left|r_{n+1} - \sqrt{2}\right| \leqslant \left|(r_n - \sqrt{2})^2\right|$$

Le nombre de décimales d'un nombre positif plus petit que 1 est $-\log_{10} x$. En posant $d_n = -\log_{10}|r_n - \sqrt{2}|$ on obtient donc que $d_{n+1} \geqslant 2d_n$: à chaque tour de boucle, on double au minimum le nombre de bonnes décimales.
La précision de la machine étant de 16 décimales si les nombres sont codés en binaires sur 64 bits, **il faut au maximum 6 itérations pour obtenir la précision maximale**.

2.3.2 Développement en fractions continues de nombres réels à partir de nombres rationnels

Nombres irrationnels On peut utiliser les approximations précédentes d'irrationnels pour approcher le développement en fractions continues d'un irrationnel.
Étudions par exemple $\sqrt{2}$:

```
In [1]: [fci(heron_it(2,rationnel(1),k)) for k in range(5)]
Out[1]:
[[1],
 [1, 2],
 [1, 2, 2, 2],
 [1, 2, 2, 2, 2, 2, 2, 2],
 [1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]]

In [2]: [fci(heron_it(3,rationnel(1),k)) for k in range(5)]
Out[2]:
[[1],
 [2],
 [1, 1, 3],
 [1, 1, 2, 1, 2, 1, 3],
 [1, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 3]]

In [3]: [fci(heron_it(5,rationnel(2),k)) for k in range(5)]
Out[3]:
[[2],
 [2, 4],
 [2, 4, 4, 4],
 [2, 4, 4, 4, 4, 4, 4, 4],
 [2, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4]]
```

Il semble que le développement en fractions continues d'une solution irrationnelle d'une équation du second degré soit périodique.
En fait, EULER démontrera la condition suffisante puis LAGRANGE la condition nécessaire.
Pour établir certaines démonstrations en cours avec les élèves, on peut se référer à l'article en ligne à cette adresse :
http://fr.wikipedia.org/wiki/Fraction_continue_d'un_nombre_quadratique
On peut toutefois, au niveau lycée, démontrer la forme des développements de $\sqrt{(n)}$ avec n un entier qui n'est pas un carré parfait.
Par exemple :

$$\begin{aligned}\sqrt{2} &= 1+\sqrt{2}-1\\ &= 1+\frac{1}{\sqrt{2}+1}\\ &= 1+\frac{1}{2+\sqrt{2}-1}\end{aligned}$$

Ainsi, $\sqrt{2}-1=\frac{1}{2+(\sqrt{2}-1)}=\frac{1}{2+\frac{1}{2+(\sqrt{2}-1)}}=\cdots$.

On en déduit que $\sqrt{2}-1=[0,2,2,2,2,2,2,\cdots]$ puis que $\sqrt{2}=[1,2,2,2,2,\cdots]$.
Pour le plaisir, regardons à quoi peut ressembler le début du développement en fractions continues du nombre d'or $\Phi=\frac{1+\sqrt{5}}{2}$, toujours avec nos outils rudimentaires :

```
In [1]: fci((rationnel(1) + heron_it(5,rationnel(2),3)) / rationnel(2))
Out[1]: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2]
```

En utilisant les réduites étudiées à la section précédente, on peut obtenir des approximations rationnelles de précision croissante. Par exemple, voici comment obtenir des approximations rationnelles de $\sqrt{2}$:

```
In [1]: [reduite(k,fci(heron_it(2,rationnel(1),5))) for k in range(8)]
Out[1]: [1, 3 / 2, 7 / 5, 17 / 12, 41 / 29, 99 / 70, 239 / 169, 577 / 408]
```

Une approximation rationnelle de e à 10^{-300} près Posons $e_n=1+\frac{1}{1!}+\frac{1}{2!}+\cdots+\frac{1}{n!}$
Alors on a aussi

$$e_n=1+1+\frac{1}{2}\left(1+\frac{1}{3}\left(1+\frac{1}{4}\left(\cdots\left(\frac{1}{n-1}\left(1+\frac{1}{n}\right)\right)\right)\right)\right)$$

Un exercice classique montre que $e-e_n<\frac{n+2}{n+1}\frac{1}{(n+1)!}$.
Ainsi, pour $n=167$, on obtiendra 300 bonnes décimales au moins de e.

```
def expor(n) :
      if n == 167:
            return rationnel(1)
      return rationnel(1) + expor(n+1) / rationnel(n)
```

Observons le début du développement en fractions continues de l'approximation rationnelle de e précédente :

```
In [1]: fci(expor(1))
Out[1]:
[2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10, 1, 1, 12, 1, 1, 14, 1, 1, 16, 1,
    1, 18, 1, 1, 20, 1, 1, 22, 1, 1, 24, 1, 1, 26, 1, 1, 28, 1, 1, 30, 1, 1, 32,
    1, 1, 34, 1, 1, 36, 1, 1, 38, 1, 1, 40, 1, 1, 42, 1, 1, 44, 1, 1, 46, 1, 1,
    48, 1, 1, 50, 1, 1, 52, 1, 1, 54, 1, 1, 56, 1, 1, 58, 1, 1, 60, 1, 1, 62, 1,
    1, 64, 1, 1, 66, 1, 1, 68, 1, 1, 70, 1, 1, 72, 1, 1, 74, 1, 1, 76, 1, 1, 78,
    1, 1, 80, 1, 1, 82, 1, 1, 84, 1, 1, 86, 1, 1, 88, 1, 1, 90, 1, 1, 92, 1, 1,
    94, 1, 1, 96, 1, 1, 98, 1, 1, 100, 1, 1, 102, 1, 1, 104, 1, 1, 106, 1, 1,
    108, 1, 1, 110, 1, 1, 112, 1, 1, 114, 1, 1, 116, 1, 1, 118, 1, 1, 120, 1, 1,
    122, 1, 1, 124, 1, 1, 126, 1, 1, 128, 1, 1, 130, 1, 1, 132, 1, 1, 134, 1, 1,
    136, 1, 1, 138, 1, 1, 140, 1, 1, 142, 1, 1, 144, 1, 1, 142, 1, 11, 1, 7,....
```

Et pour $\frac{e-1}{e+1}$:

```
In [2]: fci((expor(1) - rationnel(1)) / (expor(1) + rationnel(1)))
Out[2]:
[0, 2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62, 66, 70, 74, 78,
    82, 86, 90, 94, 98, 102, 106, 110, 114, 118, 122, 126, 130, 134, 138, 142,
    146, 150, 154, 158, 162, 166, 170, 174, 178, 182, 186, 190, 194, 198, 202,
    206, 210, 214, 218, 222, 226, 230, 234, 238, 242, 246, 250, 254, 258, 262,
    266, 270, 274, 278, 282, 286, 290, 286,....
```

Remarque. L'utilisation des développements en fractions continues peut être éclairant pour les élèves : il est difficile de différencier $\frac{1}{7}$, $\sqrt{2}$, e et π à l'aide de leurs approximations en écriture décimale (« ça ne finit jamais ?!?... ») alors que la distinction est bien marquée avec les fractions continues.

2.3.3 Méthode de Briggs pour le calcul des logarithmes

Depuis STIFFEL en 1544, on s'intéresse aux fonctions vérifiant $\ell(x \cdot y) = \ell(x) + \ell(y)$ afin de simplifier les calculs des astronomes et des navigateurs. Cependant, cela nécessite de disposer de tables *logarithmiques* ($\lambda o \gamma o \varsigma$: mot, relation, $\alpha \rho \iota \theta \mu o \varsigma$: nombre ; les logarithmes sont donc des relations entre les nombres avant d'être l'anagramme d'algorithmes...).
Pour plus de facilité, nous allons nous occuper du logarithme de base 10 que nous noterons log.
Sachant que $\log_{10}(10) = 1$ et que $\log_{10}(a^p) = p \cdot \log_{10}(a)$ pour tout rationnel p, on en déduit alors que $\log_{10}\left(\sqrt{10}\right) = \frac{1}{2}$, $\log_{10}\left(\sqrt{\sqrt{10}}\right) = \frac{1}{4}$, etc.
Nous pourrions dans un premier temps travailler avec nos rationnels et l'algorithme de HÉRON pour le calcul des racines.
Nous allons cependant, pour plus d'efficacité, travailler avec des décimaux comme les mathématiciens des XVI^e et XVII^e siècles...mais avec **Python**.
Nous pourrions utiliser l'algorithme de HÉRON pour calculer des approximations décimales des racines carrées.
Ensuite, on utilise l'algorithme de HÉRON pour avoir une approximation décimale de $\sqrt{2}$:

```
In [1]: r = heron_it(2,rationnel(1),6)
```

```
In [2]: float(r)
Out[2]: 1.414213562373095
```

Pour plus de lisibilité, nous utiliserons dorénavant le module `math` de **Python** qui permet d'obtenir directement une approximation décimale (avec 17 chiffres affichés) des racines carrées à l'aide de `sqrt` :

```
In [1]: from math import*
In [2]: sqrt(2)
Out[2]: 1.4142135623730951
In [3]: sqrt(10)
Out[3]: 3.1622776601683795
```

```
In [4]: sqrt(sqrt(10))
Out[4]: 1.7782794100389228
In [5]: sqrt(sqrt(sqrt(10)))
Out[5]: 1.333521432163324
```

On obtient donc le tableau suivant :

Nombres	10,0000	3,1623	1,7783	1,3335	1,0000
Logarithmes	1	0,5	0,25	0,125	0

Le problème, c'est que connaître le logarithme de 3,1623 n'est pas très intéressant. On aimerait plutôt connaître ceux des entiers de 1 à 10.
C'est ici qu'intervient l'anglais Henry BRIGGS qui publia en 1617 ses premières tables de logarithmes décimaux contenant 1 000 valeurs avec quatorze décimales.
Voyons comment il a procédé pour le calcul de log(2).
Il a commencé par obtenir des valeurs approchées des nombres suivant :

$$\begin{array}{ll} \sqrt{10} = 10^{\frac{1}{2}} & \sqrt{2} = 2^{\frac{1}{2}} \\ \sqrt{\sqrt{10}} = 10^{\frac{1}{2^2}} & \sqrt{\sqrt{2}} = 2^{\frac{1}{2^2}} \\ \sqrt{\sqrt{\sqrt{10}}} = 10^{\frac{1}{2^3}} & \sqrt{\sqrt{\sqrt{2}}} = 2^{\frac{1}{2^3}} \\ \vdots & \vdots \\ \sqrt{\sqrt{\cdots\sqrt{10}}} = 10^{\frac{1}{2^{54}}} & \sqrt{\sqrt{\cdots\sqrt{2}}} = 2^{\frac{1}{2^{54}}} \end{array}$$

Il a finalement abouti à :

$$10^{\frac{1}{2^{54}}} \approx \underbrace{1{,}000\,000\,000\,000\,000\,127\,819\,149\,320\,032\,35}_{1+a}$$

$$2^{\frac{1}{2^{54}}} \approx \underbrace{1{,}000\,000\,000\,000\,000\,038\,477\,397\,965\,583\,10}_{1+b}$$

Or on cherche à déterminer $x = \log(2)$ mais $x = \log(2) \Leftrightarrow 10^x = 2$.
D'après les calculs de BRIGGS :

$$1 + b = 2^{\frac{1}{2^{54}}} = \left(10^x\right)^{\frac{1}{2^{54}}} = (1 + a)^x$$

Un résultat classique montre que $(1 + a)^x \approx 1 + ax$ pour x assez proche de zéro. On en déduit que $1 + b \approx 1 + ax$ et donc que :

$$\log(2) = x \approx \frac{b}{a} = \frac{3\,847\,739\,796\,558\,310}{12\,781\,914\,932\,003\,235}$$

```
In [1]: 3847739796558310 / 12781914932003235
Out[1]: 0.30102999566398114
```

Ainsi, $\log(2) \approx 0{,}301\,029\,995\,663\,981\,14$.
Cela nous permet alors d'avoir également $\log(4) = 2 \cdots \log(2)$ et $\log(8) = 3 \times \log(2)$ mais aussi $\log(5) = \log(10) - \log(2)$.
Il nous faudrait donc obtenir de même $\log(3)$ et $\log(7)$ pour compléter notre collection car $\log(6) = \log(3) + \log(2)$ et $\log(9) = 2 \cdot \log(3)$.
On peut donc à titre d'exercice calculer ces deux logarithmes pour obtenir :

$$\log(3) \approx 0{,}477\,121\,254\,719\,662\,44 \quad \log(7) \approx 0{,}845\,098\,040\,014\,256\,7$$

La route est longue jusqu'aux 1 000 logarithmes de BRIGGS...

2.3.4 Méthode des différences de Newton et application au calcul des logarithmes

On connaît maintenant quelques valeurs du logarithme décimal et l'on voudrait connaître des valeurs intermédiaires. Nous allons pour cela procéder par interpolation : étant données quatre valeurs, nous allons chercher un polynôme P de degré 3 tel que $P(x) = \log(x)$ pour ces 4 valeurs.
Pour cela, nous allons utiliser la méthode présentée par Isaac NEWTON en 1676.
Avec des notations modernes, posons $P(x) = a + bx + cx^2 + dx^3$. On connaît les valeurs du logarithmes pour 1, 2, 3 et 4.

$x=1$	$p(x)=a+b+c+d=y_1$
$x=2$	$p(x)=a+2b+4c+8d=y_2$
$x=3$	$p(x)=a+3b+9c+27d=y_3$
$x=4$	$p(x)=a+4b+16c+64d=y_4$

En soustrayant les lignes deux par deux, on fait disparaître a :

$$\begin{array}{|l|}\hline y_2-y_1=b+3c+7d=\Delta y_1\\ y_3-y_2=b+5c+19d=\Delta y_2\\ y_4-y_3=b+7c+37d=\Delta y_3\\ \hline\end{array}$$

En soustrayant les lignes deux par deux, on fait disparaître b :

$$\begin{array}{|l|}\hline \Delta y_2-\Delta y_1=2c+12d=\Delta^2 y_1\\ \Delta y_3-\Delta y_2=2c+18d=\Delta^2 y_2\\ \hline\end{array}$$

En soustrayant les lignes deux par deux, on fait disparaître c :
$\Delta^2 y_2-\Delta^2 y_1=6d=\Delta^3 y_1$
ce qui donne :
$d=\frac{1}{6}\Delta^3 y_1$
$c=\frac{1}{2}\Delta^2 y_1-\Delta^3 y_1$
$b=\Delta y_1-\frac{3}{2}\Delta^2 y_1+3\Delta^3 y_1-\frac{7}{6}\Delta^3 y_1=\Delta y_1-\frac{3}{2}\Delta^2 y_1+\frac{11}{6}\Delta^3 y_1$
$a=y_1-\Delta y_1+\frac{3}{2}\Delta^2 y_1-\frac{11}{6}\Delta^3 y_1-\frac{1}{2}\Delta^2 y_1+\Delta^3 y_1-\frac{1}{6}\Delta^3 y_1=y_1-\Delta y_1+\Delta^2 y_1-\Delta^3 y_1$
Finalement :

$$\mathrm{P}(x)=y_1+(x-1)\Delta y_1+\frac{1}{2}(x-1)(x-2)\Delta^2 y_1+\frac{1}{6}(x-1)(x-2)(x-3)\Delta^3 y_1$$

NEWTON avait l'habitude de réunir les résultats dans un schéma de cette forme :

$$\begin{array}{lllll}\mathbf{y_1} & & & & \\ & \mathbf{\Delta y_1} & & & \\ y_2 & & \mathbf{\Delta^2 y_1} & & \\ & \Delta y_2 & & \mathbf{\Delta^3 y_1} & \\ y_3 & & \Delta^2 y_2 & & \\ & \Delta y_3 & & & \\ y_4 & & & & \end{array}$$

avec $\Delta y_i=y_{i+1}-y_i$, $\Delta^2 y_i=\Delta y_{i+1}-\Delta y_i$ et $\Delta^3 y_i=\Delta^2 y_{i+1}-\Delta^2 y_i$.
On peut généraliser à d'autres valeurs d'entiers successifs et à un nombre quelconque de pôles.
Interpolation de NEWTON passant par les points (α,y_1), $(\alpha+1,y_2)$,..., $(\alpha+n-1,y_n)$:

$$p(x)=y_1+(x-\alpha)\Delta y_1+\frac{1}{2!}(x-\alpha)(x-\alpha-1)\Delta^2 y_1+\cdots+\frac{1}{(n-1)!}(x-\alpha)\cdots(x-\alpha-(n-2))\Delta^{n-1}y_1$$

Pour des pôles quelconques, on pourra se référer à la section 4 page 199.
Et **Python** dans tout ça ? Une première méthode « à la main » :

newton.py

```
def diff_newt(x1,y1,y2,y3,y4,pas) :
    Dy1  = y2 - y1
    Dy2  = y3 - y2
    Dy3  = y4 - y3
    D2y1 = Dy2 - Dy1
    D2y2 = Dy3 - Dy2
    D3y1 = D2y2 - D2y1
    def P(x) :
        return y1 + (x - x1)*(Dy1 + (x - x1 - 1)*(D2y1/2 + (x - x1 - 2)*D3y1 /
            6))
    return [[(x1 + k*pas), P(x1 + k*pas)] for k in range(int(3 / pas))]
```

Voici une méthode plus générale :

newton.py

```
def reduit_liste(L) :
    """ Calcule la liste des différences 2 à 2 d'une liste de nbs """
    return [a - b for (a,b) in zip(L[1:], L)]

def reduit(L) :
    """ Renvoie la liste des listes de différences : le triangle de Newton
        applati """
    if L == [] :
        return L
    return [L] + reduit(reduit_liste(L))

def diff_newton(L) :
    """ Renvoie les y1, Dy1, D2y1, etc. """
    return [liste[0] for liste in reduit(L)]

def base_newton(alpha, x, n) :
    """ Renvoie la liste des évaluations des (x-a)(x-a-1).../n! accumulées dans
        une liste """
    terme = 1
    base  = [terme]
    for k in range(1,n) :
        terme *= ((x - alpha - (k - 1)) / k )
        base.append(terme)
    return base

def interpol_newton(liste, alpha, x) :
    """ renvoie l'évaluation du poly de Newton en x en partant de a avec une
        liste de valeurs donnée """
    return sum([ a * b for (a,b) in zip(diff_newton(liste), base_newton(alpha,
        x, len(liste))) ])

def liste_log_entiers(a,b) :
    """ renvoie la liste des logarithmes décimaux des entiers de a à b inclus """
    return [math.log10(k) for k in range(a, b + 1)]
```

Par exemple, pour avoir les logarithmes décimaux des réels entre 1 et 4 avec un pas de 0,25 en utilisant les valeurs déjà connues de log(2), log(3) et log(4) :

```
In [1]: [(k, interpol_newton(liste_log_entiers(1,4), 1, k)) for k in
    np.linspace(1,4,13)]
```

```
Out[1]: [(1.0, 0.0),
 (1.25, 0.09100568905994865),
 (1.5, 0.17074397829308552),
 (1.75, 0.24036777729567488),
 (2.0, 0.3010299956639812),
 (2.25, 0.35388354299426877),
 (2.5, 0.4000813288828019),
 (2.75, 0.44077626292584499),
 (3.0, 0.47712125471966244),
 (3.25, 0.51026921386051849),
 (3.5, 0.54137304994467772),
 (3.75, 0.5715856725684042),
 (4.0, 0.60205999132796229)]
```

Comme **Python** sait calculer des logarithmes, nous pouvons comparer cette méthode vieille de 350 ans avec les résultats actuels :
Regardons par exemple pour les logarithmes des nombres entre 44 et 47 :

```
In [2]: [(k, interpol_newton(liste_log_entiers(44,48), 44, k) - math.log10(k))
    for k in np.linspace(44,48,17)]
```

```
Out[2]: [(44.0, 0.0),
 (44.25, -1.4762764344311563e-09),
 (44.5, -1.4248640045622096e-09),
 (44.75, -7.4078276846023527e-10),
 (45.0, 0.0),
 (45.25, 4.8304182875824608e-10),
 (45.5, 5.9947979913488325e-10),
 (45.75, 3.9161784926022847e-10),
 (46.0, 0.0),
 (46.25, -3.8808622981889584e-10),
 (46.5, -5.8871596486653743e-10),
 (46.75, -4.7009018899757393e-10),
 (47.0, 0.0),
 (47.25, 7.0797234741348802e-10),
 (47.5, 1.3494656503354463e-09),
 (47.75, 1.3855363523163078e-09),
 (48.0, 0.0)]
```

L'erreur commise est de l'ordre de 10^{-9}.

2.3.5 Algorithme CORDIC

Avec l'apparition des premières calculateurs électroniques, les programmeurs devaient concilier précision et économie de mémoire. À partir de 1956, l'armée de l'air des États-Unis voulut améliorer le système de navigation embarqué des bombardiers B-58. C'est ainsi que Jack E. VOLDER mit au point l'algorithme CORDIC (pour Coordinate Rotation DIgital Computer) pour calculer rapidement des lignes trigonométriques. Il fut ensuite repris dans des domaines plus pacifiques, notamment dans la programmation des calculatrices scientifiques pour calculer non seulement des fonctions trigonométriques mais aussi des logarithmes comme le montra John WALTHER en 1971.

Nous aurons besoin de connaître quelques valeurs particulières de la fonction ln ; plus précisément des valeurs de $\ln\left(1+10^{-i}\right)$ pour des valeurs de i entières, telles que $1+10^{-i}$ reste compris entre 0 et 10.
Nous pouvons consulter les tables de BRIGGS. Pour aller plus vite, nous allons un peu tricher et utiliser **Python** pour calculer ces valeurs. Nous allons créer un *dictionnaire* qui à chaque valeur de i associe $\ln\left(1+10^{-i}\right)$ sauf pour $i=-1$ qui est associé à $\ln(10)$:

```
def dic_log(n) :
    dic = {x:log(1+10**(-x)) for x in range(n+1)}
    dic[-1] = log(10)
    return(dic)
```

Par exemple :

```
In [1]: from math import log
In [2]: D = dic_log(5)
In [3]: D
Out[3]: {0: 0.6931471805599453,
 1: 0.09531017980432493,
 2: 0.009950330853168092,
 3: 0.0009995003330834232,
 4: 9.999500033329732e-05,
 5: 9.999950000398841e-06,
-1: 2.30258509299404}
In [4]: D[2]
Out[4]: 0.009950330853168092
In [5]: D[-1]
Out[5]: 2.302585092994046
```

Ainsi, `D[2]` renvoie $\ln\left(1+10^{-2}\right)$.
On utilise ensuite le théorème suivant [pour une démonstration voir Bop07]) :

Théorème 3.3. Soit x un réel strictement compris entre 1 et 10. Il existe alors une suite d'entiers $(n_i)_{i\in\mathbb{N}}$ telle que :

a) $0 \leqslant n_i \leqslant 10$;

b) pour tout entier positif N :

$$x\left(\prod_{i=0}^{n}\left(1+10^{-i}\right)^{n_i}\right) \leqslant 10 \leqslant x\left(\prod_{i=0}^{n}\left(1+10^{-i}\right)^{n_i}\right)\left(1+10^{-\mathrm{N}}\right)$$

ce qui permet de montrer, en posant $y_n = \ln(10) - \sum_{i=0}^{\mathrm{N}} n_i \ln\left(1+10^{-i}\right)$:

Théorème 3.4. Pour tout entier positif N, il existe une suite d'entiers positifs $n_i \leqslant 10$ tels que y_{N} soit une valeur approchée de $\ln(x)$ à $10^{-\mathrm{N}}$ près.

Cela donne, avec **Python** :

```
def cordic(x,N) :
    D = dic_log(N)
    y = D[-1]
    for i in range(N+1) :
        a = 1 + 10**(-i)
        while x*a <= 10:
            x *= a
            y -= D[i]
    return y
```

Par exemple, comparons ce que donne `cordic` avec ce que le `log` de **Python** renvoie :

```
In [1]: cordic(3.1957,15)
Out[1]: 1.1618061561640456
```

```
In [2]: log(3.1957)
Out[2]: 1.1618061561640467
```

2.4 Réels et flottants : un très rapide survol de la norme IEEE754

2.4.1 Le constat

Le calcul de sommes est un favori de l'initiation à l'algo au lycée. Prenons la somme partielle de la série harmonique :

$$\sum_{k=1}^{k=n} \frac{1}{k}$$

Rien de plus simple à programmer :

```
def somme1(n) :
    S = 0
    for k in range(1,n+1) :
        S += 1.0 / k
    return S
```

On obtient :

```
In [1]: somme1(100000)
Out[1]: 12.090146129863335
```

Oui mais est-ce que c'est correct ? Allons voir du côté du logiciel **Sage** : c'est la même fonction qu'en **Python** mais on peut travailler avec des rationnels et ensuite demander une approximation avec la méthode n(taille de la mantisse).

Sage

```
sage: def somme1(n) :
        S = 0
        for k in range(1,n+1) :
                S += 1 / k
        return S
sage: somme1(100000).n(64)
12.090146129863279
```

Ce n'est pas tout à fait pareil. Allez, encore une idée saugrenue ! Calculons la somme dans l'autre sens : de 100 000 jusqu'à 1...

```
def somme2(n) :
    S = 0
    for k in range(n,0,-1) :
        S += 1.0 / k
    return S
```

qui donne :

```
In [22]: somme2(100000)
Out[22]: 12.090146129863408
```

Ce n'est pas pareil ! ? On nous aurait menti : la somme des réels n'est pas commutative ! ? Elle n'est d'ailleurs même pas associative :

```
In [14]: (1 + 1e-16) + 1e-16
Out[14]: 1.0
```

```
In [15]: 1 + (1e-16 + 1e-16)
Out[15]: 1.0000000000000002
```

Et la multiplication non plus :

```
In [1]: (1.00000001 * 1e-15) * 1e15
Out[1]: 1.0000000100000002
```

```
In [2]: 1.00000001 * (1e-15 * 1e15)
Out[2]: 1.00000001
```

Et cette dernière n'est pas distributive sur la première :

```
In [3]: 1e15 * (1e-16 + 1)
Out[3]: 1000000000000000.0
```

```
In [4]: (1e15 * 1e-16) + (1e15 * 1)
Out[4]: 1000000000000000.1
```

2.4.2 Comment la machine fait-elle des additions ?

Et bien pas comme nous ! Mais elle calcule « bien » quand même...selon ses critères et ses capacités. Le calcul en virgule flottante est un sujet d'étude en soi mais trop méconnu à la fois par les mathématicien(ne)s (trop près de la machine et trop loin des mathématiques « habituelles »...) et les informaticien(ne)s (trop près des mathématiques...et également trop près de la machine pour beaucoup !).

```
In [9]: 2.**53
Out[9]: 9007199254740992.0

In [10]: 2.**53 + 1
Out[10]: 9007199254740992.0

In [11]: 2.**53 + 2
Out[11]: 9007199254740994.0
```

```
In [12]: 2.**53 + 3
Out[12]: 9007199254740996.0

In [13]: 2.**53 + 4
Out[13]: 9007199254740996.0

In [14]: 2.**53 + 5
Out[14]: 9007199254740996.0
```

Il faut avoir en tête que la machine ne travaille pas dans $\mathbb{R}$ mais sur un ensemble fini où on peut parler de successeur et de prédécesseur. Nous travaillerons dans l'ensemble le plus commun, $\mathbb{V}_{64}$, celui des binary64 c'est-à-dire des nombres en base 2 qui sécrivent sur 64 bits selon le découpage suivant :

- un bit de signe s qui vaut 0 si le nombre est positif, 1 sinon ;
- 11 bits d'exposant décalé e ;
- 53 bits de mantisse (ou « significande » en franglais) m, $\quad 1 \leqslant m$.

Oups ! $1 + 11 + 53 = 65$...En fait, on représente un nombre de $\mathbb{V}_{64}$ sous **forme normale** :

$$v = (-1)^s \times 1, f \times 2^{\mathrm{E}}$$

avec $m = 1, f$: point n'est besoin de stocker le 1 qui est implicite donc on ne stocke que f et on gagne 1 bit. On appelle f la pseudo-mantisse. De même il faudrait, pour l'exposant, garder un

bit pour stocker son signe. On va en fait le décaler pour que l'exposant stocké commence à 1 : on gagne encore un bit de stockage. Pour cela on le décale de l'exposant positif maximum - 1. On va voir que l'on réserve l'exposant 0 et l'exposant maximum pour représenter des nombres spéciaux. En binary 64, cela signifie que le décalage vaut $\delta = 11111111110_2 = 2^{11-1} - 1 = 1023$ et

$$e_{\text{stocké}} = E_{\text{réel}} + \delta$$

Pour en revenir à notre problème précédent, comment représenter $2^{53} + 1$ sur 64 bits ? Sur machine

$$2^{53} = (-1)^0 \times 1,000...000 \times 2^{53+1023}$$

```
0  10000110100  0000000000000000000000000000000000000000000000000000
```

$$1 = (-1)^0 \times 1,000...000 \times 2^{0+1023}$$

```
0  01111111111  0000000000000000000000000000000000000000000000000000
```

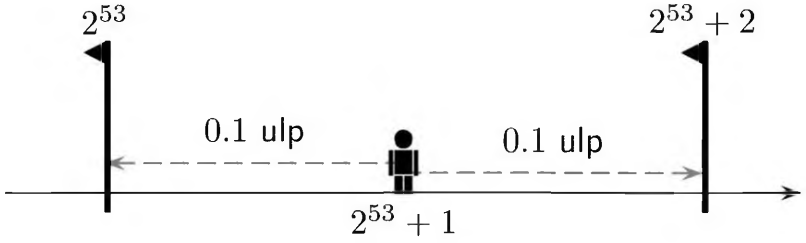
Or, pour additionner deux nombres à virgule flottante, il faut d'abord exprimer le plus petit dans l'exposant du plus grand (donc aussi éventuellement le « dénormaliser »).

$$1 = 0,\underbrace{000...000}_{52\text{bits}}\ 1 \times 2^{53}$$

Cela donne :

```
1,0000000000000000000000000000000000000000000000000000|
0,0000000000000000000000000000000000000000000000000000|1
----------------------------------------------------------
1,0000000000000000000000000000000000000000000000000000|1
```

On peut illustrer par le schéma suivant :



On ne peut pas mettre le dernier 1 dans la pseudo-mantisse qui n'a que 52 cases. Le nombre obtenu n'est donc pas un « flottant » : il va falloir l'arrondir. La norme IEEE 754 propose quatre modes d'arrondi. Le mode par défaut est l'arrondi au plus proche (RN : Round to the Nearest). En cas d'équidistance, on choisit le flottant dont la pseudo-mantisse se termine par un 0.
Il n'y a donc pas de nombre VF entre 2^{53} et $2^{53} + 2$ comme illustré sur le schéma. L'ULP est l'écart entre deux VF successifs qui varie selon l'ordre de grandeur du nombre. ULP signifie *Unit in the Last Place* : l'unité en dernière position.
Si v est un VF, entre v et son successeur, il n'y a pas de VF !

Ici, il ne faut pas oublier que l'on travaille en base 2 : notre nombre, qui n'est pas un flottant de $\mathbb{V}_{64}$, est à équidistance de $1,\underbrace{000...000}_{52\text{bits}}$ et $1,\underbrace{000...001}_{52\text{bits}}$. Il sera arrondi à celui qui se termine par un 0 : 2^{53}.

$$\mathrm{RN}(2^{53}+1) = 2^{53}$$

Vous pouvez alors comprendre pourquoi :

$$\mathrm{RN}(2^{53}+3) = 2^{53}+4$$

La mauvaise nouvelle : une addition entre deux nombres flottants peut donc, dans certains cas, créer une erreur.
La bonne nouvelle : on peut récupérer cette erreur.

2.4.3 Les sommes compensées de deux flottants

Par la suite, on notera $x \ominus y$ l'arrondi de la somme de $x-y$ c'est-à-dire $x-y+err(x\ominus y)$.
Voici un lemme publié en 1973 par Pat STERBENZ :

Théorème 3.5 (Lemme de STERBENZ (1973)). Soit $(x,y) \in \mathbb{V}^2$ vérifiant $\frac{x}{2} \leqslant y \leqslant 2x$. On a

$$x \ominus y = x - y$$

La différence de deux nombres de $\mathbb{V}$ suffisamment proches est donc exacte.

La condition $\frac{x}{2} \leqslant y \leqslant 2x$ signifie en fait que x et y sont distants d'une puissance de 2 au maximum. Que se passe-t-il quand on ne peut pas s'assurer à chaque instant que les opérandes sont suffisamment proches ? Voici maintenant un premier algorithme dû à T.J. DEKKER et W. KAHAN qui répond à cette question en **Python** :

Théorème 3.6 (Fast2Sum - DEKKER & KAHAN). On considère deux VF x et y tels que $|x| \geqslant |y|$ et l'algorithme suivant :

```
1  s ← x ⊕ y
2  y_v ← s ⊖ x
3  d ← y ⊖ y_v
4  Retourner (s, d)
```

Alors $x+y = s+d$ avec $s = x \oplus y$ et $d = \mathrm{err}(x \oplus y)$. De plus s et d ne se chevauchent pas.

La ligne 1 donne $s = x+y+\mathrm{err}(x\oplus y)$. De plus, comme $|x| \geqslant |y|$, c'est x qui « gagne » donc s et x ont le même signe. La ligne 2 calcule un y virtuel (y_v) qui vaut $s \ominus x$: on aimerait bien que ça fasse $s-x$ c'est-à-dire $y+\mathrm{err}(x\oplus y)$. Enfin la ligne 3 calcule $y \ominus y_v$: on aimerait bien aussi que ça fasse $y-y_v$ c'est-à-dire $-\mathrm{err}(x\oplus y)$ comme ça on récupérerait exactement l'erreur commise. En plus, comme $d = -\mathrm{err}(x\oplus y) =$, on a $|d| \leqslant \frac{1}{2}\mathrm{ulp}(s)$ donc s et d ne se chevauchent pas dans leur somme. On aimerait bien, on aimerait bien...En fait tout s'arrange car nous allons pouvoir utiliser le lemme de Sterbenz .
Il suffit de distinguer deux cas :

a) si x et y sont de même signe OU si $|\mathbf{y}| \leqslant \frac{|\mathbf{x}|}{\mathbf{2}}$: alors $\frac{x}{2} \leqslant s \leqslant 2x$ et on peut appliquer le lemme ;

b) sinon : on a x et y de signes opposés ET $y > \frac{|x|}{2}$ alors si y est négatif $x/2$.

On illustrera encore mieux le problème par ce schéma :

x	x_1	x_2	
$+y$		y_1	y_2
$= s$	x_1	$x_2 + y_1$	y_2 est perdu
$-x$		y_1	0
$-y$			$-y_2$	0

On récupère l'erreur $-y_2$.

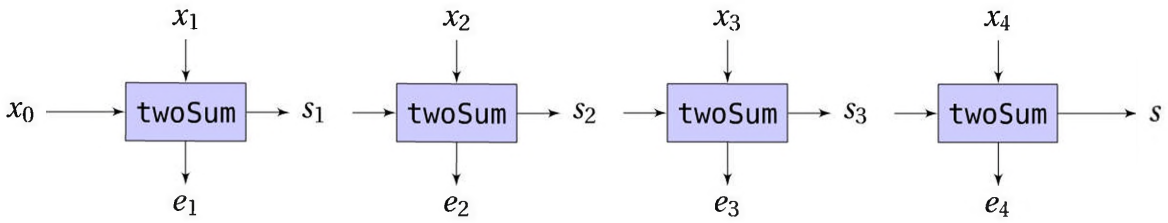
Ainsi, $x + y = s + d$ se lit :

x	x_1	x_2	
$+y$		y_1	y_2
$= s$	x_1	$x_2 + y_2$	y_2 est perdu
$+d$			y_2	0

On récupère l'erreur y_2

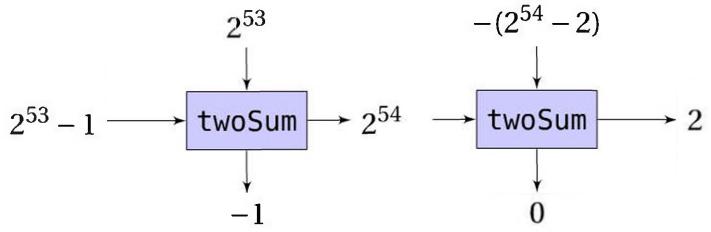
2.4.4 Les sommes compensées d'un nombre quelconque de flottants

L'idée de cet algorithme, dit de sommation en cascade, est d'additionner dans un accumulateur les erreurs puis les additionnée à la pseudo-somme à la fin. Cette idée a été publiée par Michèle PICHAT en 1972 dans un mémoire de maîtrise et simultanément par Arnold NEUMAIER et ensuite généralisé en 2008 par SIEGFRIED M. RUMP, TAKESHI OGITA et SHIN'ICHI OISHI. L'idée tient en un petit dessin :



Voyons tout de suite un exemple. On veut effectuer la somme en *binary64* de $x_0 = 2^{53} - 1$, $x_1 = 2^{53}$ et $x_2 = -(2^{54} - 2)$.

Que vaut la somme exacte (sans machine car la machine se trompe...) ?



En **Python** :

```
def Fast2Sum(a,b) :
    if a >= b:
        s = a + b
        z = s - a
        return (s, b - z)
    else:
        return Fast2Sum(b,a)
```

```
def Pichat(liste) :
    accS = 0
    accE = 0
    for k in liste:
        s,e = Fast2Sum(accS,k)
        accS  = s
        accE += e
    return accS + accE
```

et on obtient :

```
In [1]: Pichat([1.0 / k for k in range(1,100001)])
Out[1]: 12.090146129863427
```

2.4.5 La machine compte en base 2

Comment expliquer les résultats suivants :

```
In [1]: sum([0.1 for k in
    range(100000000)])
Out[1]: 9999999.98112945
```

```
In [2]: sum([0.125 for k in
    range(100000000)])
Out[2]: 12500000.0
```

Déterminons l'écriture de 0,1 en base 2. Ce nombre est égal à 1/10 en base 10 c'est-à-dire 1/1010 en base 2. Posons la division :

```
 1       | 1010
         ------
 10      | 0,00011
 100     |
 1000    |
 10000   |
- 1010   |
------   |
   1100  |
-  1010  |
-------  |
     10  |
```

Ça ne tombe pas juste, alors que 0,125 en base 2 s'écrit 0,001 exactement car il est égal à $1/8 = 2^{-3}$.
Voici un petit algorithme permettant de convertir un nombre en base 10 de valeur absolue inférieure à 1 en base 2.
Soit $x = a_1 \times 2^{-1} + a_2 \times 2^{-2} + a_3 \times 2^{-3} + \cdots + a_n \times 2^n$ l'écriture de x tel que $|x| < 1$ en base 2 avec les a_i égaux à 0 ou 1.
Alors $2x = a_1 + a_2 \times 2^{-1} + a_3 \times 2^{-2} + \cdots + a_n \times 2^{n-1}$: a_1 est donc la partie entière de $2x$.
Puis $2(2x - a_1) = a_2 + a_3 \times 2^{-1} + \cdots + a_n \times 2^{n-2}$: a_2 est alors la partie entière de $2(2x - a_1)$,...

```
from math import floor
def petits2bits(x,nmax) :
    dec = abs(x)
    assert dec < 1, "Le nombre doit être de valeur absolue < 1"
    bin = ""
    k = 0
    while dec != 0 and k < nmax:
        a = floor(2*dec)
        bin += str(a)
        dec = 2*dec - a
        k += 1
    return '0:' + bin
```

On retrouve les résultats précédents :

```
In [1]: petits2bits(0.125,50)
Out[1]: '0:001'
In [2]: petits2bits(0.1,50)
Out[2]: '0:00011001100110011001100110011001100110011001100110'
```

Il faudra donc éviter de diviser des nombres par des puissances de 10 et préférer les puissances de 2.

2.4.6 Calcul de limites - Indéterminations

Observons :

```
In [1]: x = 1e500

In [2]: 1 + x
Out[2]: inf
In [3]: x**2
Out[3]: inf
In [4]: 1 / x
Out[4]: 0.0
```

```
In [5]: 4 - x
Out[5]: -inf
In [6]: x**2 - x
Out[6]: nan
In [7]: x**2 - x == x**2 - x
Out[7]: False
In [8]: x * (x - 1) == x**2 - x
Out[8]: False
```

On rappelle qu'un VF est représenté par un bit de signe, une mantisse (ou significande en franglais) et un exposant :

$$v = (-1)^s \times m \times 2^E$$

Les formats habituels sont le *binary32* (ou *single* mais c'est ambigu) avec $(\#E, \#m) = (8,24)$ et le *binary64* (ou *double*) avec $(\#E, \#m) = (11,53)$.
Pour simplifier nos présentations, on illustrera souvent nos propos avec des représentations *toy7* plus petites, comme par exemple $(\#E, \#m) = (3,4)$.
La mantisse doit dans la mesure du possible vérifier $1 \leqslant m < 2$ ce qui minimise l'exposant et apporte plus d'informations. Par exemple, en *toy7* :

$$\begin{aligned} v &= 0,010\,01 \quad \times 2^{0} \\ v &= 0,100\,1 \quad \times 2^{-1} \\ v &= 1,001 \quad \times 2^{-2} \end{aligned}$$

C'est la *forme normale* d'un VF. On notera $m = 1,f$.
On rappelle qu'on peut gagner de la place en ne stockant pas le signe de l'exposant : il suffit de le translater de $2^{(\#E)-1} - 1$

$$e = E + 2^{(\#E)-1} - 1$$

Représentons $0,75_{10}$ en *toy7*.

$$0,75_{10} = \frac{3_{10}}{2^2_{10}} = 11_2 \times 2^{-2} = 1,100_2 \times 2^{-1}$$

L'exposant stocké e vérifie avec le décalage :

$$e - \left(2^{3-1} - 1\right) = E = -1 \Leftrightarrow e = -1 + 3 = 2 = 10_2$$

Ainsi :

s (1 bit)	e (3 bits)			f (3 bits)		
0	0	1	0	1	0	0

On doit également s'occuper des exposants extrêmes qui ne correspondent pas à des formes normales.
Nous venons de voir que l'exposant E, qui est un entier signé, est stocké sur #E bits sous la forme $e = E + 2^{(\#E)-1} - 1$.
Comme e est un entier naturel stocké sur #E bits, il varie entre $e_{\min} = 0$ et $e_{\max} = 2^{\#E} - 1$.
Par exemple, en *toy7*, #E = 3 donc $e_{\min} = 000$ et $e_{\max} = 111 = 1000 - 1$.
CEPENDANT les valeurs extrêmes de E sont réservées pour des cas spéciaux.
Cela donne :

$$\begin{aligned} E_{\min} &= \left(e_{\min} - 2^{(\#E)-1} + 1\right) + 1 = 0 - 2^{3-1} + 1 + 1 = -2 \\ E_{\max} &= \left(e_{\max} - 2^{(\#E)-1} + 1\right) - 1 = (2^3 - 1) - 2^{3-1} + 1 - 1 = 7 - 4 + 1 - 1 = 3 \end{aligned}$$

En effet, des problèmes apparaissent avec les formes normales :

- on ne peut pas représenter zéro sous forme normale ;
- des nombres du type $0.\underbrace{000...00}_{\#E}11 \times 2^0 = 1.1 \times 2^{-(\#E+1)}$ ne sont pas représentables sous forme normale car il faudrait un exposant plus petit que $E_{\min}$.

La norme IEEE 754 a donc introduit les nombres *sous-normaux* pour remplir le vide qui existerait entre 0 et le plus petit nombre normal.
Pour les signaler et les distinguer des normaux, la norme impose que leur exposant décalé e soit 0 : $e_{\min}$ est donc une valeur réservée.
Un cas particulier : si $e = e_{min}$ et $f = 0$, alors la norme ne considère ce nombre ni comme un nombre normal ni comme un nombre sous-normal. C'est la représentation de zéro.
Reprenons à présent un exemple classique : vous voulez calculer une norme $\mathrm{N} = \sqrt{x^2 + y^2}$ avec $x = 1 \times 2^3$ et $y = 1,1 \times 2^3$ en *toy7*.
Les carrés de ces nombres posent problème : leur exposant est trop grand. On est dans un cas de *dépassement* ou d'*overflow* en english de spécialité.
Une première idée serait de remplacer x^2 et y^2 par le plus grand nombre disponible $1,111 \times 2^4$ soit :

s (1 bit)	e (3 bits)			f (3 bits)		
0	1	1	1	1	1	1

Alors x^2 puis y^2 puis $x^2 + y^2$ seraient remplacés par $1,111 \times 2^4$.
On aurait donc $\mathrm{N} = \left(1,111 \times 2^4\right)^{1/2}$ arrondi à $1,010 \times 2^2$ ce qui est dramatiquement faux.
La norme introduit donc deux infinis codés en *toy7* :

$+\infty$:

s (1 bit)	e (3 bits)			f (3 bits)		
0	1	1	1	0	0	0

$-\infty$:

s (1 bit)	e (3 bits)			f (3 bits)		
1	1	1	1	0	0	0

Soit, plus généralement, $e = e_{\max}$ et $f = 0$.
On retrouve alors les limites habituelles vues au lycée :
Enfin la norme introduit le `NaN` (comme dans *Not a Number*) dans les cas où le calcul demandé est « indéfini ».
Un `NaN` n'est pas comparable. Trier un tableau comportant un `NaN` ne pose donc pas de problème.

```
In [42]: x = 1e500
In [43]: n = x - x
In [44]: n > 3
Out[44]: False
```

```
In [45]: n <= 3
Out[45]: False
In [46]: n == n
Out[46]: False
```

Reprenons nos formes indéterminées :

```
In [71]: x = 1e500

In [72]: x**2 - x
```

```
Out[72]: nan
In [73]: x*(x - 1)
Out[73]: inf
```

Pour éviter le NaN, on lève l'indétermination...
Selon la IEEE754, un `NaN` est représenté avec l'exposant maximum et une mantisse non nulle : il y a donc toute une famille de `NaN` ! En voici un exemple en *toy7* :

s (1 bit)	e (3 bits)			f (3 bits)		
1	1	1	1	0	1	0

La machine traite donc une chaîne de bits $\boxed{s}\boxed{e}\boxed{f}$ ainsi :

$e = 0...0$	$f = 0$	$\rightarrow$	$v = (-1)^s \times 0.0$
$e = 0...0$	$f \neq 0$	$\rightarrow$	$v = (-1)^s \times 0.f \times 2^{1-E_{max}}$
$e = 1...1$	$f = 0$	$\rightarrow$	$v = (-1)^s \times \infty$
$e = 1...1$	$f \neq 0$	$\rightarrow$	NaN
$0...0 < e < 1...1$	$f \neq 0$	$\rightarrow$	$v = (-1)^s \times 1.f \times 2^{e-E_{max}}$

On notera $\varepsilon_m = 2^{-n}$ l'*epsilon* de la machine, c'est-à-dire le successeur de 1.
On notera λ le plus petit VF normal positif.
On notera μ le plus petit VF sous-normal positif.
On notera Ω le plus grand VF normal.
On obtient alors le tableau suivant :

Format	#E	#f	E_{min}	E_{max}	ε_m	λ	μ	Ω
toy7	3	3	-2	3	$2^{-3} = 1/8$	$2^{-2} = 1/4$	$2^{-5} = 1/32$	$1,111 \times 2^3 = 15$
bin32	8	23	-126	127	2^{-23} $\approx 1,2 \times 10^{-7}$	2^{-126} $\approx 1,2 \times 10^{-38}$	$2^{-126-23}$ $\approx 1,4 \times 10^{-45}$	$(2^{24}-1) \times 2^{127-23}$ $\approx 3,4 \times 10^{38}$
bin64	11	52	-1022	1023	2^{-52} $\approx 2,2 \times 10^{-16}$	2^{-1022} $\approx 2,2 \times 10^{-308}$	2^{-1074} $\approx 5 \times 10^{-324}$	$(2^{53}-1)2^{1023-52}$ $\approx 1,8 \times 10^{308}$

Exercice 43. *a*) Comment expliquer les résultats suivants :

```
In [49]: f =lambda x:  x**2 / sqrt(x**3 + 1)

In [50]: f(1e500)
Out[50]: nan

In [51]: f(1e100)
Out[51]: 1e+50

In [52]: f(1e150)
---------------------------------------------------------------------------
OverflowError                             Traceback (most recent call last)
<ipython-input-52-79775d85f31a> in <module>()
----> 1 f(1e150)

<ipython-input-49-ba3dba799396> in <lambda>(x)
----> 1 f =lambda x:  x**2 / sqrt(x**3 + 1)

OverflowError: (34, 'Numerical result out of range')
```

b) Comment éviter le premier NaN ?

Solution. `1e500` dépasse la taille maximum sur 64 bits donc est égal à `inf`. Le calcul de $f(x)$ entraîne ici un rapport entre deux `inf` donc est égal à `nan`.

Pour `1e100` on a la réponse attendue sachant que `1e300 + 1 = 1e300`.
Pour le dernier, c'est une erreur due à une mauvaise implémentation de la norme IEEE754 sur **Python** qui renvoie une erreur pour le calcul de `1e150**3` mais aurait dû renvoyer `inf`.

2.4.7 Trop de précision tue la précision

On veut avoir une approximation de la dérivée d'une fonction en un point donné. On utilise la définition standard :

```
def deriv(f,h,x) :
    return (f(x + h) - f(x)) / h
```

Et pour la dérivée n^{e} :

```
def derivn(f,x,h,n) :
    if n == 0:
        return f(x)
    else:
        return derivn(lambda x: deriv(f,h,x),x,h,n-1)
```

Regardons ce qui se passe pour $x \mapsto x^4$ en 1 : on doit obtenir 1, 4, 12, 24, 24 puis toujours 0.

```
In [1]: [derivn(lambda x: x**4,1,1e-7,k) for k in range(5)]
Out[1]:
[1,
 4.000000601855902,
 12.012613126444194,
 -222044.6049250313,
 2220446049250.313]
```

Prenons alors un h plus petit pour être plus précis :

```
In [2]: [derivn(lambda x: x**4,1,1e-8,k) for k in range(5)]
Out[2]: [1, 4.000000042303498, 11.102230246251565, 0.0, 2.220446049250313e+16]
```

Insistons :

```
In [3]: [derivn(lambda x: x**4,1,1e-9,k) for k in range(5)]
Out[3]: [1, 4.000000330961484, 0.0, 0.0, 0.0]
```

Prenons en fait un plus grand h :

```
In [4]: [derivn(lambda x: x**4,1,1e-3,k) for k in range(6)]
Out[4]:
[1,
 4.006004000999486,
 12.024014000244776,
 24.03599941303014,
 24.001245435556484,
 -2.4424906541753444]
```

C'est déjà mieux. Ensuite on se souvient qu'il faut éviter les divisions par des puissances de 10 et préférer les puissances de 2 :

```
In [5]: [derivn(lambda x: x**4,1,2**-10,k) for k in range(6)]
Out[5]: [1, 4.005863190628588, 12.02345085144043, 24.03515625, 24.0, 0.0]
```

C'est quand même mieux mais il ne faut pas exagérer :

```
In [7]: [derivn(lambda x: x**4,1,2**-20,k) for k in range(6)]
Out[7]: [1, 4.000005722045898, 12.0, 0.0, 268435456.0, -844424930131968.0]
```

2.4.8 Des sujets de Bac dangereux

Le sujet du Bac S des Centres Étrangers 2012 fait étudier la suite (vu dans BV APMEP 507) définie sur $\mathbb{N} \setminus \{0\}$ par :

$$\mathrm{I}_n = \int_0^1 x^n \mathrm{e}^{x^2} \mathrm{d}x$$

On obtient par intégration par parties :

$$\mathrm{I}_{n+2} = \frac{1}{2}\mathrm{e} - \frac{n+1}{2}\mathrm{I}_n, \qquad \mathrm{I}_1 = \frac{1}{2}(\mathrm{e}-1)$$

En classe, il est souvent demandé de deviner le comportement de la suite en observant les premiers termes :

```
def bac(n) :
    if n == 1:
        return 0.5 * (exp(1.) - 1.)
    return (0.5 * (exp(1.) - (n - 1) * bac(n - 2)))
```

```
In [1]: [(2*k + 1,bac(2*k + 1)) for k
    in range(30)]
Out[1]:
[(1, 0.8591409142295225),
 (3, 0.5),
 (5, 0.35914091422952255),
 (7, 0.2817181715409549),
 (9, 0.2322682280657029),
 (11, 0.197799773901008),
 (13, 0.17234227082347453),
 (15, 0.15274501846520083),
 (17, 0.13718076650791589),
 (19, 0.12451401565827958),
 (21, 0.11400075764672679),
 (23, 0.10513258011552784),
 (25, 0.09754995284318846),
 (27, 0.09099152726807258),
 (29, 0.08525953247650642),
 (31, 0.0802479270819263),
 (33, 0.07517408091870181),
 (35, 0.08118153861159172),
 (37, -0.1021267807791284),
 (39, 3.299549749032962),
 (41, -64.63185406642971),
 (43, 1358.6280763092534),
 (45, -29888.458537889346),
 (47, 687435.9055123692),
 (49, -16498460.373155948),
 (51, 412461510.6880396),
 (53, -10723999276.52989),
 (55, 289547980467.66614),
 (57, -8107343453093.293),
 (59, 235112960139706.84)]
.
```

donc la suite oscille entre $+\infty$ et $-\infty$... pourtant la suite est décroissante et à valeurs positives si l'on étudie la définition de la suite sous forme intégrale ;-)

2.4.9 Prolongements

Pour explorer plus avant l'arithmétique des flottants et des intervalles, des références ont été données en bibilographie :

- des pages personnelles très riches : [Kah14], [dD14], [Gou14] ;
- un article de référence sur le mauvais traitement des flottants par Java : [KD98] ;
- les articles historiques introduisant les théorèmes cités dans ce chapitre : [Ste73], [Pic72] ;
- La référence pour comprendre l'arithmétique des flottants : [MBdD+10] ;
- un article qui résume l'essentiel : [Gol91].

2.5 Les nombres complexes

2.5.1 Les complexes comme couples de réels

Une première idée est de construire les complexes comme des couples de réels. On définit pour cela une classe la plus basique possible avec un opérateur `alg` pour afficher la forme algébrique :

```
class cplx:
       def __init__(self,partie_reelle,partie_imaginaire) :
              self.r = partie_reelle
              self.i = partie_imaginaire

       def alg(self) :
              return str(self.r) + ' + '+str(self.i) + '(i)'
```

Par exemple :

```
In [1]: z = cplx(3,2)
In [2]: z.alg()
Out[2]: '3 + 2(i)'
```

On peut alors définir addition et multiplication en utilisant les opérateurs de base de **Python**. On pourrait, si nécessaire, ne travailler qu'avec les rationnels définis dans une section précédente et utiliser alors les opérateurs spécifiques qui avaient été introduits.

```
def csom(z1,z2) :
    return cplx(z1.r + z2.r, z1.i + z2.i)
def cprod(z1,z2) :
    return cplx(z1.r*z2.r - z1.i*z2.i, z1.r*z2.i + z1.i*z2.r)
```

On vérifie qu'il existe bien un complexe dont le carré vaut -1 :

```
In [1]: z1 = cplx(0,1)
In [2]: cprod(z1,z1).alg()
Out[2]: '-1 + 0(i)'
```

On définit le conjugué :

```
def conj(z) :
    return cplx(z.r, -z.i)
```

Par exemple :

```
In [1]: z = cplx(3,4)
In [2]: cprod(z,conj(z)).alg()
Out[2]: '25 + 0(i)'
```

Pour le module, on utilisera l'opérateur `sqrt` du module `math` de **Python** :

```
def cmod(z) :
      return sqrt(z.r*z.r + z.i*z.i)
```

On précise à **Python** qu'on prend la partie réelle de $z \cdot \overline{z}$ même si l'on sait que la partie imaginaire est nulle pour pouvoir utiliser `sqrt` sur un objet de type réel et non pas complexe.

```
In [1]: from math import sqrt
In [2]: z = cplx(3,4)
```

```
In [3]: cmod(z)
Out[3]: 5.0
```

On peut également définir l'inverse d'un complexe non nul et donc une division :

```
def cinv(z) :
    if z == cplx(0,0) :
        raise ZeroDivisionError
    return cplx(z.r / (cmod(z))**2, -z.i / (cmod(z)**2))

def cdiv(z1,z2) :
    return cprod(z1,inv(z2))
```

Par exemple :

```
In [1]: z = cplx(1,1)
In [2]: cinv(z).alg()
Out[2]: '0.5 + -0.5(i)'
```

```
In [3]: cprod(z,cinv(z)).alg()
Out[3]: '1.0 + 0.0(i)'
```

Il y a cependant quelques petits problèmes avec le calcul des inverses dès qu'on s'approche trop de zéro :

```
In [1]: z = cplx(1,1e-8)
In [2]: cinv(z).alg()
Out[2]: '1.0 + -1e-08(i)'
```

```
In [3]: cprod(z,cinv(z)).alg()
Out[3]: '1.0 + 0.0(i)'
```

2.5.2 Les complexes comme matrices

Notons $U = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$ et $J = \left(\begin{smallmatrix} 0 & -1 \\ 1 & 0 \end{smallmatrix}\right)$. On vérifie aisément que $J^2 = -U$ et $U^2 = U$.
On considère l'ensemble des matrices obtenues par combinaisons linéaires de U et J.
Elles sont de la forme $Z = aU + bJ = \left(\begin{smallmatrix} a & -b \\ b & a \end{smallmatrix}\right)$ avec a et b des réels. Appelons ces matrices *cellules* et $\mathcal{C}$ leur ensemble.
On peut demander aux élèves de comparer $\mathcal{C}$ et $\mathbb{C}$.
On peut alors définir les opérations sur les nombres complexes à partir des opérations sur les matrices.

Nous reprendrons les notations et les opérations matricielles introduites à la section 1 page 78 pour définir une nouvelle classe de nombres et les opérations introduites dans la section précédente à partir des opérations sur les matrices :

```
class cplx:
    def __init__(self,partie_reelle,partie_imaginaire) :
        self.r = partie_reelle
        self.i = partie_imaginaire

    def mcplx(self) :
        return (U * self.r) + (J * self.i)

    def alg(self) :
       return str(self[0][0])+' + '+str(self[1][0])+'(i)'

    def __add__(self, other) :
       return self.mcplx() + other.mcplx()

    def __mul__(self, other) :
       return self.mcplx() * other.mcplx()

    def mconj(self) :
       return U * self.r + J * -self.i
```

Par exemple :

```
In [1]: Z1 = cplx(3,4), Z2 = cplx(1,1)
In [2]: Z1 * Z2
Out[2]: -1 -7
         7  -1
In [3]: (Z1 * Z2).alg()
Out[3]: '-1 + 7(i)'
```

```
In [4]:  Z2.mconj()
Out[4]:  1  1
        -1  1
In [5]: alg(mconj(Z2))
Out[5]: '1 + -1(i)'
```

2.6 Exercices d'entraînement

Les corrigés sont disponibles sur le site dunod.com à partir de la page d'accueil de l'ouvrage.

Exercice 44. (Multiplication du paysan russe) Voici ce qu'apprenaient les petits soviétiques pour multiplier deux entiers. Une variante de cet algorithme a été retrouvée sur le papyrus de Rhind datant de 1650 av. J.-C, le scribe AHMES affirmant que cet algorithme était à l'époque vieux de 350 ans. Il a survécu en Europe occidentale jusqu'aux travaux de FIBONACCI.

```
1  Fonction MULRUSSE( x: entier ,y: entier, acc:entier ): entier
2  Si x == 0 Alors
3  |  Retourner acc
4  Sinon
5  |  Si x est pair Alors
6  |  |  Retourner MULRUSSE(x // 2, y*2, acc)
7  |  Sinon
8  |  |  Retourner MULRUSSE((x - 1) // 2, y*2, acc + y)
9  |  FinSi
10 FinSi
```

Que vaut `acc` au départ ?
Écrivez une version récursive de cet algorithme en évitant l'alternative des lignes 5 à 9.
Écrivez une version Python pour notre classe `Bits`

Exercice 45. (Algorithme de Karatsouba) On s'attaque d'abord à l'algorithme naïf de multiplication en divisant la taille des opérandes par 2 :

Bits.py

```
def mul_divis(self, other) :
        """Algo dichotomique naïf de multiplication"""
        op1,op2 = self.norm(),other.norm()
        if op1.signe() == 1:
            return -((-op1).mul_divis(op2))
        if op2.signe() == 1:
            return -(op1.mul_divis(-op2))
        long = max(len(op1), len(op2))
        op1 = Bits([0 for i in range(len(op1), long)] + op1.bits)
        op2 = Bits([0 for i in range(len(op2), long)] + op2.bits)

        if long <= 2:
            return Bits([0, op1[1] & op2[1]])

        m0 = (long + 1) >> 1 # ie // 2
        m1 = long >> 1

        x0 = Bits([0] + op1[  : m0]).norm()
        x1 = Bits([0] + op1[m0 :  ]).norm()
        y0 = Bits([0] + op2[  : m0]).norm()
        y1 = Bits([0] + op2[m0 :  ]).norm()

        p0 = x0.mul_divis(y0)
        p1 = x1.mul_divis(y0)
        p2 = x0.mul_divis(y1)
        p3 = x1.mul_divis(y1)

        z0 = Bits(p0.bits + [0 for i in range(0,  m1 << 1)])
        z1 = Bits((p1 + p2).bits + [0 for i in range(0, m1)])
```

```
        z2 = p3

        return z0 + z1 + z2
```

Modifiez légèrement le code précédent pour écrire la fonction correspondant à l'algorithme de KARATSOUBA.

Exercice 46. (Le Pape Grégoire et les fractions continues) Voici l'énoncé d'un problème proposé par Ernst HAIRER et Gerhard WANNER[voir HW00, p. 78] :
La longueur d'une année astronomique est de (EULER 1748) :

365 jours 5 heures 48'55"

Calculer le développement de 5 heures 48'55" (mesurés en jours) en une fraction continue et chercher les réduites correspondantes. Le Pape Grégoire XIII aurait apprécié vos précieux conseils pour la réforme de son calendrier, même si la précision retenue est largement suffisante.

Exercice 47. (Algorithme de MALCOLM-GENTLEMAN) Expliquez le rôle de l'algorithme suivant :

1 **Algorithme** de MALCOLM-GENTLEMAN
2 $a \leftarrow 1.0$
3 $b \leftarrow 1.0$
4 **TantQue** $\mathrm{R}(\mathrm{R}(a+1.0)-a) == 1.0$ **Faire**
5 | $a \leftarrow \mathrm{R}(2 \times a)$
6 **FinTantQue**
7 **TantQue** $\mathrm{R}(\mathrm{R}(a+b)-a) \neq b$ **Faire**
8 | $b \leftarrow b+1$
9 **FinTantQue**
10 **Retourner** b

qu'on peut écrire en **Python** :

```
def base1(a) :
    if (a + 1.0) - a != 1.0:
        return a
    else:
        return base1(2.0 * a)
```

```
def base2(a, b) :
    if (a + b) - a == b:
        return b
    else:
        return base2(a, b + 1.0)
```

Regardons la première boucle. On y construit une suite (a_i) vérifiant pour tout naturel i $a_{i+1} = 2a_i$ avec $a_0 = 1.0$. Une récurrence immédiate démontre que $a_i = 2^i$ pour tout entier naturel i et notre boucle ne s'arrête jamais et pourtant :

```
In [1]: base1(1.0)
Out[1]: 9007199254740992.0
```

```
In [2]: 2**53
Out[2]: 9007199254740992
```

3 Le nombre π

3.1 L'approximation de π de Nicolas De Cues

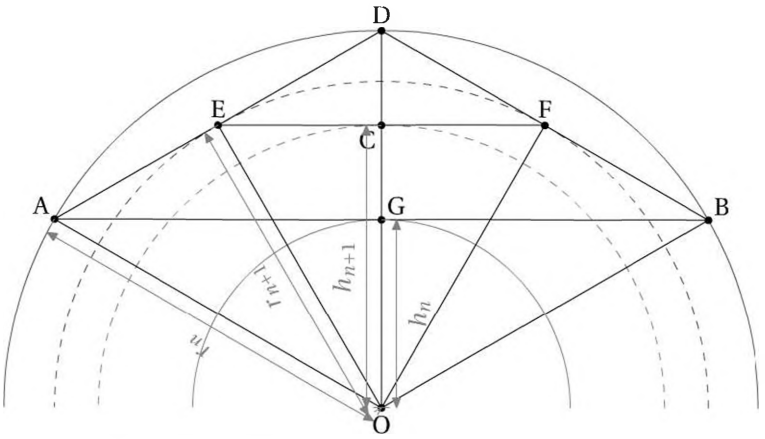
Nicolas De Cues, cardinal allemand du XV^e siècle, s'est beaucoup intéressé aux mathématiques. En 1450, il publie *De Quadratura circuli* qui, comme son nom l'indique, étudie le problème de la quadrature du cercle.
En voici une version simplifiée mais suivant le même principe : on « cale » des polygones réguliers de 2^n côtés et de périmètre 2 entre deux cercles.
On a alors :

$$2\pi h_n \leqslant 2 \leqslant 2\pi r_n$$

c'est-à-dire :

$$\frac{1}{r_n} \leqslant \pi \leqslant \frac{1}{h_n}$$

Il reste à trouver une relation donnant r_n et h_n...



Notons AB un côté du polygone régulier de 2^n côtés de périmètre 2 et de centre O. Comme $EF = \frac{AB}{2}$ alors EF est le côté du polygone suivant.
Ainsi

$$OD = r_n, \qquad OG = h_n, \qquad OE = r_{n+1}, \qquad OC = h_{n+1}$$

Comme $OC = \frac{OG+OD}{2}$, alors

$$h_{n+1} = \frac{r_n + h_n}{2}$$

Ensuite, ODE est rectangle en E, on peut faire démontrer en 2^e que $OE^2 = OH \times OD$ et donc que

$$r_{n+1} = \sqrt{r_n h_{n+1}}$$

On part d'un carré de périmètre 2 donc de côté $\frac{1}{2}$. On a donc $h_0 = \frac{1}{4}$ et $r_0 = \frac{\sqrt{2}}{4}$.
Le problème, c'est que nous ne travaillons pas en précision infinie. En chargeant le module `sys`, on a accès à la commande `float_info.epsilon` qui donne la différence entre `1.0` et le flottant suivant le plus proche.

```
>>> from sys import*
>>> float_info.epsilon
2.220446049250313e-16
```

On peut donc définir une fonction qui déterminera à partir de quand deux flottants seront considérés comme égaux par le système :

```
def pseudo_egal_float(a, b) :
    return abs(a - b) <= (float_info.epsilon * min(abs(a), abs(b)))
```

Si l'on n'y prête pas attention, on peut arriver à des résultats surprenants :

```
>>> pseudo_egal_float(.1 + .1 + .1 , 0.3)
True
>>> .1 + .1 + .1 == 0.3
False
>>> 3 * 0.1 == 0.3
False
>>> 4 * 0.1 == 0.4
True
>>> 10+float_info.epsilon-10
0.0
>>> 1+float_info.epsilon-1
2.220446049250313e-16
>>> 10+10*float_info.epsilon-10
1.7763568394002505e-15
```

Car en plus, le processeur compte en binaire...
Ces dangers étant écartés, construisons `cues()` :

```
def cues() :
    h = 1/4
    r = sqrt(2)/4
    compteur = 0
    while not(pseudo_egal_float(1/h,1/r)) :
        h = (r+h)/2
        r = sqrt(r*h)
        compteur += 1
    return 1/r,1/h,1/h-1/r,compteur
```

Alors on obtient 15 bonnes décimales en 25 itérations...Mais cela suppose de pouvoir calculer des racines carrées avec assez de précision...

```
>>> cues()
(3.1415926535897927, 3.141592653589793, 4.440892098500626e-16, 25)
```

3.2 Approximation de π et calcul approché d'intégrale

On s'intéresse à l'intégrale sur $[0,1]$ de la fonction $f : t \mapsto \sqrt{1-t^2}$ car bien sûr $\pi = 4\int_0^1 f(t)\,\mathrm{d}t$.

```
from math import sqrt,pi
def f(t) :
    return sqrt(1 - t**2)
```

Un problème demeure : comment calculer une approximation de cette intégrale ?
Pour un traitement plus complet des méthodes d'intégrations numériques, on pourra se référer à la section 6 page 205.

3.2.1 Méthode des rectangles

C'est la plus grossière :

```
def int_rec_droite(f,a,b,N) :
    S = 0
    t = a
    dt = (b-a)/N
    while t < b:
        S += f(t)*dt
        t += dt
    return S
```

```
def int_rec_gauche(f,a,b,N) :
    S = 0
    t = a
    dt = (b-a)/N
    while t+dt < b:
        S += f(t+dt)*dt
        t += dt
    return S
```

Il *semble* falloir 2^{15} itérations pour obtenir trois bonnes décimales de π...

```
In [5]: for k in range(15) : print(pi
    - 4*int_rec_gauche(f,0,1,2**k))
3.141592653589793
1.409541846020916
0.6458835854873524
0.3017735092326199
0.14333961576205168
0.06899023071266264
0.033545740810107194
0.016436861193178665
0.008099570632563058
0.004007750870138604
0.0019890120507732867
0.0009892506783506505
0.0004927672320342857
0.0002457266651032519
0.0001226310633999361 2
```

3.2.2 Méthode des trapèzes

```
def int_trap(f,a,b,N) :
    S = 0
    dt = (b-a)/N
    for k in range(N) :
        S+=(f(a+k*dt)+f(a+(k+1)*dt))*dt/2
    return S
```

```
In [7]: for k in range(15) : print(pi
    - 4*int_trap(f,0,1,2**k))
1.1415926535897931
0.4095418460209159
0.1458835854873528
0.05177350923261992
0.01833961576205212
0.006490230712662637
0.002295740810108526 4
```

```
0.0008118611931777764
0.0002870706325635019 6
0.0001015008701394926
3.58870507768394e-05
1.26881783466537e-05
4.485982035618008e-06
1.5860401174627725e-06
5.60750892830697e-07
```

Pour plus de précision, on pourrait être tenté de prendre un plus petit arc de cercle, disons entre $\frac{\pi}{3}$ et $\frac{\pi}{2}$, en utilisant encore f. En effet, la fonction n'étant pas dérivable en 1, on a un peu peur que cela crée des perturbations. Et intuitivement, on se dit qu'en limitant l'intervalle d'intégration, on devrait limiter l'ampleur de l'approximation.

```
In[8] for k in range(15) :
        print(pi -
            12*(int_trap(f,0,1/2,2**k)
            - sqrt(3)/8))
...
0.14159265358979312
0.035893249610888134
0.009008728472729821
0.002254493972832172 2
0.000563769720032691 9
```

```
0.0001409515972841113 3
3.5238472716692115e-05
8.8096540218352e-06
2.2024157457778415e-06
5.506040738900708e-07
1.3765102657714579e-07
3.4412758864732496e-08
8.603182166666556e-09
2.1507857717040224e-09
5.37708544356974e-10
```

On a utilisé des segments de droite horizontaux, des segments de droite obliques...Et si l'on utilisait des segments de parabole : intuitivement, cela colle plus à la courbe, cela devrait être plus précis.

3.2.3 Méthode de Simpson

On interpole la courbe par un arc de parabole. On veut que cet arc passe par les points extrêmes de la courbe et le point d'abscisse le milieu de l'intervalle. Pour cela, on va déterminer les c_i tels que :

$$\int_a^b f(x)\mathrm{d}x = c_0 f(a) + c_1 f\left(\frac{a+b}{2}\right) + c_2 f(b)$$

soit exacte pour $f(x)$ successivement égale à 1, x et x^2.
Posons $h = b - a$ et ramenons-nous au cas $a = 0$. On obtient le système suivant :

$$\begin{cases} c_0 + c_1 + c_2 = h \\ c_1 + 2c_2 = h \\ c_1 + 4c_2 = \frac{4}{3}h \end{cases}$$

alors $c_0 = c_2 = \frac{h}{6}$ et $c_1 = \frac{4}{6}h$

$$\int_a^b f(x)\mathrm{d}x = \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right)$$

On subdivise l'intervalle d'intégration et on utilise la méthode de SIMPSON sur chaque subdivision :

```
def int_simps(f,a,b,N) :
    S = 0
    t = a
    dt = (b-a)/N
    while t+dt <= b:
        S += (f(t) + 4*f(t+dt/2) + f(t+dt)) * dt/6
        t += dt
    return S
```

```
In [13]: for k in range(15) :
             print(pi -
                 12*(int_simps(f,0,1/2,2**k)
                 - sqrt(3)/8))
   ....:     ....:
0.0006601149512537319
4.722142667734 63e-05
3.0824728671774437e-06
1.9496909775540416e-07
1.222303502856903e-08
7.645306610015723e-10
4.779110440722434e-11
2.9887203822909214e-12
1.8474111129762605e-13
7.993605777301127e-15
-4.440892098500626e-16
-1.3766765505351941e-14
-8.881784197001252e-15
-4.440892098500626e-16
-1.7763568394002505e-15
```

On est allé un peu trop loin : aller jusqu'à 2^9 suffira.
Résumons la situation :

```
print('-'*75)
print('{:22s} | {:22s} | {:22s}'.format('rectangle gauche', 'trapeze', 'simpson'))
print('-'*75)

for k in range(10) :
    print("{:1.16e} | {:1.16e} | {:1.16e}".format(\
        pi - 12*(int_rec_gauche(f, 0, 1/2, 2**k) - sqrt(3)/8), \
        pi - 12*(int_trap(f, 0, 1/2, 2**k) - sqrt(3)/8), \
        pi - 12*(int_simps(f, 0, 1/2, 2**k) - sqrt(3)/8)))
```

On obtient un beau tableau :

```
---------------------------------------------------------------------------
rectangle gauche       | trapeze                | simpson
---------------------------------------------------------------------------
5.7396688649431091e+00 | 1.4159265358979312e-01 | 6.6011495125373187e-04
2.8349313552875461e+00 | 3.5893249610888134e-02 | 4.7221426677346301e-05
1.4085277813110586e+00 | 9.0087284727298211e-03 | 3.0824728671774437e-06
7.0201402039199667e-01 | 2.2544939728321722e-03 | 1.9496909775540416e-07
3.5044353292961450e-01 | 5.6376972003269188e-04 | 1.2223035028569029e-08
1.7508083320207346e-01 | 1.4095159728411133e-04 | 7.6453066100157230e-10
8.7505179275115363e-02 | 3.5238472716692115e-05 | 4.7791104407224339e-11
4.3743780055221393e-02 | 8.8096540218352004e-06 | 2.9887203822909214e-12
2.1869687616345779e-02 | 2.2024157457778415e-06 | 1.8474111129762605e-13
1.0934293204373891e-02 | 5.5060407389007082e-07 | 7.9936057773011271e-15
```

On distingue mieux cette fois la hiérarchie des méthodes selon leur efficacité.
Nous n'avons pas le choix : il est temps à présent d'invoquer les mathématiques pour préciser un peu ces observations.
En fait, ces trois méthodes sont des cas particuliers d'une même méthode : on utilise une interpolation polynomiale de la fonction f de degré 0 pour les rectangles, de degré 1 pour les trapèzes et de degré 2 pour SIMPSON.
On peut également voir la méthode de SIMPSON comme le premier pas de l'accélération de convergence de ROMBERG de la méthode des trapèzes...mais bon, laissons cela pour plus tard.
Nous allons nous contenter dans un premier temps d'étudier l'erreur commise par la méthode des trapèzes. Nous avons juste besoin du théorème de ROLLE qui peut être étudié en Terminale à titre d'exercice.
On considère une fonction f de classe $\mathcal{C}^2$ sur un intervalle $[a,b]$ à valeurs réelles. On considère une subdivision régulière de $[a,b]$ et on pose :

$$x_j = a + j\frac{b-a}{\mathrm{N}}$$

où N désigne le nombre d'intervalles. On cherche une fonction polynomiale P de degré au plus 1 telle que $\mathrm{P}(a) = f(a)$ et $\mathrm{P}(b) = f(b)$.
L'intégrale de P sur $[a,b]$ vaut $\mathrm{J} = (b-a)\frac{f(a)+f(b)}{2}$.
Soit $x \in]a,b[$. On introduit la fonction g_x définie sur $[a,b]$ par :

$$g_x(t) = f(t) - \mathrm{P}(t) + k_x(t-a)(b-t)$$

où le réel k_x est choisi tel que $g_x(x) = 0$.
La fonction g_x est de classe $\mathcal{C}^2$ et s'annule en a, x et b donc on peut appliquer le théorème de ROLLE sur $]a,x[$ et $]x,b[$ et g_x' s'annule donc sur chacun de ces intervalle. On applique alors une nouvelle fois le théorème de ROLLE à g_x' sur $]a,b[$ et g_x'' s'annule donc au moins une fois en un réel c_x de $]a,b[$.
On pose $k_x = \dfrac{f''(c_x)}{2}$. On obtient finalement que pour tout $x \in]a,b[$, il existe $c_x \in]a,b[$ tel que :

$$f(x) - \mathrm{P}(x) = -(x-a)(b-x)\frac{f''(c_x)}{2}$$

On peut évidemment inclure les bornes de l'intervalle.
Soit M_2 un majorant de $|f''|$ sur $[a,b]$. On obtient donc, par intégration de l'égalité précédente :

$$\left|\int_a^b f(x)\,\mathrm{d}x - \mathrm{J}\right| \leqslant \frac{\mathrm{M}_2}{12}(b-a)^3$$

puis, par application de cette inégalité sur chaque intervalle $[x_j, x_{j+1}]$:

$$\left|\int_a^b f(x)\,\mathrm{d}x - \mathrm{J}_\mathrm{N}\right| \leqslant \frac{\mathrm{M}_2}{12}\frac{(b-a)^3}{\mathrm{N}^2}$$

avec $\mathrm{J}_\mathrm{N} = \frac{b-a}{\mathrm{N}}\left(\frac{f(a)}{2} + \sum_{j=1}^{\mathrm{N}-1} f(x_j) + \frac{f(b)}{2}\right)$.
On obtient donc que quand N est multiplié par 2, le majorant de l'erreur est divisé par 4.
Cela correspond en effet à ce que nous observons avec **Python** :

```
>>> for k in range(1,15) :
    print(pi-12*(int_trap(f,0,1/2,2**k)-sqrt(3)/8),"\t",\
              (pi-12*(int_trap(f,0,1/2,2**(k-1))-sqrt(3)/8))/4)
... ... ...
0.035893249610888134     0.03539816339744828
0.009008728472729821     0.008973312402722033
0.0022544939728321722    0.0022521821181824553
0.0005637697200326919    0.0005636234932080431
0.00014095159728411133   0.00014094243000817297
3.5238472716692115e-05   3.523789932102783e-05
8.8096540218352e-06      8.809618179173029e-06
2.2024157457778415e-06   2.2024135054588e-06
5.506040738900708e-07    5.506039364444604e-07
1.3765102657714579e-07   1.376510184725177e-07
3.4412758864732496e-08   3.4412756644286446e-08
8.603182166666556e-09    8.603189716183124e-09
2.1507857717040224e-09   2.150795541666639e-09
5.37708544356974e-10     5.376964429260056e-10
```

Dans la colonne de gauche se trouvent les résultats déjà calculés. Dans la colonne de droite, le résultat de la ligne i est celui de la ligne $i-1$ de la colonne de gauche divisé par 4 : l'erreur sur le majorant et sur le calcul lui-même est bien corrélée.

On peut démontrer de même qu'avec la méthode de SIMPSON, si f est de classe $\mathcal{C}^4$ sur $[a,b]$, alors le majorant de l'erreur est

$$\frac{M_4}{2880}\frac{(b-a)^5}{N^4}$$

avec M_4 un majorant de $f^{(4)}$ sur $[a,b]$.
Cette méthode subtile, comme celles aussi de ROMBERG ou de GAUSS, sont efficaces pour des fonctions régulières (ici de classe $\mathcal{C}^4$). Pour les fonctions plus « sauvages », la méthode des trapèzes sera souvent meilleure.
Cette fois, on obtient que quand N est multiplié par 2, le majorant de l'erreur est divisé par 16. On observe bien le même phénomène qu'avec la méthode des trapèzes :

```
>>> for k in range(1,9) :
    print(pi-12*(int_simps(f,0,1/2,2**k)-sqrt(3)/8),"\t",\
              (pi-12*(int_simps(f,0,1/2,2**(k-1))-sqrt(3)/8))/16)
... ... ...
4.72214266773463e-05     4.125718445335824e-05
3.0824728671774437e-06   2.951339167334144e-06
1.9496909775540416e-07   1.9265455419859023e-07
1.222303502856903e-08    1.218556860971276e-08
7.645306610015723e-10    7.639396892855643e-10
4.779110440722434e-11    4.778316631259827e-11
2.9887203822909214e-12   2.986944025451521e-12
1.8474111129762605e-13   1.867950238931826e-13
```

Alors, sommes-nous condamnés à ne pas dépasser une précision de 16 décimales ? Pourtant on parle du calcul d'un milliard de décimales de π. Pourtant, les logiciels de calcul formel dépassent eux aussi cette limitation.

Pour cela, il faut repenser totalement la façon de calculer et réfléchir à des algorithmes permettant de passer outre la limitation de représentation des nombres. Mais ceci est une autre histoire...
Moralité, comme aurait pu le dire un contemporain français de NEWTON, *démonstration mathématique et domination de la machine valent mieux qu'observation et utilisation au petit bonheur* ou encore *les mathématiques ne sont pas uniquement une science expérimentale assistée par ordinateur*[1].

3.3 Le nombre π et les arctangentes

3.3.1 Le nombre π et Machin

Python possède un module de calcul en multiprécision avec les quatre opérations arithmétiques de base et la racine carrée :

```
>>> from decimal import*
# on regle la precision a 30 chiffres
>>> getcontext().prec = 30
# on rentre les nombres entre apostrophes, comme des chaines
>>> deux = Decimal('2')
>>> deux.sqrt()
Decimal('1.41421356237309504880168872421')
>>> getcontext().prec = 50
>>> deux.sqrt()
Decimal('1.4142135623730950488016887242096980785696718753769')
```

Mais ça ne va pas nous avancer à grand chose puisque pour avoir 15 bonnes décimales avec la méthode de SIMPSON, il nous a fallu dix millions d'itérations.
Faisons un petit détour par l'histoire...
Tout commence à peu près en 1671, quand l'écossais James GREGORY découvre la formule suivante :

$$\text{Arctan}(x) = \sum_{k=0}^{+\infty} \frac{(-1)^k x^{2k+1}}{2k+1}$$

L'inévitable LEIBNIZ en publie une démonstration en 1682 dans son *Acta Eruditorum* où il est beaucoup question de ces notions désuètes que sont la géométrie et la trigonométrie.
Dans la même œuvre, il démontre ce que nous appelons aujourd'hui le critère spécial des séries alternées :

Soit (u_n) *une suite réelle alternée. Si* $(|u_n|)$ *est décroissante et converge vers 0 alors :*

- $\sum u_n$ *converge ;*
- $|R_n| \leqslant |u_{n+1}|$ *où* (R_n) *est la suite des restes associés à* $\sum u_n$.

On peut même envisager d'évoquer une démonstration de ce théorème en terminale puisqu'on y parle essentiellement de suites adjacentes [voir par exemple Rod10, p. 53].

1. Toute allusion aux nouveaux programmes du lycée ne serait que pure coïncidence...

On en déduit donc que :

$$\left|\text{Arctan}(x) - \sum_{k=0}^{n} \frac{(-1)^k x^{2k+1}}{2k+1}\right| < \frac{x^{2n+3}}{2n+3}$$

On peut alors utiliser **Python** pour déterminer une fonction `mini_greg(a,d)` avec $0 < a < 1$ déterminant une valeur de n telle que $\frac{a^{2n+3}}{2n+3} \leqslant 10^{-d}$:

pi.py

```
def mini_greg(a,d) :
    getcontext().prec = d+2
    n = 1
    ad = Decimal('1') / Decimal(str(a))
    D = Decimal('1e-' + str(d))
    nd = ad**3
    dd = Decimal('3')
    fd = nd/ad
    while fd > D :
        nd *= ad*ad
        dd += 2
        fd = nd/ad
        n += 1
    return n
```

Elle n'est pas très efficace. On peut toutefois remarquer que l'inégalité équivaut à

$$\left(2+\frac{3}{n}\right)\log\frac{1}{a} \geqslant \frac{d}{n} - \frac{\log(2n+3)}{n}$$

Donc pour n grand, le rapport entre le nombre de décimales cherchées et le nombre de termes nécessaires est de l'ordre de $2\log\frac{1}{a}$.
On peut établir les formules d'addition des tangentes puis montrer que, a et b étant deux réels tels que $ab \neq 1$, on a

$$\arctan a + \arctan b = \arctan\left(\frac{a+b}{1-ab}\right) + \varepsilon\pi$$

avec $\varepsilon = 0$ si $ab < 1$, $\varepsilon = 1$ si $ab > 1$ et $a > 0$ et $\varepsilon = -1$ si $ab > 1$ et $a < 0$.
Seul le premier cas va nous intéresser dans ce paragraphe.
En 1706, l'anglais John MACHIN en déduisit la fameuse formule :

$$4\arctan\frac{1}{5} - \arctan\frac{1}{239} = \frac{\pi}{4}$$

ce qui lui permit d'obtenir 100 bonnes décimales de π sans l'aide d'aucune machine mais avec la formule proposée par son voisin écossais.

On appelle polynômes de GREGORY les polynômes $G_n(X) = \sum_{k=0}^{n} \frac{(-1)^k X^{2k+1}}{2k+1}$.

Écrivons une fonction `greg(a,N)` qui donne une valeur de $G_N(a)$:

pi.py

```
def greg(a,N) :
    ad = Decimal('1')/Decimal(str(a))
```

```
    nd = ad
    dd = Decimal('1')
    s = nd/dd
    for k in range(1,N) :
        nd *= Decimal('-1')*ad*ad
        dd += Decimal('2')
        s += nd/dd
    return s
```

Il ne reste plus qu'à l'utiliser dans la formule de MACHIN :

pi.py

```
def pi_machin(d) :
    getcontext().prec = d+2
    rap = 2*log(5)/log(10)
    N = int(d/rap)+1
    return 4*(4*greg(5,N)-greg(239,N))
```

Ce n'est pas optimal car on lance deux fois `greg`.
On récupère une approximation de π quelque part pour vérification et on compare juste pour le plaisir (enfin pas seulement car on a un peu peur des erreurs d'arrondi !) :

```
In [212]: PI - pi_machin(1000)
Out[212]: Decimal('-2.119047427...9901E-1000')
```

En une seconde et demie, on a mille bonnes décimales de π : les voici...

```
In [213]: pi_machin(1000)
Out[215]: Decimal('3.14159265358979323846264338327 9...
...27876611195909216420199 15')
```

Pour d'autres formules avec des arctangentes, on pourra se référer à un vieil article de la revue PLOT [Bri91].

3.3.2 Détour par l'arithmétique : « dissection » d'un arctangente

On a noté que plus le plus petit dénominateur d'une formule à la MACHIN sera grand, moins il faudra d'itérations (vous avez plus ou moins suivi ?).
On veut pouvoir transformer un $\arctan\frac{1}{n}$ en un $\arctan\frac{1}{a}+\arctan\frac{1}{b}$ avec, on espère, a et b plus grands que n.
Or, en manipulant la fameuse formule d'addition des tangentes :

$$\tan(a+b)=\frac{\tan(a)+\tan(b)}{1-\tan(a)\tan(b)}$$

on démontre que, sous nos conditions de température et de pression, comme déjà vu page précédente :

$$\arctan\frac{1}{a}+\arctan\frac{1}{b}=\arctan\left(\frac{a+b}{ab-1}\right)$$

puis à :

$$n(a+b)+1=ab$$

On relie une somme et un produit de a et de b. Il fut un temps (maintenant très reculé) où un(e) élève de lycée en déduisait qu'il fût judicieux d'introduire un polynôme quadratique dont a et b fussent les racines :

$$n^2+1 = n^2 - n(a+b) + ab$$

Finalement, la dissection de l'arctangente de $\frac{1}{n}$ revient à chercher a et b tels que :

$$n^2+1 = (a-n)(b-n)$$

C'est donc un problème de recherche des diviseurs de l'entier n^2+1.
Python peut bien sûr nous y aider :

pi.py

```
def dissec(n) :
    N = n*n+1
    L = [[1,N],[-1,-N]]
    if N%2 == 0:
        L += [[2,N//2],[-2,-N//2]]
    else:
        for k in range(3,N//2,2) :
            if N%k == 0:
                L += [[k,N//k],[-k,-N//k]]
    for j in range(len(L)) :
        if not(-n in L[j]) :
            print('Arctan 1/',n,'= Arctan 1/',L[j][0]+n,' + Arctan
                1/',L[j][1]+n,end='\n')
```

Par exemple, on a :

$$\frac{\pi}{4} = \arctan(1)$$

Or :

```
In [225]: dissec(1)
Arctan 1/ 1 = Arctan 1/ 2  + Arctan 1/ 3
Arctan 1/ 1 = Arctan 1/ 3  + Arctan 1/ 2
```

donc on en déduit, comme HUTTON en 1776 :

$$\frac{\pi}{4} = \arctan\frac{1}{2} + \arctan\frac{1}{3}$$

puis, comme :

```
In [226]: dissec(2)
Arctan 1/ 2 = Arctan 1/ 3  + Arctan 1/ 7
Arctan 1/ 2 = Arctan 1/ 1  + Arctan 1/ -3
```

on en déduit deux autres formules dues à HUTTON :

$$\frac{\pi}{4} = 2\arctan\frac{1}{3} + \arctan\frac{1}{7} = 2\arctan\frac{1}{2} - \arctan\frac{1}{7}$$

et on peut continuer ainsi très longtemps..

On peut même s'amuser à « trisséquer » les arctangentes sous la forme :

$$\arctan\frac{1}{n} = \arctan\frac{1}{a} + \arctan\frac{1}{b} + \arctan\frac{1}{c}$$

avec des considérations similaires pour arriver à la condition :

$$(n^2+1)(a+b+c-n) = (a-n)(b-n)(c-n)$$

et c'est parti pour de nouvelles formules...
En voici quelques-unes de célèbres :

- $12\arctan\frac{1}{18} + 8\arctan\frac{1}{57} - 5\arctan\frac{1}{239}$ (GAUSS) ;
- $6\arctan\frac{1}{8} + 2\arctan\frac{1}{57} + \arctan\frac{1}{239}$ (STÖRMER - 1896).

3.3.3 Calculs

On s'aperçoit vite que sous cette forme, nos algorithmes peuvent nous fournir quelques milliers de décimales mais à partir de dix mille, il faut commencer à être vraiment patient.
On peut tenter d'accélérer un peu les choses en ne relançant pas plusieurs fois `greg`.
On peut également généraliser la fonction pour l'adapter à toutes les formules.

pi.py

```
def greg(p,C,A) :
    rap = 2 * log10(A[0])
    N = int(p/rap) + 1
    s = (-1)**N
    getcontext().prec = p + 2
    cx = [ s * Decimal(str(C[j]))   for j in range(len(C)) ]
    ax = [ Decimal(str(A[j]))       for j in range(len(C)) ]
    X  = [ cx[j] / ax[j]**(2*N + 1) for j in range(len(C)) ]
    kx = [ ax[j] * ax[j]            for j in range(len(C)) ]
    S  = sum(X) / (2*N + 1)
    for k in range(2*N - 1, 0, -2) :
        X  = [ -X[j] * kx[j] for j in range(len(C)) ]
        S += sum(X) / Decimal(str(k))
    return S
```

Quelques explications :

- on peut montrer que le rapport entre le nombre de décimales désirées et le nombre de termes de la somme nécessaires est de l'ordre de $2\log_{10}(a)$ avec a égal au plus petit dénominateur intervenant dans la formule ;
- on rentre les coefficients multiplicateurs dans la liste `C` (en n'oubliant pas de les multiplier par 4) ;
- on rentre les dénominateurs dans la liste `A` ;
- on les transforme en « `Decimal` » dans `cx` et `ax` ;
- on va sommer par ordre décroissant des puissances ;
- comme on va de 2 en 2 pour les puissances, on multiplie à chaque itération par le carré des « x ».

Comparons les formules après avoir chargé les modules `math` et `time` :

- pour la formule de MACHIN :

```
In [233]: %timeit greg(1000,[16,-4],[5,239])
100 loops, best of 3: 6.71 ms per loop
```

- pour dix mille décimales avec la formule de GAUSS :

```
In [234]: %timeit greg(10000,[48,32,-20],[18,57,239])
1 loops, best of 3: 238 ms per loop
```

3.4 Méthodes modernes

En 1975, Richard BRENT et Eugene SALAMIN mettent au point un algorithme qui leur permet de calculer en 1999 plus de 200 milliards de décimales de π.
Il est basé sur des travaux de GAUSS et de LEGENDRE et sur les intégrales elliptiques. On se reportera à l'article que Jean-Pierre DEMAILLY consacre au sujet [Dem08].
Le principe est très simple :

$$a_0 = 1 \qquad b_0 = \frac{1}{\sqrt{2}} \qquad s_0 = \frac{1}{2}$$

puis :

$$a_{n+1} = \frac{a_n + b_n}{2}, \quad b_{n+1} = \sqrt{a_n b_n}, \quad c_{n+1} = a_{n+1}^2 - b_{n+1}^2, \quad s_{n+1} = s_n - 2^{n+1} c_{n+1}.$$

Alors, $p_N = \frac{2a_N^2}{s_N}$ est une approximation de π.

pi.py

```
def brent(n) :
    d = Decimal('2')
    a = Decimal('1')
    u = Decimal('0.5')
    s = u
    b = s*d.sqrt()
    c, k = a*a - b*b, 1
    for j in range(n) :
        a, b = (a + b)*u, (a*b).sqrt()
        c, k = a*a - b*b, 2*k
        s    = s - c*k
    return d*a*a / s
```

En une seconde, on a presque 6 000 décimales de π et on double le nombre de bonnes décimales à chaque itération...

```
In [242]: PI - brent(15)
Out[242]: Decimal('-3.4252257719542....3684791E-6020')

In [243]: %timeit brent(15)
1 loops, best of 3: 718 ms per loop
```

3.5 Algorithme de Plouffe

Le Canadien Simon PLOUFFE est mondialement connu pour son algorithme de calcul de la n-ième décimale de π[2].
Voici quelques éléments en `Cython` pour aller vite (18 s pour le dix-millionième caractère du développement en base 16 de π) en n'oubliant pas au préalable de prévenir ipython :

```
In [1]: %load_ext cythonmagic
```

puis il suffira de coller les lignes suivantes avec `%paste` :

plouffe.py

```
%%cython -a

cimport cython

cdef float mod_exp(int a, int b, int m):
    cdef int bb, p
    cdef long aa
    aa = a
    bb = b
    p = 1
    while bb > 0:
        p = cython.cmod(p * (aa**(bb & 1 == 1)) , m)
        aa = cython.cmod(aa * aa,m)
        bb = bb >> 1
    return p

@cython.cdivision(True)
cdef float s1part(int n, int k, int j):
    cdef float nu, de
    nu = mod_exp(16, n - 1 - k, 8*k + j)
    de = (k * 8.0 + j)
    return nu / de

@cython.cdivision(True)
cdef float s2part(int n, int k, int j):
    cdef float de
    de = (((k << 3) + j) << ((k + 1 - n) << 2))
    return 1.0 / de

cdef float s1(int n):
    cdef float s
    cdef int k
    s = 0.0
    for k from 0 <= k < n:
        s = s + 4.0*s1part(n,k,1) - 2.0*s1part(n,k,4) - 1.0*s1part(n,k,5) -
            1.0*s1part(n,k,6)
    return s
```

2. https://fr.wikipedia.org/wiki/Formule_BBP

```
cdef float s2(int n):
    cdef float s
    cdef int k
    s = 0.0
    for k from n <= k < n + 5:
        s = s + 4.0*s2part(n,k,1) - 2.0*s2part(n,k,4) - 1.0*s2part(n,k,5) -
            1.0*s2part(n,k,6)
    return s

cdef extern from "math.h":
    float floor(float r)

cdef float prem_hexa(float r):
        cdef float x
        x = r - floor(r)
        return floor(16*x)


def pi_plouffe(int n):
    cdef float s
    s = s1(n) + s2(n)
    return hex(int(prem_hexa(s)))[2:]
```

Voici le temps de calcul pour obtenir le dix-millionième caractère du développement en base 16 de π :

```
In [15]: pi_plouffe(10000000)
Out[15]: '8'

In [16]: %timeit pi_plouffe(10000000)
1 loops, best of 3: 18.5 s per loop
```

Et les dix-mille premiers caractères du développement de π en base 16 :

```
In [40]: ''.join([pi_plouffe(k) for k in range(10000)])
Out[40]:
'3243f6a8885a308d313198a2e03707344a4093822299f31d0082efa98ec4e6c89452821e638d01377be
5466cf34e90c6cc0ac29b7c97c50dd3f84d5b5b54709179216d5d98979fb1bd1310ba698dfb5ac2ffd72
...
c63ad456edf5f457a814551875a64cd3099f169b5f18a8c73ee0b5e57368f6c79f4bb7a595926aab49ec'
```


4 Probabilités

Remarque. On utilisera dans toute cette section le module `numpy.random` et non pas le module `random` qui est beaucoup plus lent. Attention ! Avec le premier, la borne supérieure n'est pas comprise alors qu'elle l'est avec le second.

4.1 Pile ou face ?

On lance trois fois de suite une pièce de monnaie. On compte combien de fois pile (ou face) tombe :

probas.py

```
def piece(n):
    T = []
    for k in range(n):
        X = 0
        for j in range(3):
            X += randint(0,2)
        T.append(X)
    return([ (T.count(i) / float(n)) for i in range(0,4) ])
```

sachant que :

- `randint(1,2)` renvoie un entier aléatoirement compris entre 1 et 2 ;
- `T.count(j)` compte le nombre de fois ou `j` apparaît dans la liste `T`.

Par exemple, pour un million de lancers :

```
In [250]: piece(1000000)
Out[250]: [0.124758, 0.375741, 0.374742, 0.124759]
```

En effet, $\binom{3}{0}\left(\frac{1}{2}\right)^0\left(\frac{1}{2}\right)^3 = \frac{1}{8} = 0,125$ et $\binom{3}{1}\left(\frac{1}{2}\right)^1\left(\frac{1}{2}\right)^2 = \frac{1}{8} = 0,375$.

4.2 Tirage de boules dans des urnes

On peut adapter sans problème ce programme à la situation classique suivante : on dispose de trois urnes, la première contenant 7 boules blanches et 4 noires, la deuxième 5 blanches et 2 noires, la troisième 6 blanches et 3 noires.
On tire une boule dans chaque urne et on note le nombre de boules blanches obtenues.

probas.py

```
def boules(n):
    T = []
    for k in range(n):
        X = 0
        X += randint(1,12) <= 7
        X += randint(1,8)  <= 5
        X += randint(1,10)  <= 6
        T.append(X)
    return([ ( T.count(j) / float(n) ) for j in range(0,4) ])
```

et on obtient par exemple pour un million de simulations :

```
In [252]: boules(1000000)
Out[252]: [0.03488, 0.217687, 0.444994, 0.302439]
```

En effet, par exemple, $\frac{7}{11} \times \frac{5}{7} \times \frac{6}{9} \approx 0,303$ et $\frac{4}{11} \times \frac{2}{7} \times \frac{3}{9} \approx 0,034$.

4.3 Tirage de boules avec remise

Simulons le tirage successif de quatre boules avec remise dans une urne contenant 7 boules blanches et 3 boules noires. Comptons la fréquence des tirages contenant :

- exactement deux boules blanches ;
- au moins une boule blanche.

probas.py

```
def boule(n,nb):
    X=0
    for k in range(n):
        B = 0
        for j in range(1,6):
            B += randint(1,11) <= 7
        X += B == nb
    return X / n
```

Alors pour 100 000 tirages, on obtient pour deux boules blanches :

```
In [254]:  boule(1000000,2)
Out[254]: 0.264975
```

En effet, le tirage se fait sans remise. La probabilité d'obtenir (N, N, B, B) est :

$$\frac{3}{10} \times \frac{3}{10} \times \frac{7}{10} \times \frac{7}{10} = \frac{441}{10\,000}$$

Les tirages de deux boules exactement sont les anagrammes de (N, N, B, B).
On multiplie donc le résultat précédent par $\frac{4!}{2!\times 2!} = 6$.

$$6 \times \frac{441}{10\,000} \approx 0{,}264\,6$$

Obtenir au moins une boule blanche est le contraire de n'en obtenir aucune :

```
In [255]: 1-boule(1000000,0)
Out[255]: 0.992068
```

En effet

$$1 - \left(\frac{3}{10}\right)^4 \approx 0{,}991\,9$$

4.4 Problème du Duc de Toscane

Cosme II de Médicis (Florence 1590-1621), Duc de Toscane, fut le protecteur de l'illustre Galilée (né à Pise le 15 février 1564 et mort à Florence le 8 janvier 1642) son ancien précepteur. Profitant d'un moment de répit du savant entre l'écriture d'un théorème sur la chute des corps et la création de la lunette astronomique, le Grand Duc lui soumet le problème suivant : il a observé qu'en lançant trois dés cubiques et en faisant la somme des numéros des faces, on obtient plus souvent 10 que 9, alors qu'il y a autant de façons d'obtenir 9 que 10, à savoir six. Après quelques réflexions, Galilée rédigea un petit mémoire sur les jeux de hasard en 1620 expliquant le phénomène.

probas.py

```
def toscane(n):
    T = n*[0]
    for k in range(n):
        T[k] = randint(1,6) + randint(1,6) + randint(1,6)
    return 'Obtenir 9 :' + str(T.count(9)*100.0/n) + '%, Obtenir 10 :' +
        str(T.count(10)*100.0/n) + '%'
```

Pour un million de simulations :

```
In [279]: toscane(1000000)
Out[279]: 'Obtenir 9 :11.5296%, Obtenir 10 :12.4879%'
```

On peut préférer comparer les données expérimentales aux données théoriques.

probas.Py

```
from numpy.random import randint
from collections  import Counter
from itertools import product

nObs = 144000

# séries d'observations de séries de lancers de 3 dés
sObs  = lambda n : [sum( randint(1,7,size = n) )  for k in range(nObs)]
sTheo = lambda n : [sum(t) for t in product(range(1,7), repeat = n)]

# dictionnaire des fréquences de ces séries
# dico.items() renvoie la liste des couples (clé,valeur) de dico
def pObs(n) :
    s = sObs(n)
    return {x: 100*v/len(s)  for x,v in Counter(s).items()}

def pTheo(n):
    s = sTheo(n)
    return {x: 100*v/len(s) for x,v in Counter(s).items()}
```

```
In [6]: pObs(3)
Out[6]:
{3: 0.5027777777777778,
 4: 1.363888888888889,
 5: 2.7840277777777778,
 6: 4.572916666666667,
 7: 6.838888888888889,
 8: 9.699305555555556,
 9: 11.861805555555556,
 10: 12.506944444444445,
 11: 12.460416666666667,
 12: 11.565277777777778,
 13: 9.696527777777778,
 14: 6.9847222222222225,
 15: 4.625694444444444,
 16: 2.682638888888889,
 17: 1.3597222222222223,
 18: 0.49444444444444446}

In [7]: pTheo(3)
Out[7]:
{3: 0.46296296296296297,
 4: 1.3888888888888888,
 5: 2.7777777777777777,
 6: 4.62962962962963,
 7: 6.944444444444445,
 8: 9.722222222222221,
 9: 11.574074074074074,
```

```
10: 12.5,
11: 12.5,
12: 11.574074074074074,
13: 9.722222222222221,
14: 6.944444444444445,
15: 4.62962962962963,
16: 2.7777777777777777,
17: 1.3888888888888888,
18: 0.46296296296296297}
```

4.5 Lancers d'une pièce et résultats égaux

On lance 10 fois de suite une pièce de monnaie et l'on s'intéresse au nombre maximal de résultats consécutifs égaux. On crée un programme qui simule autant de séries de lancers que l'on désire.

probas.py

```
def piece_max(n):
    S = n*[1]
    for k in range(n):
        s = []
        P = [randint(0, 2) for z in range(10)]
        p = 0
        while p < 9:
            j = p + 1
            while (P[j] == P[p] and j < 9):
                j += 1
            s . append(j - p)
            p = j
        s.sort(reverse=True)
        S[k] = s[0]
    m = sum(S) / n
    return 'Sur '+str(n)+'  séries de 10 lancers, la moyenne du nombre maximal
        de résultats consécutifs égaux est '+str(m)
```

- On crée une liste S remplie de n zéros pour y mettre les maxima ;
- on boucle les essais ;
- on crée une liste s enregistrant le nombre de résultats consécutifs égaux ;
- on crée une liste P de 10 lancers de 0 ou 1 ;
- tant qu'on n'est pas au bout de la liste p<9, on regarde le suivant (j=p+1)
- tant que le suivant est égal, on continue (j+=1) ;
- on rajoute le nombre de lancers égaux et on enlève p car on a commencé à p+1 (s+=[j-p]) ;
- on va ensuite voir le prochain résultat différent (p=j) ;
- on classe dans l'ordre décroissant les résultats (s.sort(reverse=True)) ;
- on prend le premier de la liste : c'est le plus grand (S[k]=s[0]) ;

On obtient :

```
In [286]:  piece_max(100000)
Out[286]: 'Sur 100000  séries de 10 lancers, la moyenne du nombre maximal de
    résultats consécutifs égaux est 3.52318'
```

4.6 Le Monty Hall

Le jeu oppose un présentateur à un candidat. Ce joueur est placé devant trois portes fermées. Derrière l'une d'elles se trouve une voiture et derrière chacune des deux autres se trouve une chèvre. Il doit tout d'abord désigner une porte. Puis le présentateur ouvre une porte qui n'est ni celle choisie par le candidat, ni celle cachant la voiture (le présentateur sait quelle est la bonne porte dès le début). Le candidat a alors le droit ou bien d'ouvrir la porte qu'il a choisie initialement, ou bien d'ouvrir la troisième porte.
Les questions qui se posent au candidat sont :

- Que doit-il faire ?
- Quelles sont ses chances de gagner la voiture en agissant au mieux ?

On peut avoir deux idées contradictoires :

- le présentateur ayant ouvert une porte, le candidat a une chance sur deux de tomber sur la voiture ;
- Au départ, le candidat a une probabilité de $\frac{1}{3}$ de tomber sur la voiture. Ensuite, s'il garde son choix initial, il a toujours la même probabilité $\frac{1}{3}$ de gagner donc s'il change d'avis, la probabilité qu'il gagne est donc $1 - \frac{1}{3} = \frac{2}{3}$.

Qui a raison ?

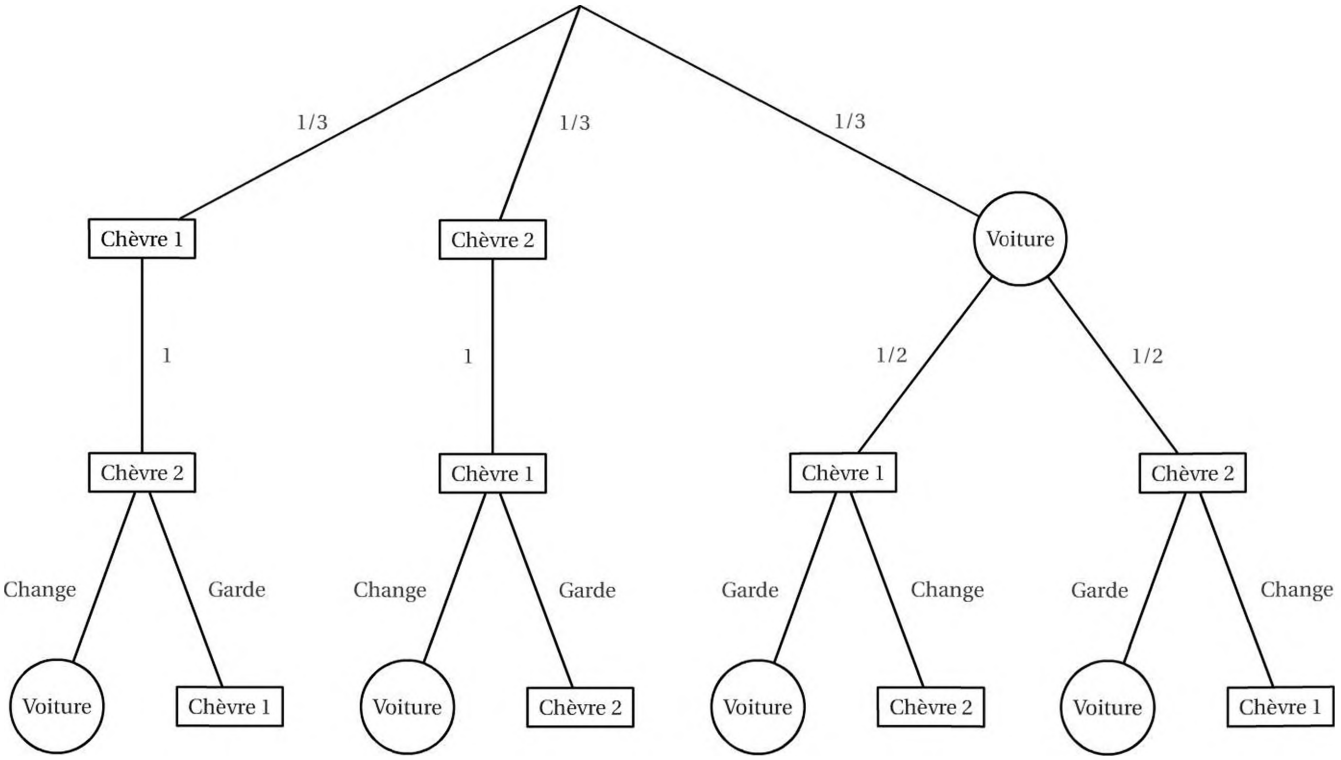
probas.py

```
def monty(n):
    gagne_sans_changer = 0
    gagne_en_changeant = 0
    for j in range(n):
        voiture = randint(0,2)
        choix   = randint(0,2)
        if choix == voiture:
            ouverte = (voiture + 1 + randint(0,1)) % 3
        else:
            ouverte = 0 + 1 + 2 - choix - voiture
        changement = 0 + 1 + 2 - choix - ouverte
        if choix == voiture:
            gagne_sans_changer += 1
        if changement == voiture:
            gagne_en_changeant += 1
    return 'Gagne en changeant : '+str(100.*gagne_en_changeant/n)+'%   Gagne
        sans changer : '+ str(100.*gagne_sans_changer/n)+'%'
```

Alors, pour 100 000 expériences :

```
In [288]: monty(1000000)
Out[288]: 'Gagne en changeant : 66.7266%   Gagne sans changer : 33.2734%'
```

La deuxième option semble la bonne...il reste à le prouver !
Dressons un arbre en trois étapes :



- Le candidat choisit d'abord une porte de manière équiprobable ;
- Le présentateur, sachant où se trouve la voiture, choisit une porte cachant une chèvre ;
- Le candidat prend ensuite la décision de garder ou de changer son choix initial.

Nous en déduisons que :

- s'il conserve son choix initial, la probabilité que le candidat gagne la voiture est :

$$\frac{1}{3} \times \frac{1}{2} + \frac{1}{3} \times \frac{1}{2} = \frac{1}{3}$$

- s'il change son choix initial, la probabilité que le candidat gagne la voiture est :

$$\frac{1}{3} \times 1 + \frac{1}{3} \times 1 = \frac{2}{3}$$

Il vaut donc mieux pour le candidat changer son choix initial.

4.7 Exercice d'entraînement

Le corrigé est disponible sur le site dunod.com à partir de la page d'accueil de l'ouvrage.

Exercice 48. (Le lièvre et la tortue) On rappelle le jeu bien connu : on dispose d'un dé cubique. On le lance. Si un six sort, le lièvre gagne sinon la tortue avance d'une case. La tortue a gagné lorsqu'elle a avancé six fois de suite. Quelle est la probabilité que la tortue gagne ?

5 Relations binaires et graphes

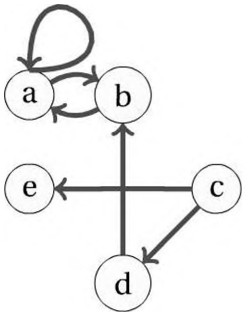
5.1 Relations binaires sur un ensemble

5.1.1 Définition et représentations

Rappelons ce qu'est une relation binaire sur un ensemble :
On appelle relation binaire sur l'ensemble E *tout couple* $(E, G_{\mathcal{R}})$ *où* $G_{\mathcal{R}}$ *est une partie de* $E \times E$.
Notons $\mathcal{R}$ *(ou* R *ou* f *ou ...) ce couple :*

$$\mathcal{R} = (E, G_{\mathcal{R}})$$

On peut représenter une relation par un diagramme sagittal ou un dictionnaire :



Sommet	en relation avec
a	$\{a, b\}$
b	$\{a\}$
c	$\{d, e\}$
d	$\{b\}$
e	$\{\}$

5.1.2 Représentation sur Python

Avec **Python**, on peut utiliser également un dictionnaire qui est un type de variable présent (cf le chapitre 1) :

```
R = {'a' : set(['a','b']),
     'b' : set(['a']),
     'c' : set(['d','e']),
     'd' : set(['b']),
     'e' : set([])
    }
```

Les sommets en relations avec un sommet donné sont donnés dans un ensemble (de type **set**) ce qui induira les opérateurs permettant de le manipuler :

```
>>> R['a']
set(['a', 'b'])
>>> R['a'] | R['c']
set(['a', 'b', 'e', 'd'])
>>> 'b' in R['d']
True
```

Voici quelques méthodes utiles pour manipuler les dictionnaires :

```
>>> list(R)
['a', 'c', 'b', 'e', 'd']
>>> len(R)
5
>>> for u in R:
    print u
```

```
a c b e d
>>> for u in R:
    for v in R[u]:
        print u,v
a a   a b   c e   c d   b a   d b
```

Il faudra garder en tête que l'affectation de structures complexes sous **Python** est une copie d'adresses : il faudra donc utiliser `deepcopy` du module `copy` pour copier un dictionnaire *en profondeur* :

```
>>> B = R
>>> id(B)
147845164
>>> id(R)
```

```
147845164
>>> C = deepcopy(R)
>>> id(C)
149979180
```

Nous sommes maintenant prêts à manipuler les relations binaires sur un ensemble.

5.1.3 Composition de relations

Le graphe de $\mathcal{S} \circ \mathcal{R}$ est l'ensemble des couples (x, y) de E tels qu'il existe au moins un z dans E tel que $x\mathcal{R}z$ et $z\mathcal{S}y$.
On appelle aussi cette relation « $\mathcal{R}$ suivi de $\mathcal{S}$ ».
Voici une définition possible sur **Python** :

graphes.py

```
def compose(R,S):
    """ R suivi de S """
    C = {} # Le dictionnaire de la composée est vide au départ
    for u in R: # pour chaque sommet u de E
        C[u] = set([]) # u n'est encore en relation avec personne par C
        for v in R[u]: # pour chaque v tel que uRv
            C[u] |= S[v] # uCw <-> vSw
    return C
```

On aura besoin de définir sur quel ensemble est définie la relation et la relation identité sur cet ensemble :

graphes.py

```
def ens(R):
    return set(list(R))

def id(E):
    D = {}
    for u in E: # chaque sommet u vérifie uRu
        D[u] = set([u])
    return D
```

On en déduit une définition récursive de la composition itérée de d'une relation :

graphes.py

```
def iter_compo(R,n):
    """ R^n version récursive """
    if n == 0:
        return id(ens(R)) # R^0 = id
    else: # sinon R^n = R.R^(n-1)
        return compose(R,iter_compo(R,n-1))
```

5.1.4 Propriétés d'une relation

On vérifie si une relation est réflexive, c'est-à-dire que chaque élément est en relation avec lui-même :

graphes.py

```
def est_reflexive(R):
    for u in R:
        if not(u in R[u]): # dès qu'on trouve un contre-exemple, on sort
            return False
    return True # si on n'est pas déjà sorti c'est que la relation est réflexive
```

On vérifie qu'une relation est symétrique c'est-à-dire que $u\mathcal{R}v \Longrightarrow v$:

graphes.py

```
def est_symetrique(R):
    for u in R:
        if u not in [R[v] for v in R[u]]:
            return False
    return True
```

On a utilisé une liste par compréhension pour plus d'efficacité.
Le test d'antisymétrie s'en déduit :

graphes.py

```
def est_antisymetrique(R):
    for u in R:
        if u in [R[v] for v in R[u] if v != u]:
            return False
    return True
```

Le `v != u` permet de ne pas exclure les cas « $u\mathcal{R}u$ ».
Pour la transitivité, on utilisera le résultat classique :
La relation $\mathcal{R}$ est transitive si, et seulement si, $\mathcal{R}^2 \subseteq \mathcal{R}$.
On notera que l'inclusion des ensemble se note `<=` :

graphes.py

```
def est_transitive(R):
    R2 = compose(R,R)
    for u in R2:
        if not(R2[u] <= R[u]):
            return False
    return True
```

On rappelle que la fermeture transitive d'une relation est notée $\mathcal{R}^+$ et est définie par :

$$\mathcal{R}^+ = \bigcup_{k=1}^{|E|} \mathcal{R}^k$$

On fera bien attention à l'utilisation de `deepcopy` :

graphes.py

```
def r_plus(R):
    Rp = deepcopy(R)
    for k in range(1,len(Rp)+1):
        Rk = iter_compo(R,k)
        for u in Rp:
            Rp[u] |= Rk[u]
    return Rp
```

Il s'agit de la plus petite relation transitive contenant $\mathcal{R}$ au sens de l'inclusion.

5.2 Graphes

Les graphes sont étudiés en utilisant la programmation orientée objet dans le chapitre consacré aux classes (cf page page 236).

Nous étudierons ici d'autres algorithmes et sans utiliser de classes.

5.2.1 Recherche de plus court chemin

Intéressons-nous tout d'abord à l'algorithme de BELLMAN-FORD, plus efficace et général mais plus facile à programmer.

On considère un graphe valué sans circuit de valeur négative (ou circuit absorbant) mais ayant éventuellement des arcs pondérés négativement.

On s'intéresse à la distance d'un sommet particulier, la source, aux différents sommets du graphe.

On note $\pi(v)$ la distance de la source s à un sommet v.

Au départ, on pose $\pi(s) = 0$ et pour tous les autres sommets du graphe, $\pi(v) = +\infty$.

L'idée est que, pour tout sommet v, si passer par un prédécesseur u de v raccourcit la distance, alors on remplace $\pi(v)$ par $\pi(u) + p(u, v)$ avec $p(u, v)$ la pondération de l'arc (u, v).

La propriété de BELLMAN est donc la suivante :

$$\pi(v) = \min_{u \in \Gamma^{-1}(v)} \{\pi(u) + p(u, v)\}$$

Algorithme Bellman
$\pi(s) = 0$
Pour tous les sommets du graphe $\neq s$ **Faire**
  $\pi(s) \leftarrow +\infty$
FinPour
TantQue π change **Faire**
  Pour chaque arc (u, v) de G **Faire**
    Si $\pi(v) > \pi(u) + p(u, v)$ **Alors**
      $\pi(v) \leftarrow \pi(u) + p(u, v)$
    FinSi
  FinPour
FinTantQue
Retourner π

Comme pour les relations, les graphes seront considérés comme des dictionnaires. Dans le cas qui nous occupe, il s'agit de graphes valués : nous les programmerons donc comme des dictionnaires dont les sommets font référence à un « sous-dictionnaire ».

```
g = {'a': {'b': 4, 'c': 2},
     'b': {'a' : 4,'d': 5, 'c': 1},
     'c': {'a' : 2, 'b' : 1, 'd' : 8,'e': 10},
     'd': {'b': 5, 'c': 8, 'e': 2},
     'e': {'c': 10, 'd' : 2, 'f': 3},
     'f': {'e' : 3,'d' : 6}}
```

Ainsi, `g['a']` est lui-même un dictionnaire :

```
>>> g['a']
{'c': 2, 'b': 4}
>>> g['a']['b']
4
>>> 'c' in g['a']
True
```

Voici une traduction quasi directe de l'algorithme en langage **Python** qui utilise la méthode `deepcopy` :

graphes.py

```
from copy import deepcopy

def bellman_ford(graphe, source):
    """ Programme direct sans les chemins """
    pi = {}
    for s in graphe:
        pi[s] = float('inf')
    pi[source] = 0
    pi0 = {}
    k = 1
```

```
    while pi != pi0 :
        pi0 = deepcopy(pi)
        for u in graphe:
            for v in graphe[u]:
                if pi[v] > pi[u] + graphe[u][v]:
                    pi[v]  = pi[u] + graphe[u][v]
        k+=1
    return pi
```

On a utilisé la possibilité de calculer avec « l'infini » qui est en fait programmé pour être un flottant suivant les propriétés du calcul dans $\overline{\mathbb{R}}$. On le note `float('inf')`.

Rappelons à présent l'algorithme de DIJKSTRA qui permet lui aussi de déterminer les plus courts chemins dans un arbre pondéré, mais uniquement dans le cas d'arcs pondérés positivement.

Algorithme DIJKSTRA
Début
- $pi \leftarrow$ dictionnaire vide
- π[début]=0
- Visités ← liste vide
- Chaîne ← [début]
- x ← début
- **Pour** tout s de X \{début} **Faire**
  - π(s) ← $+\infty$
  - chaîne ← a
- **FinPour**
- **TantQue** x ≠ fin **Faire**
  - **Pour** v ∉ Visités et voisin de x **Faire**
    - π[v] ← min $\{\pi[v], \pi[x] + p(x, v)\}$
  - **FinPour**
  - Extraire un sommet $u \notin$ Visités tel que $\pi[u] = \min\{\pi[y], y \notin \text{Visités}\}$
  - x ← u
  - Visités ← Visités $\cup\{u\}$
- **FinTantQue**
- **Retourner** π

Fin

Python permet de gagner un peu de temps grâce à des petites particularités de traitement des dictionnaires. Voici même une version récursive de l'algorithme.
Pour trouver la clé correspondant à la valeur minimale d'un dictionnaire, on utilise une syntaxe très pythonesque. Par exemple, pour avoir le voisin le plus proche de 'a' :

```
>>> min(g['a'],key=g['a'].get)
'c'
```

De plus **dict**.get(cle,defaut) retourne la valeur de cle si elle se trouve dans le dictionnaire et defaut sinon.

On procède en trois étapes :

graphes.py

```
def affiche_peres(pere,depart,extremite,trajet):
    """
    À partir du dictionnaire des pères de chaque sommet on renvoie
    la liste des sommets du plus court chemin trouvé. Calcul récursif.
    On part de la fin et on remonte vers le départ du chemin.

    """
    if extremite == depart:
        return [depart] + trajet
    else:
        return (affiche_peres(pere, depart, pere[extremite], [extremite] +
            trajet))

def plus_court(graphe,etape,fin,visites,dist,pere,depart):
    """
    Recherche de la plus courte chaine entre debut et fin avec l'algo de Dijkstra
    visites est une liste et dist et pere des dictionnaires
    graphe  : le graphe étudié  (dictionnaire)
    étape : le sommet en cours d'étude (sommet)
    fin     : but du trajet (sommet)
    visites : liste des sommets déjà visités (liste de sommets)
    dist    : dictionnaire meilleure distance actuelle entre départ et les
        sommets du graphe (dict sommet : int)
    pere    : dictionnaire des pères actuels des sommets correspondant aux
        meilleurs chemins (dict sommet : sommet)
    depart  : sommet global de départ (sommet)
    Retourne le couple (longueur mini (int), trajet correspondant (liste
        sommets))

    """
    # si on arrive à la fin, on affiche la distance et les peres
    if etape == fin:
       return dist[fin], affiche_peres(pere,depart,fin,[])
    # si c'est la première visite, c'est que l'étape actuelle est le départ : on
        met dist[etape] à 0
    if  len(visites) == 0 : dist[etape]=0
    # on commence à tester les voisins non visités
    for voisin in graphe[etape]:
        if voisin not in visites:
            # la distance est soit la distance calculée précédemment soit
                l'infini
            dist_voisin = dist.get(voisin,float('inf'))
            # on calcule la nouvelle distance calculée en passant par l'étape
            candidat_dist = dist[etape] + graphe[etape][voisin]
            # on effectue les changements si cela donne un chemin plus court
```

```
            if candidat_dist < dist_voisin:
                dist[voisin] = candidat_dist
                pere[voisin] = etape
    # on a regardé tous les voisins : le noeud entier est visité
    visites.append(etape)
    # on cherche le sommet *non visité* le plus proche du départ
    non_visites = dict((s, dist.get(s,float('inf'))) for s in graphe if s not in
        visites)
    noeud_plus_proche = min(non_visites, key = non_visites.get)
    # on applique récursivement en prenant comme nouvelle étape le sommet le
        plus proche
    return plus_court(graphe,noeud_plus_proche,fin,visites,dist,pere,depart)


def dij_rec(graphe,debut,fin):
    return plus_court(graphe,debut,fin,[],{},[],1)
```

5.2.2 Coloration de graphes

Colorier les sommets du graphe simple non orienté G = (X,A), c'est affecter une couleur à chaque sommet de sorte que deux sommets quelconques adjacents aient toujours des couleurs différentes. Le nombre minimal de couleurs nécessaires pour colorier les sommets du graphe est appelé le **nombre chromatique** du graphe.
La coloration des sommets d'un graphe est un problème difficile en ce sens que si le nombre de sommets est grand et si le graphe comporte beaucoup d'arêtes, on ne connaît pas d'algorithme performant pour déterminer la solution minimale.
On propose ici un algorithme glouton de coloration : à chaque étape, on choisit la meilleure coloration pour le sommet considéré en espérant que globalement elle sera satisfaisante.
Soit G un graphe défini par un dictionnaire. On codera les couleurs par des entiers dans l'ordre croissant. Considérons un sommet associé à une couleur. Cherchons le prochain sommet non coloré. On le colorie avec le plus petit entier non utilisé par ses voisins.
On affiche ensuite le nombre de couleurs utilisées.

graphes.py

```
def colo(g):
    n = len(g) # nb de sommets
    couleurs_dispo = set(range(1,n+1)) # ensemble des couleurs disponibles
    g_color = {} # le dictionnaire des sommets colorés avec leurs couleurs
    for u in g: # pour chaque sommet du graphe
        couleurs_interdites  = set([g_color[v] for v in g[u] if v in g_color])
        # on interdit les couleurs des adjacents
        g_color[u] = min(couleurs_dispo - couleurs_interdites)
        # on choisit celle qui porte le plus petit numéro
    return g_color,max(g_color.values())
```

L'ordre dans lequel les sommets sont choisis a-t-il une influence ? Essayer avec un autre ordre en créant une permutation aléatoire des sommets.
On va utiliser `max(g.values())` qui donne la valeur maximale des valeurs associées aux labels du dictionnaire g.

On importera la commande `shuffle` du module `random` qui permet de mélanger les éléments d'une liste :

graphes.py

```
from numpy.random import shuffle, random

def colo_rand(g):
    n = len(g) # nb de sommets
    dic_col = {} # dictionnaire des meilleures colorations indexé par le nb
        chromatique provisoire
    liste_sommets = list(g)
    for k in range(1000):
        shuffle(liste_sommets,random=random.random) # les sommets mélangés en
            place
        couleurs_dispo = set(range(1,n+1)) # ens des couleurs disponibles
        g_color = {} # le dic des sommets colorés avec leurs couleurs
        for u in liste_sommets: # pour chaque sommet du graphe ordre aléatoire
            couleurs_interdites = set([g_color[v] for v in g[u] if v in g_color])
            g_color[u] = min(couleurs_dispo - couleurs_interdites)
        m = max(g_color.values())
        dic_col[m] = g_color # on associe le dictionnaire de coloration à
            chi_provisoire
    mi = min(dic_col.keys()) # le plus petit nb chromatique provisoire
    return 'chi(g) <= '+str(mi),dic_col[mi]  # on affiche une plus petite
        coloration possible
```

On obtient par exemple une réponse du style :

```
>>> ('chi(g) <= 4', {'bd': 4, 'be': 1, 'ac': 2, 'ae': 2, 'ba': 1, 'ed': 1, 'ca':
    2, 'dc': 3, 'ec': 1, 'da': 3, 'cd': 1})
```

4

Méthodes numériques

Sommaire

Ce chapitre est consacré à la présentation de quelques méthodes usuelles d'analyse numérique. On s'attache surtout à programmer ces méthodes en **Python**. L'exposé des fondements théoriques des schémas numériques ne rentrant pas dans le cadre de cet ouvrage, on renvoie le lecteur intéressé à l'ouvrage de référence [Dem96]. Pour un exposé systématique des méthodes numériques les plus classiques, on pourra également consulter [Gri09].

1 Les nombres en notation scientifique

Comme nous y avons déjà fait allusion notamment au chapitre 1, les nombres décimaux sont représentés en **Python** par une *mantisse* (c'est-à-dire la partie fractionnaire) et un *exposant* (la puissance de 10 par laquelle il faut multiplier la mantisse).
Considérons par exemple la représentation de 2^{100} en entier et en notation scientifique :

```
>>> 2**100, 2.**100
(1267650600228229401496703205376, 1.2676506002282294e+30)
```

La mantisse de `2.**100` est alors `1.2676506002282294` et son exposant `+30`.
Pour éviter de manipuler des représentations différentes d'un même nombre, on convient de placer la virgule décimale [1] après le premier chiffre non nul, d'où le nom de *nombre à virgule flottante* (plus couramment appelé *nombre en notation scientifique*) :

```
>>> 200e100, 0.2e100
(2e+102, 2e+99)
```

Les nombres réels sont représentés en **Python** avec le type double précision de la norme IEEE 754. Comme nous l'avons déjà mentionné à la page 31, les valeurs possibles pour l'exposant sont limités ; approximativement, en deçà de 10^{-323} le nombre est arrondi à 0 ; au delà de 10^{308}, il y a dépassement de capacité : le nombre est considéré comme infini (de valeur `inf`).

```
>>> 1e-324, 1e-323
(0.0, 1e-323)
>>> 1e308, 1e309
(1e+308, inf)
```

Il existe également des limitations pour la mantisse ; les flottants sont représentés avec 15 chiffres significatifs, comme on peut le vérifier à l'aide d'une boucle :

```
>>> i = 1
>>> while 1. + 10**(-i) > 1.:
...     i += 1
...
>>> print("1+10^(-{:d})==1 ---> True".format(i))
1+10^(-16)==1 ---> True
```

Plus précisément, le module `sys` permet d'accéder à la constante `float_info.epsilon` qui donne la différence entre 1.0 et le flottant suivant le plus proche :

```
>>> from sys import *
>>> float_info.epsilon
2.220446049250313e-16
```

On notera ce nombre ε dans la suite du paragraphe.
Notons que cette constante peut-être utilisée pour écrire une fonction qui teste l'« égalité » entre deux flottants dans la limite de la précision de l'interpréteur :

```
def equal_float(a, b):
    return abs(a - b) <= sys.float_info.epsilon
```

1. Noter que la virgule décimale (comme on dit en français) est en fait un point dans les pays anglo-saxons (et en informatique).

Un nombre réel x sera donc stocké en mémoire sous forme arrondie $\tilde{x}$ vérifiant

$$\tilde{x} = x(1+\varepsilon_x) \quad \text{avec} \quad |\varepsilon_x| \lesssim \varepsilon$$

Le nombre ε_x est alors appelé l'*erreur relative* et le nombre $\delta_x = x\varepsilon_x$ l'*erreur absolue.*
Remarquons qu'avec ces notations, le nombre ε est défini par

$$\varepsilon = \inf\{x > 0 \,|\, \widetilde{1+x} - 1 > 0\}$$

Soit à calculer la somme de deux nombres x et y. La valeur stockée sera $\tilde{z}$ avec $z = \tilde{x} + \tilde{y}$.
Dans ce cas, l'erreur absolue est

$$\delta_z = \delta_x + \delta_y + z\varepsilon = x\varepsilon_x + y\varepsilon_y + z(x+y)$$

et l'erreur relative est majorée par

$$|\varepsilon_z| \leq \frac{|x|\,|\varepsilon_x| + |y|\,|\varepsilon_y|}{|x+y|} + \varepsilon \leq \left(\frac{|x|+|y|}{|x+y|} + 1\right)\varepsilon$$

On constate notamment que si l'on calcule la différence de deux nombres très proches l'un de l'autre (si l'on calcule $x-y$, avec $|x-y|$ très petit devant $|x|$ (et $|y|$)) la perte de précision risque d'être très importante.
En outre, en calculant l'erreur relative de la somme de trois nombres, on peut constater que l'addition des réels n'est plus commutative.
Pour un exemple de calcul avec perte vertigineuse de précision, on reviendra à l'exemple donné dans la section 2.5 du chapitre 2.
Pour un examen précis des questions liées aux calculs numériques approchés, on renvoie au premier chapitre de [Dem96]. Pour une étude encore plus approfondie, on consultera [LZ04] [2].
En conclusion, les problèmes d'arrondis sont à prendre en compte dès que l'on travaille avec des nombres à virgule flottante. En général, pour éviter les pertes de précision, on essayera autant que peut se faire de respecter les règles suivantes :

- éviter d'additionner ou de soustraire deux nombres d'ordres de grandeur très différents ;
- éviter de soustraire deux nombres presque égaux.

Sinon, on travaillera en précision étendue.
De plus, pour calculer une somme de termes ayant des ordres de grandeur très différents (par exemple dans le calcul des sommes partielles d'une série), on appliquera le principe dit de « la photo de classe » : les petits devant, les grands derrière.
Par exemple, si on souhaite donner une approximation décimale de $\zeta(4) = \sum_{n=1}^{+\infty} \frac{1}{n^4}$ dont la valeur exacte est $\frac{\pi^4}{90}$, on constate qu'on obtient une meilleure précision en commençant par sommer les termes les plus petits :

2. Article disponible à l'adresse http://hal.inria.fr/inria-00071477/PDF/RR-5105.pdf

```
from math import pi

def zeta(n, p):
    return sum(1/i**p for i in range(1, n+1))

def zeta_reverse(n, p):
    return sum(1/i**p for i in range(n, 0, -1))

n, p = 2**18, 4
print("{: .16f}".format(zeta(n, p) - pi**4/90))
print("{: .16f}".format(zeta_reverse(n, p) - pi**4/90))
```

```
-0.0000000000002764
 0.0000000000000002
```

2 Résolution d'équations non linéaires

Pour résoudre numériquement une équation non linéaire de la forme $f(x) = 0$, il faut au préalable étudier les variations de la fonction pour isoler les zéros. On suppose dans la suite que cette étude est déjà réalisée et que l'on dispose d'un intervalle $[a, b]$ sur lequel la fonction est strictement monotone, et tel que l'on ait $f(a)f(b) < 0$. D'après le théorème de la bijection monotone, on en déduit que f admet un unique zéro dans $]a, b[$.
Pour programmer les différentes méthodes numériques de résolution, nous allons utiliser la classe parente suivante :

equations.py

```
class NonLineaire(object):
    def __init__(self, approx, eps=1e-14, maxIter=10**7):
        self.approx = approx
        self.eps = eps
        self.maxIter = maxIter

    def resolution(self, f):
        raise NotImplementedError
```

Pour éviter les risques de bouclage, on a introduit une borne `maxIter` pour limiter ultérieurement le nombre d'itérations.

2.1 Méthode de dichotomie

Cette méthode consiste à construire un couple de suites adjacentes qui convergent vers la racine ℓ de f.
On pose $a_0 = a$ et $b_0 = b$. Puis pour tout $n \in \mathbb{N}$, on pose

$$(a_{n+1}, b_{n+1}) = \begin{cases} (a_n, c_n) & \text{si } f(a_n)f(c_n) < 0 \\ (c_n, b_n) & \text{si } f(a_n)f(c_n) \geq 0 \end{cases} \quad \text{où} \quad c_n = \frac{a_n + b_n}{2}$$

L'erreur absolue de l'approximation de ℓ par a_n ou b_n est alors majorée par $b_n - a_n = \frac{b-a}{2^n}$; et l'erreur relative est de l'ordre de $\frac{b_n - a_n}{|c_n|}$.

equations.py

```
class Dichotomie(NonLineaire):
    def subdivise(self, a, b, f):
        return (a+b) * 0.5

    def resolution(self, f):
        [a, b] = self.approx
        a, b = min(a, b), max(a, b)
        eps, maxIter = self.eps, self.maxIter
        for i in range(maxIter):
            c = self.subdivise(a, b, f)
            if b-a < eps:
                return c
            if f(a) * f(c) < 0:
                b = c
            else:
                a = c
        return c
```

2.2 Méthode « regula falsi »

Cette méthode est une variante de la précédente : au lieu de choisir, pour découper l'intervalle $[a_n, b_n]$, le milieu $c_n = \frac{a_n+b_n}{2}$, on considère l'intersection de la corde joignant les points $A_n(a_n, f(a_n))$ et $B_n(b_n, f(b_n))$ avec l'axe des abscisses. On pose alors

$$c_n = a_n - f(a_n) \cdot \frac{b_n - a_n}{f(b_n) - f(a_n)}$$

La convergence est souvent plus rapide qu'avec la méthode de la dichotomie.
Pour écrire une classe `RegulaFalsi`, il suffit de la faire hériter des méthodes de la classe parente `Dichotomie` et en surchargeant le méthode `subdivise` (c'est un exemple de polymorphisme) :

equations.py

```
class RegulaFalsi(Dichotomie):
    def subdivise(self, a, b, f):
        return a - f(a) * (b-a) / (f(b) - f(a))
```

2.3 Méthode du point fixe

On se propose de résoudre une équation de la forme

$$f(x) = x$$

Les solutions de cette équation sont appelés des *points fixes* de f ; géométriquement, ce sont les abscisses des points d'intersection entre la courbe représentative de f et la première bissectrice des axes.

Une méthode fructueuse pour résoudre ce type d'équation est de procéder par itération, à condition toutefois que soient vérifiées par exemple les hypothèses du *théorème du point fixe*, dont voici l'énoncé :

Soit I un intervalle fermé stable pour une application contractante f de rapport de LIPSCHITZ $k \in [0,1[$; alors

- f possède un unique point fixe $\ell \in I$;
- pour tout $\alpha \in I$, la suite $\begin{cases} x_0 = \alpha \\ \forall n \in \mathbb{N} \quad x_{n+1} = f(x_n) \end{cases}$ converge vers ℓ et on a la majoration

$$\forall n \in \mathbb{N}, \quad |x_n - \ell| \leq k^n |x_0 - \ell|$$

Les points fixes peuvent être classés en différentes catégories suivant la valeur de $f'(\ell)$ en points fixes attractifs ($|f'(\ell)| < 1$), répulsifs ($|f'(\ell)| > 1$), super-attractifs ($f'(\ell) = 0$), indifférents ($|f'(\ell)| = 1$).

equations.py

```
class PointFixe(NonLineaire):
    def resolution(self, f):
        eps, maxIter = self.eps, self.maxIter
        x0, x1 = self.approx, f(self.approx)
        for i in range(maxIter):
            if abs(x0 - x1) < eps:
                return x1
            x0, x1 = x1, f(x1)
        return x1
```

2.4 Méthode de Newton

La méthode de NEWTON est une des méthodes les plus utilisées pour la résolution des équations non linéaires ; en effet, d'une part, elle converge généralement très vite, d'autre part, elle se généralise aux systèmes non linéaires.
Le principe en est le suivant : partant d'une valeur x_0, on trace la tangente à la courbe représentative de f au point $M_0(x_0, f(x_0))$; cette droite coupe l'axe des abscisses en un point d'abscisse

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

On réitère alors l'opération tant que deux valeurs consécutives sont distantes d'au moins la valeur de seuil `eps`. Ceci revient à appliquer la méthode du point fixe avec la fonction

$$\varphi(x) = x - \frac{f(x)}{f'(x)}$$

Pour programmer cet algorithme, on utilise alors la classe `PointFixe`, dont on surcharge la méthode `resolution` :

equations.py

```
class Newton(PointFixe):
    def resolution(self, f, fp):
        return PointFixe.resolution(self, lambda x:x-f(x)/fp(x))
```

2.5 Méthode de la sécante

Dans le cas où l'évaluation numérique de la dérivée de f s'avère complexe, on peut modifier la méthode de NEWTON en remplaçant le calcul de la dérivée par un taux d'accroissements ; on obtient alors la méthode de la tangente, qui est une méthode à deux points :

$$x_{n+1} = x_n - f(x_n) \cdot \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$

Dans ce cas, la convergence est moins rapide que dans le cas de la méthode de Newton, mais le résultat peut être obtenu plus rapidement si le calcul de f' est très long.

equations.py

```
class Secante(NonLineaire):
    def resolution(self, f):
        eps, maxIter = self.eps, self.maxIter
        [x0, x1] = self.approx
        for i in range(maxIter):
            x2 = x1 - f(x1) * (x1-x0) / (f(x1) - f(x0))
            if abs(x0 - x2) < eps:
                    return x2
            x0, x1 = x1, x2
        return x2
```

2.6 Comparatif des différentes méthodes

Comparons les différentes méthodes de résolution sur la recherche d'une approximation décimale de $\sqrt{2}$. D'une part, on peut appliquer les méthodes précédentes (hormis celle du point fixe) à l'équation $x^2 - 2 = 0$.

D'autre part, on peut appliquer la méthode du point fixe en réécrivant cette équation sous la forme $x = f(x)$ où $f(x) = \frac{1}{2}\left(x + \frac{a}{x}\right)$. Il est aisé de prouver que l'intervalle $]0, +\infty[$ est stable par f et que $f(]0, +\infty[) \subset [\sqrt{a}, +\infty[$; donc quelle que soit la valeur de $x_0 > 0$, on a $\forall n \geq 1,\ x_n \geq \sqrt{a}$; de plus la fonction f est $\frac{1}{2}$-lipschitzienne sur $[\sqrt{a}, +\infty[$ et possède (sur cet intervalle) pour unique point fixe le réel $\sqrt{a}$, qui est un point super-attractif.

Donc la suite définie par $\begin{cases} x_0 = a \\ \forall n \in \mathbb{N} \quad x_{n+1} = f(x_n) \end{cases}$ converge vers $\sqrt{a}$.

Comme cela a déjà été mentionné dans la section 2.3.1 du chapitre 3, l'algorithme était déjà connu des Grecs, notamment de HÉRON D'ALEXANDRIE[3].

equations.py

```
from math import *
print("{:>7}| {:^9}|{:^9}| {:^9} | {:^9} | {:^9} "
      .format("maxIter", "Dichotomie", "RegulaFalsi",
      "Sécante", "PointFixe", "Newton"))
for i in range(1, 8):
   di = Dichotomie([1, 2], maxIter=i)
```

3. Si l'on prend $a = 2$ et si l'on choisit $x_0 = 3/2$, alors on trouve $x_2 = 17/12$ et $x_3 = \frac{1}{2}\left(\frac{17}{12} + \frac{24}{17}\right) = 1 + \frac{24}{60} + \frac{51}{60^2} + \frac{10}{60^3} + \cdots$ en numération sexagésimale. On a trouvé un jour cette valeur sur une tablette babylonienne du XVIII^e siècle avant JÉSUS-CHRIST ! Par ailleurs un texte indien pouvant dater du VI^e siècle avant l'ère chrétienne fournit sans explication pour $\sqrt{2}$ la valeur $1 + \frac{1}{3} + \frac{1}{3\times4} - \frac{1}{3\times4\times34}$, autrement dit x_3 !

```
    rg = RegulaFalsi([1, 2], maxIter=i)
    pf = PointFixe(2, maxIter=i)
    nw = Newton(2, maxIter=i)
    se = Secante([1, 2], maxIter=i)
    comp = [eval(m).resolution(lambda x:x**2-2) for m in ['di', 'rg', 'se']]
    comp.append(pf.resolution(lambda x:0.5*(x+2/x)))
    comp.append(nw.resolution(lambda x:x**2-2, lambda x:2*x))
    comp = [sqrt(2) - c for c in comp]
    print("{:>6d} | {: .2e} | {: .2e} | {: .2e} | {: .2e} | {: .2e}"
          .format(i, comp[0], comp[1], comp[2], comp[3], comp[4]))
```

```
maxIter| Dichotomie|RegulaFalsi|  Sécante  | PointFixe |  Newton
     1 | -8.58e-02 |  8.09e-02 |  8.09e-02 | -2.45e-03 | -2.45e-03
     2 |  1.64e-01 |  1.42e-02 |  1.42e-02 | -2.12e-06 | -2.12e-06
     3 |  3.92e-02 |  2.45e-03 | -4.21e-04 | -1.59e-12 | -1.59e-12
     4 | -2.33e-02 |  4.20e-04 |  2.12e-06 |  2.22e-16 |  0.00e+00
     5 |  7.96e-03 |  7.21e-05 |  3.16e-10 |  2.22e-16 |  2.22e-16
     6 | -7.66e-03 |  1.24e-05 | -2.22e-16 |  2.22e-16 |  2.22e-16
     7 |  1.51e-04 |  2.12e-06 |  0.00e+00 |  2.22e-16 |  2.22e-16
```

La méthode du point fixe converge ici à une vitesse vertigineuse du fait que le point fixe est super-attractif. On constate de plus que l'algorithme de HÉRON semble coïncider avec la méthode de NEWTON appliquée à la fonction $x \mapsto x^2 - 2$, ce que l'on vérifie par un calcul rapide. À noter que même si la méthode de NEWTON permet en général d'obtenir une convergence quadratique, un mauvais choix de la valeur initiale peut provoquer la divergence de cette méthode (notamment si la courbe représentative de f présente au point d'abscisse x_0 une tangente à peu près horizontale). D'où l'importance d'une étude préalable soignée de la fonction f.

3 Résolution numérique d'équations différentielles

3.1 Position du problème

Dans cette section, on s'intéresse aux équations différentielles d'ordre 1 dont la fonction inconnue est soit à valeurs réelles, soit à valeurs vectorielles.
Soit U un ouvert de $\mathbb{R} \times \mathbb{R}^n$ et $f\colon U \longrightarrow \mathbb{R}^n$ une application continue sur l'ouvert U.
Alors pour $(t_0, x_0) \in U$, on appelle *solution du problème de* CAUCHY :

$$\begin{cases} x'(t) = f(t, x(t)) \\ x(t_0) = x_0 \end{cases} \qquad (\mathcal{C})$$

tout couple (I, x) où I est un intervalle contenant t_0 et $x\colon I \longrightarrow \mathbb{R}^n$ une solution sur I de l'équation différentielle $x'(t) = f(t, x(t))$ telle que $x(t_0) = x_0$.
L'équation différentielle qui apparaît dans $(\mathcal{C})$ sera appelée équation différentielle scalaire si $n = 1$, et système différentiel sinon.
Si l'application $f\colon U \longrightarrow \mathbb{R}^n$ est de *classe* $\mathcal{C}^1$ sur l'ouvert U, alors, d'après le théorème de CAUCHY-LIPSCHITZ, pour $(t_0, x_0) \in U$, il existe une unique solution maximale au problème de CAUCHY $(\mathcal{C})$.

Dans la plupart des cas, on ne sait pas résoudre explicitement le problème de Cauchy $(\mathcal{C})$; d'où la nécessité de mettre au point des méthodes numériques de résolution approchée d'un tel problème.
Nous allons créer une classe `ODE` qui servira d'interface pour la programmation des différents schémas numériques. Cette classe possède comme attribut la fonction f, le pas h, les conditions initiales (t_0, x_0) ; de plus, elle programme la résolution de l'équation de départ sur un intervalle $\mathrm{I} = [a, b]$; pour ce faire, il faut parcourir l'intervalle I à partir de t_0 vers la droite jusqu'à atteindre b, puis à partir de t_0 vers la gauche jusqu'à atteindre a.

ode.py

```
class ODE(object):
    def __init__(self, f, h):
        self.f = f
        self.h = h

    def iteration(self):
        raise NotImplementedError

    def CI(self, t0, x0):
        self.liste = [[t0, x0]]
        self.k = 0

    def resolution(self, a, b):
        self.indice = -1
        while self.liste[-1][0] <= b:
            self.liste.append(self.iteration())
        self.h = - self.h
        self.indice = 0
        while self.liste[0][0] >= a:
            self.liste.insert(0, self.iteration())
        return self.liste
```

3.2 Méthode d'Euler

La méthode d'Euler est la méthode la plus simple pour résoudre numériquement une équation différentielle. Elle présente un réel intérêt théorique, puisque elle peut être utilisée pour démontrer le théorème de Cauchy-Lipschitz. Toutefois, son intérêt pratique est limité par sa faible précision.
L'idée d'Euler consiste à utiliser l'approximation :

$$x'(t) \approx \frac{x(t+h) - x(t)}{h} \quad \text{pour } h \text{ petit}$$

Autrement dit,

$$x(t+h) \approx x(t) + h \cdot x'(t) = x(t) + h \cdot f(t, x(t))$$

Partant du point (t_0, x_0), on suit alors la droite de pente $f(t_0, x_0)$ sur l'intervalle de temps $[t_0, t_0 + h]$. On pose alors :

$$\begin{cases} t_1 = t_0 + h \\ x_1 = x_0 + h \cdot f(t_0, x_0) \end{cases}$$

De nouveau, partant du point (t_1, x_1), on suit alors la droite de pente $f(t_1, x_1)$ sur l'intervalle de temps $[t_1, t_1 + h]$, et ainsi de suite.

On construit ainsi une suite de points de la manière suivante :

$$\forall k \in [\![0, \mathrm{N}]\!], \quad \begin{cases} t_{k+1} = t_k + h \\ x_{k+1} = x_k + h \cdot f(t_k, x_k) \end{cases}$$

La ligne brisée joignant les points $\{(t_k, x_k) \,|\, k \in [\![0, \mathrm{N}]\!]\}$ donnera une solution approchée de notre équation différentielle.

Voici une manière de programmer ce schéma numérique :

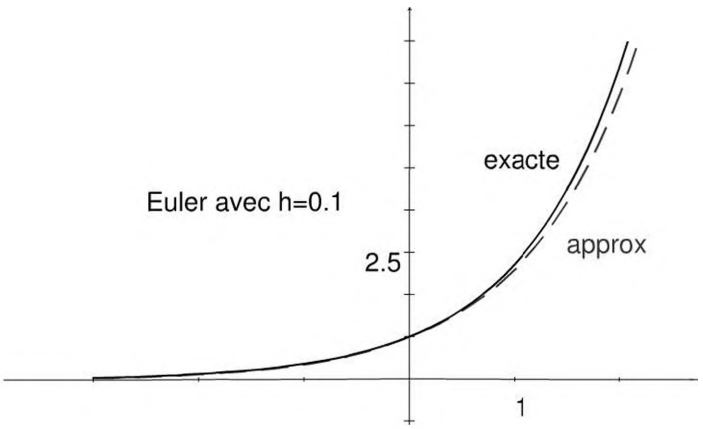
ode.py

```
class Euler(ODE):
    def iteration(self):
        f, h = self.f, self.h
        [t, x] = self.liste[self.indice]
        return [t + h, x + h * f(t, x)]
```

Testons l'efficacité de cette méthode pour le problème de CAUCHY :

$$\begin{cases} x'(t) = x(t) \\ x(0) = 1 \end{cases} \qquad (\mathcal{C})$$

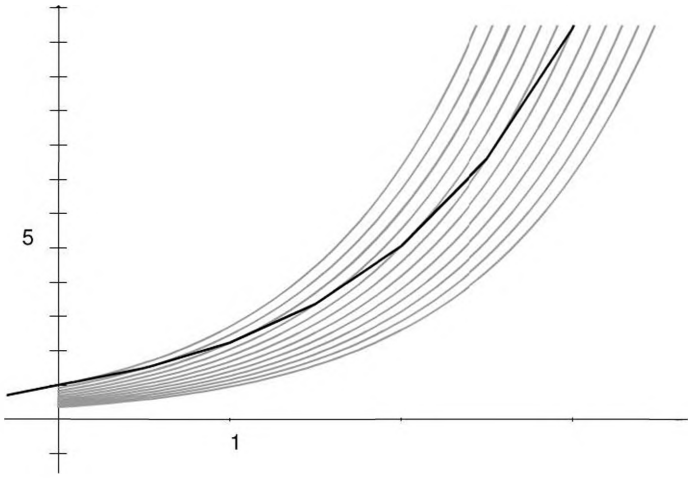
dont la solution exacte est la fonction exponentielle.



ode.py

```
from math import exp
import PostScript as ps

def f(t, x): return x
a, b, h = -3, 3, 0.1
approx = Euler(f, h); approx.CI(0, 1.0)
liste_approx = approx.resolution(a, b)
liste_exacte = [[t, exp(t)] for t in ps.srange(a, b, h)]

nomFichier, boite = "ode_exponentielle1", [-4.5, -2, 3.5, 14]
zoom, rouge, bleu, noir = 40, (1, 0, 0), (0, 0, 1), (0, 0, 0)

ps.plot(nomFichier, boite, zoom, [[liste_approx, bleu], [liste_exacte, rouge]],
     ['approx', [1.5, 3], bleu], ['exacte', [0.7, 5], rouge],
     ['Euler avec h=0.1', [-3, 9], noir], ratioY=.4, marge=1.1)
```

Il faut bien comprendre qu'à chaque étape, on repart dans la direction d'une solution exacte de l'équation différentielle, mais qui n'est pas celle qui est solution du problème de CAUCHY initial. Sur le graphique ci-contre, on trace une solution approchée pour la condition initiale $x(0) = 1$ et les courbes intégrales de notre équation qui passent par les points (t_k, x_k).



ode.py

```
from math import exp
import PostScript as ps

def f(t, x): return x
a, b, h = 0, 5, 0.5
approx = Euler(f, h); approx.CI(0, 1.0)
liste_approx = approx.resolution(a, b)
courbes = []
for x in liste_approx:
    [x0, y0] = x
    courbes.append([[[t, exp(t-x0) * y0] for t in
          ps.srange(a, b, 0.05)], (1 - x0/10, 0.1, x0/30), 0.4])
courbes.append([liste_approx, (0, 0, 1), 1])
ps.plot('ode_exponentielle2', [-0.5, -2.5, 3.5, 11.5], 80, courbes, ratioY=.2)
```

Pour juger de la qualité d'une méthode (ou schéma) numérique de résolution d'équations différentielles, il faut prendre en compte plusieurs critères. Pour un exposé rigoureux de ces notions, on se reportera à l'ouvrage de référence [Dem96].

- L'*erreur de consistance* donne l'ordre de grandeur de l'erreur effectué à chaque pas. Par exemple, dans le cas de la méthode d'EULER, l'erreur de consistance mesure l'erreur qu'entraîne le fait d'approcher le nombre dérivé par un taux d'accroissement. La sommation des erreurs de consistance à chaque pas donnera l'ordre de grandeur de l'erreur globale (sous des hypothèses de régularité pour la fonction f). À l'aide de la formule de TAYLOR-LAGRANGE, on peut montrer que dans le cas de la méthode d'EULER, l'erreur de consistance est dominée par h^2 ; on dit alors que cette méthode est d'ordre 1. Ainsi, d'un point de vue théorique, plus le pas est petit, meilleure sera l'approximation.
  Un schéma numérique est dit *consistant* si la somme des erreurs de consistance tend vers 0 quand le pas h tend vers 0 ; ce qui est le cas de la méthode d'EULER (en gros, car $Nh^2 \sim h \to 0$ où N désigne le nombre d'itérations).
- La *stabilité* contrôle la différence entre deux solutions approchées correspondant à deux conditions initiales voisines : un schéma est dit *stable* si un petit écart entre les conditions initiales $x(t_0) = x_0$ et $\tilde{x}(t_0) = x_0 + \varepsilon$ et de petites erreurs d'arrondi dans le calcul récurrent des $\tilde{x}_k$ provoquent une erreur finale $|x_k - \tilde{x}_k|$ contrôlable. La méthode d'EULER

est une méthode stable.

- Un schéma est dit *convergent* lorsque l'erreur globale (qui est le maximum des écarts entre la solution exacte et la solution approchée) tend vers 0 lorsque le pas tend vers 0. En fait, pour que le schéma soit convergent, il suffit que la méthode soit consistante *et* stable.
- Enfin pour la mise en application du schéma, il faut aussi prendre en compte l'influence des erreurs d'arrondi. En effet, afin de minimiser l'erreur globale théorique, on pourrait être tenté d'appliquer la méthode d'EULER avec un pas très petit, par exemple de l'ordre de 10^{-16}, mais ce faisant, outre que le temps de calcul deviendrait irréaliste, les erreurs d'arrondi feraient diverger la solution approchée très rapidement, puisque pour des flottants **Python** de l'ordre de 10^{-16}, les calculs ne sont plus du tout exacts !
  En pratique, il faut prendre h assez petit (pour que la méthode converge assez rapidement du point de vue théorique), mais pas trop petit non plus (pour que les erreurs d'arrondis ne donnent pas lieu à des résultats incohérents, et que les calculs puissent être effectués en un temps fini). La question de trouver de manière théorique le pas optimal peut s'avérer un problème épineux. Dans un premier temps, on peut se contenter de faire des tests pratiques comme dans l'exemple suivant.

Dans le script suivant, on applique la méthode d'EULER sur l'intervalle $[0,1]$ avec la condition initiale $x(0) = 1$. Pour minimiser le temps de calcul, l'algorithme est réécrit sous forme abrégée.

ode.py

```
from math import exp
import PostScript as ps

print("{:^10} | {:^12} | {:^12} | {:^13}".format('h','x(t)','exp(t)','erreur'))
print('-'*57)
for i in range(1, 8):
    h, t, x  = 10**(-i), 0, 1
    while t < 1:
        t, x = t + h, x * (h + 1)
    print("{0:>10g} | {1:>.10f} | {2:>.10f} | {3:> .10f}"
            .format(h, x, exp(t),  exp(t) - x))
```

```
    h      |     x(t)     |    exp(t)    |    erreur
---------------------------------------------------------
       0.1 | 2.8531167061 | 3.0041660239 |  0.1510493178
      0.01 | 2.7048138294 | 2.7182818285 |  0.0134679990
     0.001 | 2.7169239322 | 2.7182818285 |  0.0013578962
    0.0001 | 2.7184177414 | 2.7185536702 |  0.0001359288
     1e-05 | 2.7182954199 | 2.7183090114 |  0.0000135915
     1e-06 | 2.7182804691 | 2.7182818285 |  0.0000013594
     1e-07 | 2.7182819660 | 2.7182820996 |  0.0000001336
     1e-08 | 2.7182817983 | 2.7182818347 |  0.0000000363
     1e-09 | 2.7182820738 | 2.7182818299 | -0.0000002438
```

Pour un pas $h \lesssim 10^{-9}$, le temps de calcul devient trop important, de plus, on voit poindre l'effet des erreurs d'arrondis, puisque la solution approchée est devenu supérieure à l'expo-

nentielle, ce qui est mathématiquement faux, en vertu de l'inégalité :

$$\forall n \in \mathbb{N}, \quad \left(1+\frac{1}{n}\right)^n \leq e$$

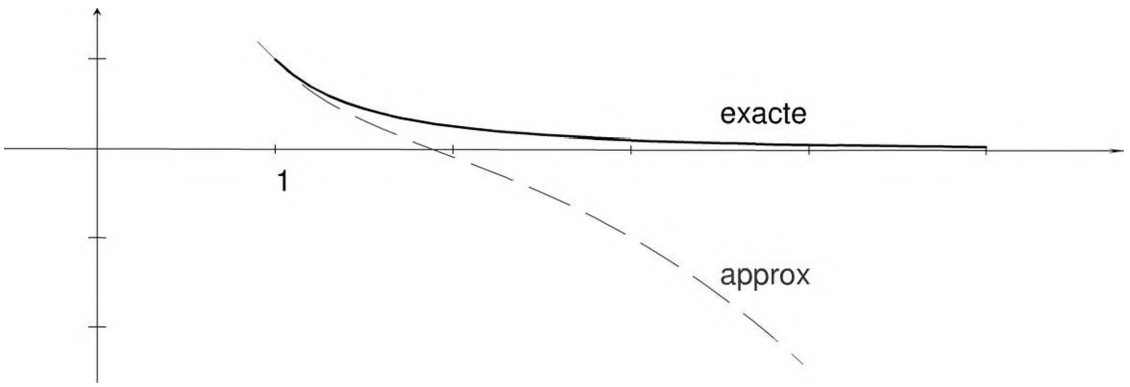
En général, il ne suffit pas qu'un schéma numérique soit convergent pour qu'il donne de bons résultats sur n'importe quelle équation différentielle. Encore faut-il que le problème soit mathématiquement bien posé (en particulier, les hypothèses du théorème de CAUCHY-LIPSCHITZ doivent être vérifiées), qu'il soit numériquement bien posé (continuité suffisamment bonne par rapport aux conditions initiales) et qu'il soit bien conditionné (temps de calcul raisonnable).
Considérons par exemple le problème de CAUCHY :

$$\begin{cases} x(1) = 1 \\ x'(t) = 3\frac{x(t)}{t} - \frac{5}{t^3} \end{cases}$$

dont la solution est la fonction $\varphi : t \mapsto 1/t^2$. La résolution numérique sur $[1,5]$ s'effectue de la manière suivante :

ode.py

```
from math import exp
import PostScript as ps

def f(t, x): return 3 * x / t - 5 / t**3

a, b, h = 1, 5, 0.1
approx = Euler(f, h); approx.CI(1, 1.0)
liste_approx = approx.resolution(a, b)
liste_exacte = [[t, 1 / t**2] for t in ps.srange(a, b, h)]
rouge, bleu, boite = (1, 0, 0), (0, 0, 1), [-.5, -5.5, 5.5, 2.7]

ps.plot("mal_pose", boite, 60, [[liste_approx, bleu], [liste_exacte, rouge]],
  ['approx', [3.5,-3], bleu], ['exacte', [3.5,.3], rouge], ratioY=.5, marge=1.1)
```



On constate qu'ici la solution approchée s'écarte assez rapidement de la solution exacte φ. En effet, la forme générale des solutions de l'équation différentielle précédente est donnée par $t \mapsto \lambda x^3 + 1/x^2$. La condition $x(1) = 1$ impose $\lambda = 0$, mais dès qu'on s'écarte de la solution φ on suit « tangentiellement » des courbes intégrales qui comportent un terme en λx^3 et donc qui

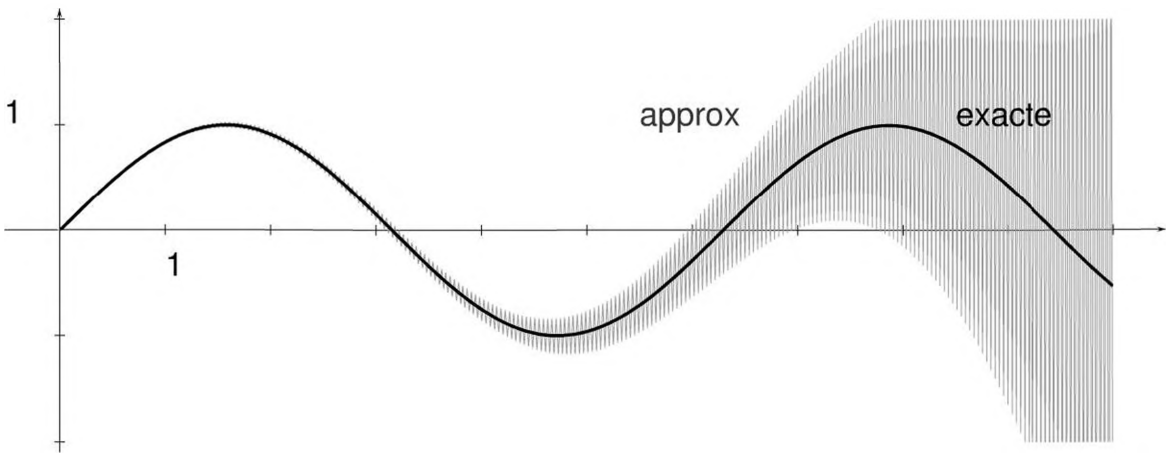
diffère notablement de la solution. Donc le problème est *mal posé*. Ici le schéma numérique n'est pas en cause.
Soit à résoudre à présent le problème de CAUCHY :

$$\begin{cases} x(1) = 1 \\ x'(t) = 100\,(\sin(t) - x) \end{cases}$$

dont la solution est la fonction $\varphi\colon t \mapsto \frac{1}{10001}(-100\cos t + 10000\sin t + 100\exp(-100\cdot t)) \approx \sin t$.
Que se passe-t-il si on résout ce problème avec un pas de l'ordre de 0.02 ?

ode.py

```
from math import *
import PostScript as ps

def f(t, x): return 100 * (sin(t) - x)

def g(t): return (-100*cos(t) + 10000*sin(t) + 100*exp(-100*t))/10001

a, b, h = 0, 10, 0.02012
approx = Euler(f, h); approx.CI(0.0, 0.0)
liste_approx = approx.resolution(a, b)
liste_exacte = [[t, g(t)] for t in ps.srange(a, b, h)]
rouge, bleu, boite = (1, 0, 0), (0, 0, 1), [-.5, -2, 10, 2]

ps.plot("mal_conditionne", boite, 80, [[liste_approx, bleu],
        [liste_exacte, rouge, 1]], ['approx', [1.5, 3], bleu],
        ['exacte', [0.7, 5], rouge], marge=1.1)
```



Avec un tel pas, la solution approchée oscille en s'éloignant de plus en plus de la solution exacte. En effet, le schéma numérique est donné par :

$$x_{k+1} = x_k + 100\,h\,(\sin t_k - x_k) = (1 - 100h)x_k + 100h\sin t_k$$

Donc une erreur de ε_k sur x_k aura les répercussions suivantes sur le calcul des termes ultérieurs :

$$\varepsilon_{k+1} = (1 - 100h)\varepsilon_k \implies \varepsilon_{k+n} = (1 - 100h)^n\varepsilon_k$$

Ainsi donc, tant que $1-100h \geq -1$, c'est-à-dire tant que $h \lesssim 0.002$, une petite erreur sur l'un des termes aura des répercussions allant en s'amplifiant.
Bien que le problème soit numériquement bien posé, il est en fait *mal conditionné.*

3.3 Méthode de Runge-Kutta d'ordre 4

Si, comme nous l'avons déjà dit, la méthode d'EULER présente un intérêt théorique, on préfère en pratique des méthodes d'ordre plus élevé. Parmi la multitude des schémas numériques (méthodes à un pas comme celle de TAYLOR, méthodes à pas multiples comme celles d'ADAMS-BASHFORTH, d'ADAMS-MOULTON, de prédiction-correction) celle qui présente le meilleur rapport précision/complexité est certainement celle de RUNGE-KUTTA d'ordre 4.
En voici le schéma pour $\forall k \in [\![0, N]\!]$,

$$\begin{cases} t_{k+1} &= t_k + h \\ x_{k+1} &= x_k + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{cases} \quad \text{où} \quad \begin{cases} k_1 &= h \cdot f(t_n, x_n) \\ k_2 &= h \cdot f\left(t_n + \frac{h}{2}, x_n + \frac{k_1}{2}\right) \\ k_3 &= h \cdot f\left(t_n + \frac{h}{2}, x_n + \frac{k_2}{2}\right) \\ k_4 &= h \cdot f(t_n + h, x_n + k_3) \end{cases}$$

Les coefficients qui apparaissent dans ces formules mystérieuses sont judicieusement ajustés pour obtenir une méthode d'ordre 4, sans pour autant avoir à calculer les dérivées successives de f (comme c'est le cas dans les méthodes de TAYLOR), ou à recourir à une formule de récurrence d'ordre au moins 2 pour définir x_k (comme c'est le cas dans les méthodes à pas multiples).
Sans plus attendre, complétons notre fichier `ode.py` en y ajoutant ce schéma.

ode.py

```
class RK4(ODE):
    def iteration(self):
        f, h = self.f, self.h
        [t, x] = self.liste[self.indice]
        k1 =  h * f(t, x)
        k2 =  h * f(t + h/2, x + k1/2)
        k3 =  h * f(t + h/2, x + k2/2)
        k4 =  h * f(t + h, x + k3)
        return [t + h, x + (k1 + 2*k2 + 2*k3 + k4) / 6]
```

Effectuons une comparaison des méthodes d'EULER et de RUNGE-KUTTA en les appliquant au problème de CAUCHY :

$$\begin{cases} x'(t) = x(t) \\ x(0) = 1 \end{cases} \qquad (\mathcal{C})$$

ode.py

```
from math import exp
import PostScript as ps

def f(t, x): return x

a, b, h = 0, 1, 0.1
liste_exacte = [[t, exp(t)] for t in ps.srange(a, b, h)]
```

```
print("{:^10} | {:^17} | {:^17}".format('h', 'erreur euler', 'erreur rk4'))
print('-'*52)

for i in range(1, 7):
    h = 10**(-i)
    euler = Euler(f, h); euler.CI(0, 1.0)
    e = euler.resolution(a, b)[-1]
    rk4 = RK4(f, h); rk4.CI(0, 1.0)
    r = rk4.resolution(a, b)[-1]
    print("{0:>10g} | {1:> .10e} | {2:> .10e} "
          .format(h, exp(e[0]) - e[1], exp(r[0]) - r[1]))
```

```
    h      |   erreur euler    |    erreur rk4
----------------------------------------------------
       0.1 |  1.5104931784e-01 |  2.5338874514e-06
      0.01 |  1.3467999038e-02 |  2.2464341498e-10
     0.001 |  1.3578962232e-03 |  2.2204460493e-14
    0.0001 |  1.3592881559e-04 | -2.6645352591e-13
     1e-05 |  1.3591551171e-05 | -5.2180482157e-12
     1e-06 |  1.3591611072e-06 |  2.1464163780e-11
```

On constate que pour $h = 0.001$, la méthode d'EULER ne donne que trois décimales significatives du nombre e, alors que la méthode de RUNGE-KUTTA en donne quatorze ! On remarque également qu'il est inutile d'appliquer la méthode de RUNGE-KUTTA avec un pas $h \gtrsim 0.001$, puisque dans ce cas les erreurs d'arrondi prennent le dessus par rapport au gain théorique de précision.

3.3.1 Une équation différentielle non linéaire, du 1er ordre

Soit à représenter au voisinage de l'origine les courbes intégrales de l'équation différentielle :

$$x'(t) = 1 + t^2 \cdot x(t)^2$$

Si l'on tente de résoudre cette équation avec la classe précédente, le déroulement du programme sera arrêté à un moment par une erreur indiquant un dépassement de capacité pour les flottants. En effet, si on considère par exemple la solution maximale correspondant à la solution initiale $x(0) = 0$, on peut montrer qu'elle est définie sur un intervalle ouvert borné et symétrique par rapport à 0 ; de plus, elle admet des limites infinies aux bornes de cet intervalle, ce qui explique qu'à partir d'une certaine valeur de t, x prend des valeurs au-delà des valeurs permises pour les flottants.
Nous sommes donc amenés à modifier notre classe `ODE` en ajoutant la possibilité d'indiquer une condition d'arrêt dans la résolution numérique d'une équation différentielle. On s'inspire ici d'une classe donnée dans [Lan09].

ode.py

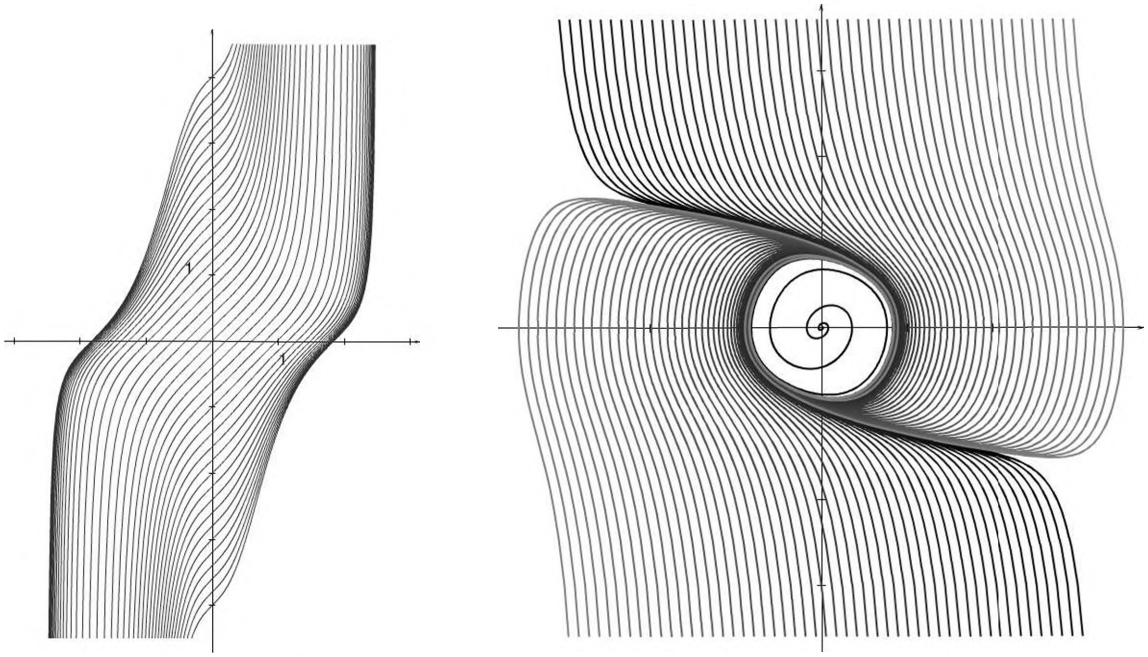
```
class ODE(object):
    def __init__(self, f, h):
        self.f = f
        self.h = h
```

```
    def iteration(self):
        raise NotImplementedError

    def CI(self, t0, x0):
        self.liste = [[t0, x0]]
        self.k = 0

    def resolution(self, a, b, termine=None):
        if termine is None:
            termine = lambda x: False
        self.indice = -1
        while (self.liste[-1][0] <= b and not termine(self.liste[-1][1])):
            self.liste.append(self.iteration())
        self.h = - self.h
        self.indice = 0
        while (self.liste[0][0] >= a and not termine(self.liste[0][1])):
            self.liste.insert(0, self.iteration())
        return self.liste

class RK4(ODE):
    def iteration(self):
        f, h = self.f, self.h
        [t, x] = self.liste[self.indice]
        k1 =  h * f(t, x)
        k2 =  h * f(t + h/2, x + k1/2)
        k3 =  h * f(t + h/2, x + k2/2)
        k4 =  h * f(t + h, x + k3)
        return [t + h, x + (k1 + 2*k2 + 2*k3 + k4) / 6]
```

Pour résoudre l'équation $x' = 1 + t^2 x^2$, on indique alors comme condition d'arrêt de la résolution numérique, le dépassement d'une certaine valeur par x : dès que $|x(t)| > 4.5$, on sort de la boucle.

ode.py

```
import PostScript as ps

def f(t, x): return 1 + t**2 * x**2
a, b, h, bleu, boite = -4, 4, 0.01, (0, 0, 1), [-4.5, -4.5, 4.5, 4.5]
def horscadre(x): return abs(x) > 4.5

courbes = []
for k in ps.srange(-4, 4, 0.2):
    sol = RK4(f, h); sol.CI(0, k)
    courbes.append([sol.resolution(a, b, horscadre), bleu])

ps.plot("exemple_rk4", boite, 40, courbes)
```



(a) Courbes intégrales de $x' = 1 + t^2x^2$

(b) L'oscillateur de van der Pol

FIGURE 4.1 – Résolutions numériques par la méthode de Runge-Kutta

3.3.2 Un système autonome

Soit à représenter les courbes intégrales du système autonome suivant, appelé oscillateur de van der Pol :

$$\begin{cases} \frac{dx}{dt} = y \\ \frac{dy}{dt} = \frac{y}{2} - x - y^3 \end{cases}$$

Il faut au préalable adapter la classe RK4 aux cas des systèmes différentiels : en effet, dans la fonction `iteration`, les variables x, $k1$,... $k4$ représentent à présent des listes. Or une instruction comme $x + k1/2$ renverra un message d'erreur. En effet, la division d'une liste par un entier n'est pas définie ; de plus, l'addition de deux listes effectue la concaténation des listes, et non l'addition des listes composante par composante. Si on souhaite utiliser la classe RK4 sans modification, il faut alors travailler non plus avec des listes, mais avec des objets supportant la vectorisation, comme c'est le cas des tableaux de la librairie de tierce partie NumPy ; en outre, cette option a l'avantage de réduire de manière significative les temps de calcul. Cette option sera présentée dans l'exercice 51.
Dans un but pédagogique, faisons ici le choix de réécrire une nouvelle classe adaptant la méthode de RUNGE-KUTTA à des vecteurs représentés par des listes.

ode.py

```
class RK4_systeme(ODE):
    def iteration(self):
        f, h = self.f, self.h
        [t, x] = self.liste[self.indice]
        k1 =  [h * u for u in f(t, x)]
```

```
        v = [w[0] + w[1]/2 for w in zip(x, k1)]
        k2 =  [h * u for u in f(t + h/2, v)]
        v = [w[0] + w[1]/2 for w in zip(x, k2)]
        k3 =  [h * u for u in f(t + h/2, v)]
        v = [w[0] + w[1] for w in zip(x, k3)]
        k4 =  [h * u for u in f(t + h/2, v)]
        v = [w[0] + (w[1] + 2*w[2] + 2*w[3] + w[4])/6
            for w in zip(x, k1, k2, k3, k4)]
        return [t + h, v]
```

On l'applique ensuite à l'oscillateur de VAN DER POL donné plus haut et on obtient la figure 4.1b :

ode.py

```
import PostScript as ps

def f(t, x): return [x[1], 0.5 * x[1] - x[0] - x[1]**3]
def horscadre(x): return abs(max(x)) > 4

a, b, h, boite = 0, 30, 0.01, [-3.5, -3.5, 3.5, 3.5]
sol = RK4_systeme(f, h)
sol.CI(0, [0.01, 0.01])
solution = [x[1] for x in sol.resolution(a, b, horscadre)]
courbes = [[solution, (1, 0, 0)]]

b = 15
for k in ps.srange(0, 20, 0.4):
    sol = RK4_systeme(f, h)
    sol.CI(0, [0.3*k-3, 3])
    solution = [x[1] for x in sol.resolution(a, b, horscadre)]
    courbes.append([solution, (k/20, 0.32, 0.48-k/44)])
    sol = RK4_systeme(f, h)
    sol.CI(0, [3-0.3*k, -3])
    solution = [x[1] for x in sol.resolution(a, b, horscadre)]
    courbes.append([solution, (k/20, 0.32, 0.48-k/44)])

ps.plot("syst_autonome", boite, 40, courbes, etiquette=False)
```


3.3.3 Un exemple d'équation différentielle linéaire d'ordre 2

On rappelle qu'une équation différentielle linéaire d'ordre n peut toujours s'écrire comme un système différentiel d'ordre 1. Illustrons ceci dans le cas $n = 2$: en notant :

$$\mathrm{A}(t) = \begin{pmatrix} 0 & 1 \\ -b(t) & -a(t) \end{pmatrix}, \quad \mathrm{X}(t) = \begin{pmatrix} x(t) \\ x'(t) \end{pmatrix}, \quad \mathrm{C}(t) = \begin{pmatrix} 0 \\ c(t) \end{pmatrix}$$

on a les équivalences

$$\forall t \in \mathrm{I} \quad x'' + a(t)x' + b(t)x = c(t) \iff \forall t \in \mathrm{I} \quad \begin{cases} x' &= & x' \\ x'' &= -bx - ax' + c \end{cases}$$

$$\iff \forall t \in \mathrm{I} \quad \frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} x \\ x' \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -b & -a \end{pmatrix} \cdot \begin{pmatrix} x \\ x' \end{pmatrix} + \begin{pmatrix} 0 \\ c \end{pmatrix}$$

$$\iff \forall t \in \mathrm{I} \quad \mathrm{X}' = \mathrm{A}(t) \cdot \mathrm{X} + \mathrm{C}(t)$$

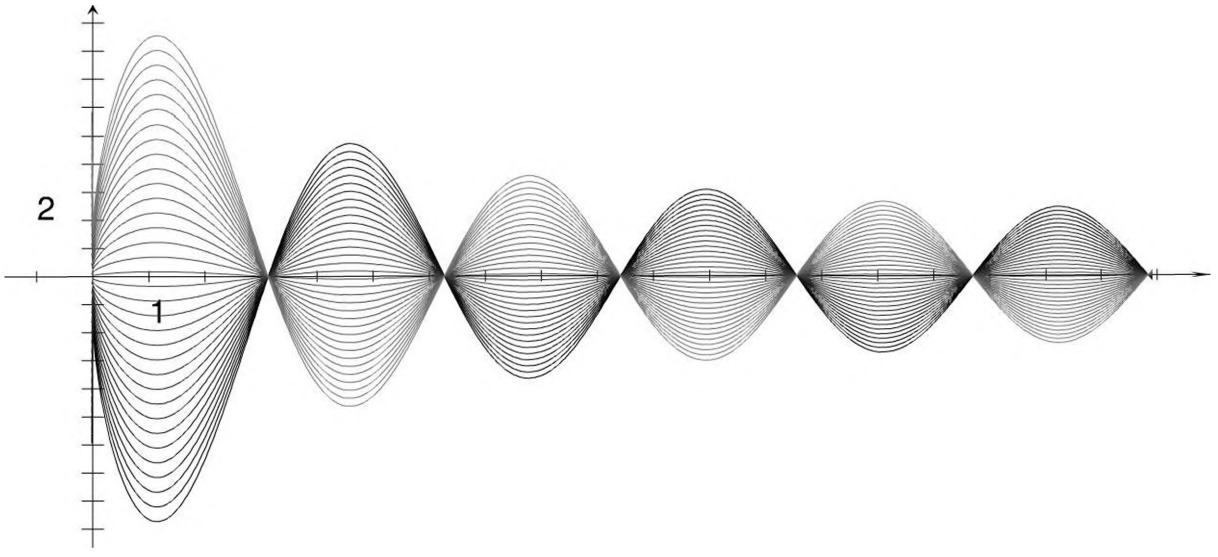
Considérons, par exemple, l'équation différentielle du 2e ordre dite de BESSEL :

$$t^2x''(t) + tx'(t) + \left(t^2 - \alpha^2\right)x(t) = 0 \iff \mathrm{X}'(t) = \mathrm{A}(t) \cdot \mathrm{X}(t) \quad \text{avec} \quad \mathrm{A}(t) = \begin{pmatrix} 0 & 1 \\ \frac{\alpha^2 - t^2}{t^2} & -\frac{1}{t} \end{pmatrix}$$

Traçons quelques solutions pour $\alpha = \frac{1}{2}$ à l'aide de la classe `RK4_systeme` :

ode.py

```
import PostScript as ps

def f(t, x):
    alpha = 0.5
    return [x[1], (alpha**2 - t**2) / t**2 * x[0] - x[1] / t]

a, b, h, boite = 0.01, 18.9, 0.01, [-1.5, -9.2, 19, 9.2]
courbes = []

for k in ps.srange(0, 20, 0.6):
    sol = RK4_systeme(f, h); sol.CI(a, [0.2*k-2, 0])
    solution = [[x[0], x[1][0]] for x in sol.resolution(a, b)]
    courbes.append([solution, (k/20, 0.32, 0.48-k/44)])

ps.plot("bessel", boite, 17, courbes, ratioY=0.5, gradH=0.2)
```



3.3.4 Inconvénients de la méthode de Runge-Kutta

La méthode de Runge-Kutta est appréciée par sa simplicité de mise en œuvre ; en effet, elle ne nécessite que quatre évaluations de f par pas, tout en offrant une bonne précision.
Cependant, aucune méthode numérique n'étant parfaite... elle souffre de deux inconvénients. D'une part, on ne dispose d'aucune indication sur l'erreur de troncature au cours des calculs. D'autre part, elle présente des problèmes de stabilité dès que le second membre varie rapidement (problèmes raides).
Si l'on souhaite surveiller la précision, une solution simple, mais coûteuse, consiste à conduire deux calculs simultanément, l'un avec un pas h et l'autre avec un pas $2h$. Tant que la différence entre les deux résultats ne dépasse pas une certaine valeur, on admet que l'erreur de troncature est acceptable. Sinon, on repart avec un pas plus petit.

4 Interpolation polynomiale

L'*interpolation* consiste à remplacer une fonction difficile à calculer numériquement ou une fonction empirique (dont on ne connaît les valeurs qu'en quelques points) par une fonction plus simple, par exemple une fonction polynomiale.
Soit à approcher une fonction f sur un segment $[a, b]$ par un polynôme P. On suppose connaître les valeurs de la fonction aux points d'une subdivision $(x_0, \ldots, x_n)$ du segment $[a, b]$.
La première idée consiste à décomposer le polynôme P dans la base canonique de $\mathbb{R}_n[\mathrm{X}]$, puis à exprimer les conditions d'interpolation :

$$\mathrm{P} = a_0 + a_1\mathrm{X} + \cdots + a_n\mathrm{X}^n \Longrightarrow \begin{cases} a_0 + a_1 x_0 + \cdots + a_n x_0^n = f(x_0) \\ a_0 + a_1 x_1 + \cdots + a_n x_1^n = f(x_1) \\ \qquad\qquad\qquad\qquad\qquad\quad \vdots \\ a_0 + a_1 x_n + \cdots + a_n x_n^n = f(x_n) \end{cases}$$

Mais en pratique, la résolution de ce système est trop lourd.
On pourrait alors penser à décomposer le polynôme P, non pas dans la base canonique, mais dans la base $(\mathrm{L}_i)_{i\in[\![0,n]\!]}$ des polynômes interpolateurs de LAGRANGE définis par

$$\forall i \in [\![0, n]\!], \quad \mathrm{L}_i = \prod_{\substack{k\in[\![0,n]\!] \\ k\neq i}} \left(\frac{\mathrm{X} - x_k}{x_i - x_k}\right)$$

Dans cette base, le polynôme P se décompose simplement sous la forme

$$\mathrm{P} = f(x_0)\mathrm{L}_0 + \cdots + f(x_n)\mathrm{L}_n$$

Cependant, cette méthode souffre de deux limitations : d'une part, elle sera assez sensible aux erreurs d'arrondi vu le grand nombre de multiplications à effectuer ; d'autre part, on ne peut pas calculer par récurrence la suite des polynômes L_i : dès qu'on rajoute un point à la subdivision, il faut recalculer tous les polynômes de la base.
La méthode utilisée le plus couramment en analyse numérique est celle des *différences divisées* ; elle consiste à décomposer le polynôme P dans la base des polynômes de NEWTON. Cette méthode a déjà été utilisée à la section 2.3.4 du chapitre 3 pour le calcul des logarithmes.

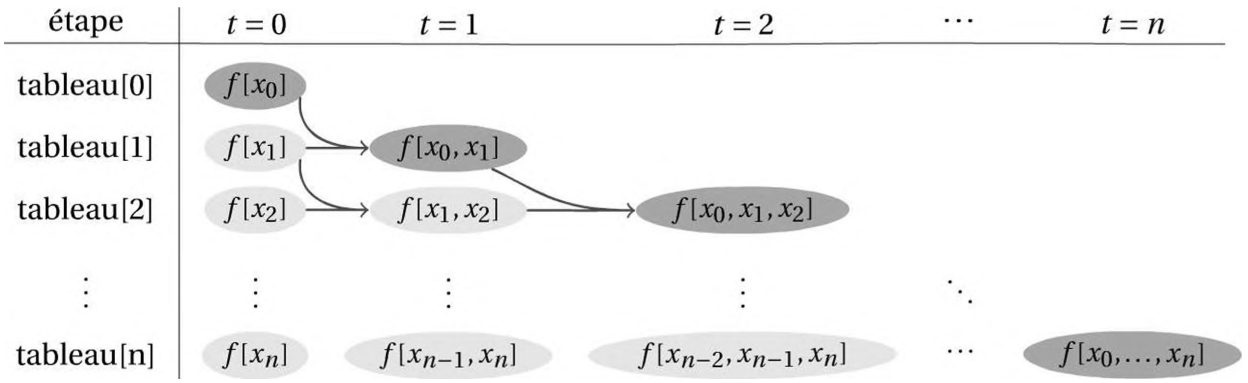
On peut montrer (cf. [Dem96]) que le polynôme interpolateur P se décompose de la manière suivante :

$$\mathrm{P} = f(x_0) + \sum_{k=1}^{n} f[x_0, \ldots, x_k](\mathrm{X} - x_0) \cdots (\mathrm{X} - x_k)$$

où les différences divisées sont définies par la relation de récurrence

$$f[x_0] = f(x_0) \quad \text{et} \quad \forall k \geq 1, \quad f[x_0, \ldots, x_k] = \frac{f[x_1, \ldots, x_k] - f[x_0, \ldots, x_{k-1}]}{x_k - x_0}$$

Pour programmer cet algorithme, on remplit un tableau triangulaire à double entrée : à chaque étape, on remplit une colonne supplémentaire.



Une fois le tableau rempli, on extrait les termes diagonaux $c_0, \ldots, c_n$; puis on utilise l'algorithme de HORNER pour calculer les valeurs du polynôme d'interpolation p_n en un point x :

$$p_n(x) = c_0 + (x - x_0)\,(c_1 + (x - x_1)\,(c_2 + (x - x_2)\,(c_3 + \cdots + (x - x_{n-1})c_n)))$$

Le programme peut s'écrire comme ceci :

interpolation.py

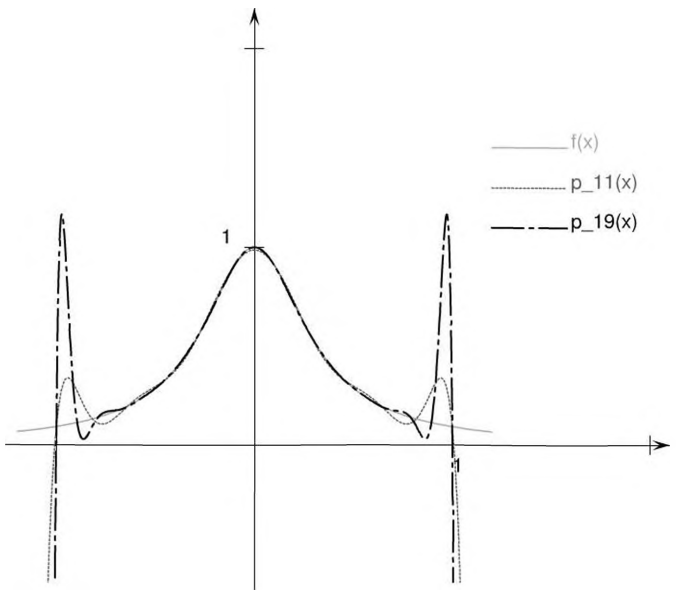
```
def diffDiv(liste):
    taille = len(liste)
    tableau = [[x[1]] for x in liste]
    for t in range(1, taille):
        for i in range(t, taille):
            diff = ((tableau[i][t-1] - tableau[i-1][t-1])
                    / (liste[i][0] - liste[i-t][0]))
            tableau[i].append(diff)
    return [ligne[-1] for ligne in tableau]

def interpol(x, liste):
    tableau = diffDiv(liste)
    valeur = tableau[-1]
    for k in range(len(liste)-2, -1, -1):
        valeur *= x - liste[k][0]
        valeur += tableau[k]
    return valeur
```

On considère à présent la fonction $f : x \mapsto \frac{1}{1+10x^2}$. Que se passe-t-il si l'on essaie d'en donner une approximation polynomiale par interpolation en des points régulièrement répartis dans l'intervalle $[-1,1]$?

interpolation.py

```
from PostScript import plot, nrange
from interpolation import interpol

def f(x): return 1/(1+10*x**2)
numpoints, a, b, zoom, nomFichier = 1000, -1.2 , 1.2, 100, "runge"
rouge, vert, bleu, boite = (1,0,0), (0,1,0), (0,0,1), [-1.5, -0.7, 1.7, 2.1]

liste_f = [[x, f(x)] for x in nrange(a, b, numpoints)]

subdiv = nrange(-1, 1, 11)
l = [[x, f(x)] for x in subdiv]
liste_p1 = [[x, interpol(x, l)] for x in nrange(a, b, numpoints)]

subdiv = nrange(-1, 1, 19)
l = [[x, f(x)] for x in subdiv]
liste_p2 = [[x, interpol(x, l)] for x in nrange(a, b, numpoints)]

plot(nomFichier, boite, zoom, [[liste_f, bleu], [liste_p1, vert],
    [liste_p2, rouge]], ['1/(1+10x^2)', [1.1, 0.1], bleu],
    ['p_11(x)', [1.1, -0.3], vert], ['p_19(x)', [1.1, 1], rouge])
```

Avec 11 points, le polynôme d'interpolation approche relativement bien la fonction tant que l'on ne s'approche pas des bornes du segment $[-1,1]$. On serait alors tenté d'augmenter le nombre de points d'interpolation. Or comme on le voit sur la figure suivante, avec 19 points d'interpolation, le mal empire ! C'est le fameux phénomène de RUNGE.



Comme le polynôme d'interpolation p_n s'écarte de la fonction aux bords du segment sur lequel on travaille, la bonne idée est de modifier la subdivision en concentrant légèrement les points de la subdivision aux extrémités du segment.

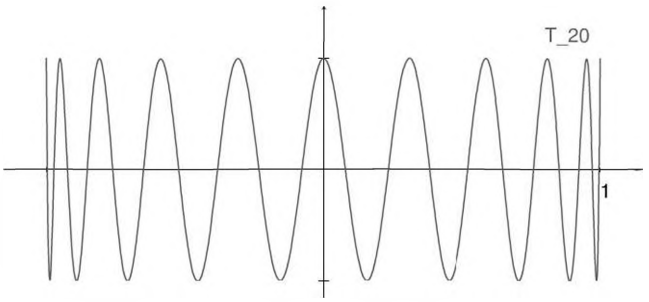
Si on travaille sur l'intervalle $[-1,1]$, on utilise alors généralement les n racines du polynôme de TCHEBYCHEV, de première espèce, d'ordre n défini par

$$T_n(x) = \cos(n \arccos x)$$

Ses racines sont alors données par

$$\sigma = \left\{ \cos \frac{2i+1}{n}\pi \;\middle|\; i \in [\![0, n-1]\!] \right\}$$

En effet, comme on le constate sur le tracé du polynôme T_{20}, la répartition des racines est plus dense au voisinage des bornes de l'intervalle de travail :



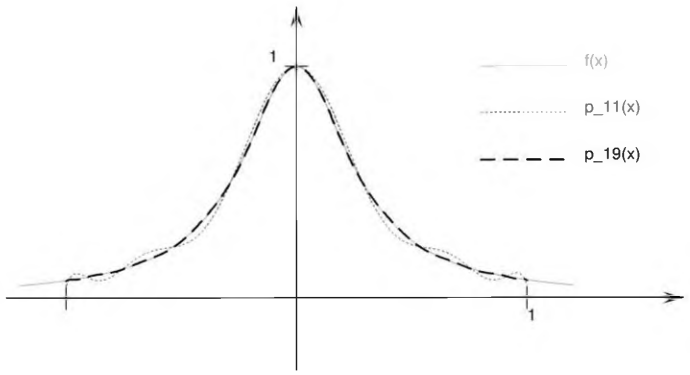
interpolation.py

```
from math import *
from postscript import plot, nrange
numpoints, a, b, zoom, boite = 1000, - 1 , 1, 140, [-1.1, -1.1, 1.1, 1.4]
liste = [[x, cos(20*acos(x))] for x in nrange(a, b, numpoints)]
plot("tchebychev", boite, zoom, [[liste]], ['T_20', [0.8, 1.15], bleu],
    ratioY=.4, gradH=.02)
```

Complétons alors notre fichier `runge.py`. En reprenant les essais précédents, remplaçons la subdivision régulière de $[-1,1]$ par la subdivision σ :

interpolation.py

```
from math import *
def tcheby(n): return [cos(pi * (2*i+1) / (2*n)) for i in range(n)]

subdiv = tcheby(11); l = [[x, f(x)] for x in subdiv]
liste_p1 = [[x, interpol(x, l)] for x in nrange(a, b, numpoints)]

subdiv = tcheby(19); l = [[x, f(x)] for x in subdiv]
liste_p2 = [[x, interpol(x, l)] for x in nrange(a, b, numpoints)]

boite = [-1.5, -0.3, 1.7, 1.2] # xmin, ymin, xmax, ymax
plot("tcheby", boite, zoom, [[liste_f, bleu], [liste_p1, vert],
  [liste_p2, rouge]], ['1/(1+10x^2)', [1.1, 0.1], bleu],
  ['p_11(x)', [1.1, -0.1], vert], ['p_19(x)', [1.1, 0.3], rouge])
```



Cette fois-ci, le polynôme d'interpolation fournit une approximation de f tout à fait fidèle. Plus précisément, on peut prouver que si f est lipschitzienne sur l'intervalle $[-1,1]$, la suite des polynômes de Tchebychev converge uniformément vers f sur cet intervalle (pour une démonstration de ce résultat, se reporter à [Dem96]).

Notons que si l'on souhaite effectuer une interpolation sur un segment $[a,b]$ quelconque, on se ramène à l'étude précédente par le changement de variable affine

$$\varphi : t \mapsto \frac{t-a}{b-a}$$

Avant de clore cette section, signalons que l'on pourrait chercher à approcher la fonction f par des polynômes autres que des polynômes d'interpolation :

- les polynômes de BERNSTEIN associés à la fonction f :

$$\mathrm{B}_n(f)(x) = \sum_{k=0}^{n} f\left(\frac{k}{n}\right)\binom{n}{k} x^k (1-x)^{n-k}$$

- le polynôme de degré au plus n obtenu par la méthode des moindres carrés, c'est-à-dire le polynôme qui réalise le minimum suivant :

$$\inf\left\{\int_{-1}^{1} |f(x) - \mathrm{P}(x)|^2 \,\mathrm{d}x \,\middle|\, \mathrm{P} \in \mathbb{R}_n[\mathrm{X}]\right\}$$

  Si on munit l'espace vectoriel $\mathcal{C}^0([-1,1])$ du produit scalaire $(f|g) = \int_{-1}^{1} fg$, le polynôme recherché est la projection orthogonale de la fonction f sur le sous-espace vectoriel $\mathbb{R}_n[\mathrm{X}]$. La détermination de ce polynôme se ramène à la résolution d'un système linéaire : en effet, P est le projeté de f si et seulement si la fonction $(\mathrm{P} - f)$ est orthogonale à chacun des vecteurs de la base canonique de $\mathbb{R}_n[\mathrm{X}]$.
  Encore faut-il disposer d'une méthode d'intégration numérique...
- etc.

5 Dérivation numérique

Lorsqu'une fonction est trop compliquée pour que le calcul analytique de sa dérivée soit pratique, on peut obtenir une valeur approchée de la dérivée en calculant un taux d'accroissement τ_h :

$$f'(x) \approx \underbrace{\frac{f(x+h) - f(x)}{h}}_{\tau_x(h)} \quad \text{avec } h \text{ petit}$$

En théorie, plus h est petit, plus $\tau_x(h)$ est proche de $f'(x)$. Mais, eu égard aux problèmes d'arrondi, il ne faut pas prendre h trop petit. En effet, Le calcul de $\tau_x(h)$ se fait avec un précision absolue de l'ordre de $2\varepsilon \cdot |f(x)|/h$ (où $\varepsilon \approx 10^{-15}$) ; par ailleurs, d'après l'inégalité de TAYLOR-LAGRANGE, on a

$$|\tau_x(h) - f'(x)| \leq \frac{h}{2}\|f''\|_\infty$$

L'inégalité triangulaire entraîne alors

$$\left|f'(x) - \widetilde{\tau_x(h)}\right| \leq \frac{h}{2}\|f''\|_\infty + \frac{2\varepsilon\,|f(x)|}{h}$$

Une étude rapide de la fonction h: $h \mapsto \frac{h}{2}\|f''\|_\infty + \frac{2\varepsilon|f(x)|}{h}$ montre que cette fonction possède un minimum absolu sur $\mathbb{R}^+$ atteint en

$$h = 2\sqrt{\varepsilon \frac{|f(x)|}{\|f''\|_\infty}}$$

Pour une fonction suffisamment régulière, il est donc judicieux de choisir une valeur de h qui soit de l'ordre de $\sqrt{\varepsilon}$, c'est-à-dire de l'ordre de 10^{-8}.
L'écriture d'une classe pour la dérivation numérique peut être :

derivation.py

```
class Diff(object):
    def __init__(self, f, h=1e-8):
        self.f = f
        self.h = h

class DiffAvant1(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x))/h
```

Comparons pour différentes valeurs de h les valeurs données pour la dérivée de la fonction sinus en 0 :

derivation.py

```
import math as m
print('{:^7} | {:^20}'.format('h', 'cos')); print('-'*28)
for n in range(1, 13):
    x, h = 0, 10**(-n); cos = DiffAvant1(m.sin, h)
    print('{:7.1e} | {:.15f}'.format(h, cos(x)))
```

```
   h    |         cos
----------------------------
1.0e-01 | 0.998334166468282
1.0e-02 | 0.999983333416666
1.0e-03 | 0.999999833333342
1.0e-04 | 0.999999998333333
1.0e-05 | 0.999999999983333
1.0e-06 | 0.999999999999833
```

```
   h    |         cos
----------------------------
1.0e-07 | 0.999999999999998
1.0e-08 | 1.000000000000000
1.0e-09 | 1.000000000000000
1.0e-10 | 1.000000000000000
1.0e-11 | 1.000000000000000
1.0e-12 | 1.000000000000000
```

On constate qu'à partir de $h = 10^{-8}$ la valeur donnée est exacte.
Notons qu'il existe d'autres formules d'approximation de la dérivée, basées sur des calcul de différences finies et qui permettent d'obtenir un ordre d'approximation plus élevé :

$$f'(x) = \frac{f(x) - f(x-h)}{h} + \mathrm{O}(h) \qquad \text{différence arrière d'ordre 1}$$

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathrm{O}(h^2) \qquad \text{différence centrée d'ordre 2}$$

$$f'(x) = \frac{-f(x+2h) + 4f(x+h) - 3f(x)}{2h} + \mathrm{O}(h^2) \qquad \text{différence avant d'ordre 2}$$

$$f'(x) = \frac{3f(x) - 4f(x-h) + f(x-2h)}{2h} + \mathrm{O}(h^2) \qquad \text{différence arrière d'ordre 2}$$

$$f'(x) = \frac{-\frac{1}{6}f(x+2h) + f(x+h) - \frac{1}{2}f(x) - \frac{1}{3}f(x-h)}{h} + \mathrm{O}(h^3) \qquad \text{différence avant d'ordre 3}$$

etc.

Pour les programmer, il suffit d'ajouter une classe pour chacune dans le fichier `derivation.py` :

derivation.py

```
class DiffCentre2(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x-h))/(2*h)
```

Comparons, par exemple, l'efficacité de la formule de la différence centrée d'ordre 2 pour le calcul de la dérivée de la fonction racine carrée :

derivation.py

```
import math as m

def f(x): return m.sqrt(1+x)

for n in range(1, 13):
    x, h = 0, 10**(-n)
    g1 = DiffAvant1(f, h)
    g2 = DiffCentre2(f, h)
    print('{0:7.1e} | {1:.15f} | {2:.15f}'.format(h,g1(x),g2(x)))
```

```
1.0e-01 | 0.488088481701516 | 0.500627750598189
1.0e-02 | 0.498756211208895 | 0.500006250273449
1.0e-03 | 0.499875062460964 | 0.500000062500006
1.0e-04 | 0.499987500623966 | 0.500000000624445
1.0e-05 | 0.499998750003172 | 0.500000000003276
1.0e-06 | 0.499999875058776 | 0.500000000014378
1.0e-07 | 0.499999988079480 | 0.500000000291934
1.0e-08 | 0.499999996961265 | 0.500000002512380
1.0e-09 | 0.500000041370185 | 0.500000041370185
1.0e-10 | 0.500000041370185 | 0.500000041370185
1.0e-11 | 0.500000041370185 | 0.500000041370185
1.0e-12 | 0.500044450291171 | 0.500044450291171
```

Cette fois-ci, on voit apparaître très nettement la perte de précision lorsque h est trop petit.

6 Intégration numérique

Les méthodes les plus courantes pour approcher numériquement une intégrale reposent sur une formule de la forme

$$\mathrm{I} = \int_a^b f(x)\,\mathrm{d}x = \sum_{j=1}^{n} w_j f(x_j) + \mathrm{E} \qquad (*)$$

où les w_j sont appelés les *poids*, les x_j les *pivots* ou *nœuds*. Le nombre E est l'erreur de troncature.

Les méthodes classiques d'intégration numérique se répartissent en deux types :

- les algorithmes de NEWTON-COTES où les pivots sont équidistants. Les n poids sont donc les seuls paramètres ajustables ; ils sont alors choisis pour rendre la méthode exacte (c'est-à-dire telle que $\mathrm{E} = 0$) pour n'importe quel polynôme de degré au plus $n-1$.

- les algorithmes de quadrature de GAUSS où les pivots et les poids sont ajustables ; ces $2n$ paramètres sont alors choisis pour rendre la méthode exacte pour n'importe quel polynôme de degré au plus $2n-1$.

Si les méthodes de GAUSS sont plus performantes (en terme de précision pour un même temps de calcul) que les méthodes de NEWTON-COTES, elles sont en revanche plus lourdes à programmer.
Avant de décrire quelques méthodes d'intégration numérique, on peut commencer par écrire une super-classe qui repose sur la formule $(*)$:

integration.py

```
class Integration(object):
    def __init__(self, a, b, n):
        self.a, self.b, self.n = a, b, n
        self.poids_pivots = self.quadrature()

    def quadrature(self):
        raise NotImplementedError

    def integrale(self, f):
        return sum(w * f(x) for w, x in self.poids_pivots)
```

6.1 Méthodes de Newton-Cotes composées

Pour donner une valeur approchée de l'intégrale I, une première idée est de remplacer la fonction f par un polynôme P qui interpole f en plusieurs points. Cependant, dès qu'on augmente le nombre de pivots, on risque fort de se retrouver confronté au phénomène de RUNGE.
On modifie alors l'idée de départ ainsi : on commence par subdiviser régulièrement l'intervalle d'intégration en n sous-intervalles $[x_j, x_{j+1}]$ où

$$x_0 = a, \quad x_n = b, \quad \text{et} \quad \forall j \in [\![0, n]\!], \quad x_j = a + jh, \quad \text{avec } h = \frac{b-a}{n}$$

Ensuite, on remplace f par un polynôme P_j qui interpole f sur chacun des « petits » segments $[x_j, x_{j+1}]$.

- Si P_j interpole f au point x_j, alors le graphe de P_j est une droite horizontale : c'est la méthode des rectangles.
- Si P_j interpole f aux points x_j et x_{j+1}, alors le graphe de P_j est une droite affine : c'est la méthode des trapèzes.
- Si P_j interpole f aux points x_j, $\frac{x_j+x_{j+1}}{2}$ et x_{j+1}, alors le graphe de P_j est une parabole : c'est la méthode de SIMPSON 1/3.

Enfin, on somme les intégrales de chaque polynôme P_j sur $[x_j, x_{j+1}]$, et on obtient une valeur approchée de I :

$$\mathrm{I} = \sum_{j=0}^{n-1} \int_{x_j}^{x_{j+1}} f(x)\,\mathrm{d}x = \sum_{j=0}^{n-1} \left(\int_{x_j}^{x_{j+1}} \mathrm{P}_j(x)\,\mathrm{d}x + \mathrm{E}_j \right) \approx \sum_{j=0}^{n-1} \int_{x_j}^{x_{j+1}} \mathrm{P}_j(x)\,\mathrm{d}x$$

Comme ces méthodes ont déjà été introduites à la section 3, on ne s'attarde que sur la méthode de SIMPSON 1/3 composée.
On se place sur l'intervalle $[x_j, x_{j+1}]$ et on note $\xi_j = \frac{x_j+x_{j+1}}{2}$; alors d'après les formules des différences divisées (cf. § 4), le polynôme d'interpolation de f aux points (x_j, ξ_j, x_{j+1}) est donné par

$$\mathrm{P}_j(x) = f[x_j] + f[x_j, \xi_j](x - x_j) + f[x_j, \xi_j, x_{j+1}](x - x_j)(x - \xi_j)$$

On en déduit, après calculs,

$$\int_{x_j}^{x_{j+1}} \mathrm{P}_j(x)\,\mathrm{d}x = \frac{h}{6}\left(f(x_j) + 4f(\xi_j) + f(x_{j+1})\right)$$

Par sommation on obtient

$$\begin{aligned}
\mathrm{I} &\approx \frac{b-a}{6n}\left(f(x_0) + 4f\left(\tfrac{x_0+x_1}{2}\right) + 2f(x_1) + 4f\left(\tfrac{x_1+x_2}{2}\right) + \cdots + 2f(x_{n-1}) + 4f\left(\tfrac{x_{n-1}+x_n}{2}\right) + f(x_n)\right) \\
&\approx \frac{b-a}{6n}\left(f(x_0) + 2\sum_{j=1}^{n-1} f(x_j) + 4\sum_{j=0}^{n-1} f\left(\tfrac{x_j+x_{j+1}}{2}\right) + f(x_n)\right) \\
&\approx \frac{b-a}{6n}\sum_{k=0}^{2n} w_k f(a_k) \quad \text{avec} \quad a_k = a + k\cdot\frac{b-a}{2n} \text{ et } w_k = \begin{cases} 1 \text{ si } k = 0 \text{ ou } 2n \\ 4 \text{ si } 0 < k < 2n \text{ et } k \text{ impair} \\ 2 \text{ si } 0 < k < 2n \text{ et } k \text{ pair} \end{cases}
\end{aligned}$$

On peut montrer (cf. [Dem96]) que l'erreur commise est majorée par

$$|\mathrm{E}| \le \frac{1}{2880}\cdot h^4 \cdot \left\| f^{(4)} \right\|_\infty \cdot (b-a) \qquad (**)$$

On dit alors que la méthode est d'ordre 4 ; noter que la méthode est exacte pour tout polynôme de degré au plus 3.
Programmons les trois méthodes sus-mentionnées :

integration.py

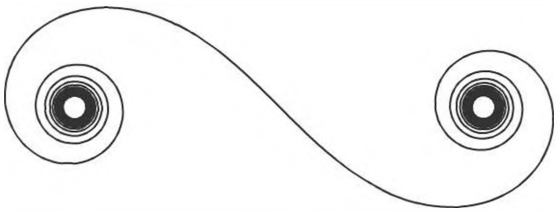
```
class Rectangles(Integration):
    def quadrature(self):
        a, b, n = self.a, self.b, self.n
        h = (b-a) / n
        return [(h, a + i*h) for i in range(n)]

class Trapezes(Integration):
    def quadrature(self):
        a, b, n = self.a, self.b, self.n
        h = (b-a) / n
        return [(h, a + (i+0.5)*h) for i in range(n)]

class Simpson(Integration):
    def coeff(self, i, N):
        if i == 0 or i == N:
            return 1
        elif i % 2 == 1:
            return 4
```

```
        else:
            return 2

    def quadrature(self):
        a, b, n = self.a, self.b, self.n
        if n % 2 != 1:
            n += 1
        h = (b-a) * 0.5 / n
        return [(h/3*self.coeff(i, 2*n), a + i*h) for i in range(2*n+1)]
```

Exercice 49. Représenter la spirale de CORNU définie par

$$x(t) = \int_0^t \cos\left(u^2\right) \mathrm{d}u \quad \text{et} \quad y(t) = \int_0^t \sin\left(u^2\right) \mathrm{d}u$$

À une rotation de $\frac{\pi}{4}$ près, on obtient :



Solution.

integration.py

```
from math import *
from PostScript import plot, nrange

def z(t):
    s = Simpson(0, t, 1000)
    x = lambda u:cos(u**2)
    y = lambda u:sin(u**2)
    return [s.integrale(x), s.integrale(y)]

numpoints, a, b, zoom, boite = 1000, -10 , 10, 100, [-1, -1, 1, 1]
liste = [z(x) for x in nrange(a, b, numpoints)]
plot("cornu", boite, zoom, [[liste, (0, 0, 1)]], axes=False)
```

Pour une présentation de cette courbe, le lecteur pourra consulter le site de référence en matière de courbes et de surfaces : http://www.mathcurve.com/courbes2d/cornu/cornu.shtml

Pour diminuer le nombre de points à tracer, et donc le temps de calcul, on pourrait utiliser des courbes de BÉZIER avec comme points de contrôle les vecteurs vitesse (qui se calculent très simplement ici).

Pour conclure cette section, revenons un instant sur l'inégalité $(**)$: cette majoration de l'erreur possède un intérêt théorique indéniable ; néanmoins, elle ne permet pas de déterminer directement l'ordre du pas h à choisir pour obtenir une approximation de I avec une précision donnée. En effet, comment déterminer une borne de la dérivée quatrième de f ? Numériquement, ce problème est trop complexe. En pratique, on applique la méthode avec deux pas différents (h et $2h$ pour minimiser les évaluations de f) et on utilise la différence des deux approximations numériques comme estimation de l'erreur du moins bon résultat.

6.2 La méthode de Romberg

La méthode de ROMBERG utilise le procédé d'extrapolation de RICHARDSON pour accélérer la convergence de la méthode des trapèzes composée.
Soit f une fonction de classe $\mathcal{C}^\infty$ sur le segment $[a,b]$. On décompose le segment $[a,b]$ en n sous-intervalles égaux. Alors l'approximation de l'intégrale I de f sur ce segment par la méthode des trapèzes vaut

$$\mathrm{T}(h) = h\left(\frac{1}{2}f(x_0) + \sum_{j=1}^{n-1} f(x_j) + \frac{1}{2}f(x_n)\right) \quad \text{où} \quad \forall j \in [\![0,n]\!], \quad x_j = a + jh, \quad \text{et} \quad h = \frac{b-a}{n}$$

On peut alors montrer (cf. [Dem96]) à l'aide de la formule d'EULER-MACLAURIN que l'erreur commise dans la méthode des trapèzes possède un développement limité à tout ordre de la forme

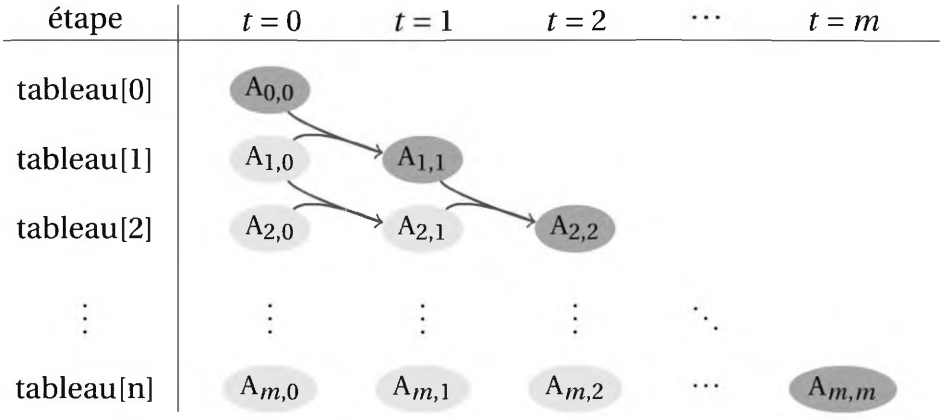
$$\mathrm{E}(h) = a_1 h^2 + a_2 h^4 + \cdots + a_{k-1} h^{2k-2} + \mathrm{O}\left(h^{2k}\right) \quad \text{où} \quad \mathrm{E}(h) = \mathrm{T}(h) - \int_a^b f(t)\,\mathrm{d}t$$

De plus, les a_i ne dépendent ni de n, ni de h.
L'idée est alors de faire une combinaison linéaire de $\mathrm{T}(h)$ et $\mathrm{T}(h/2)$ pour annuler le premier terme du développement de E :

$$\begin{array}{r} \mathrm{T}(h) = \int_a^b f(t)\,\mathrm{d}t + a_1 h^2 + a_2 h^4 + \cdots + a_{k-1} h^{2k-2} + \mathrm{O}\left(h^{2k}\right) \\ \mathrm{T}\left(\frac{h}{2}\right) = \int_a^b f(t)\,\mathrm{d}t + a_1 \frac{h^2}{2^2} + a_2 \frac{h^4}{2^4} + \cdots + a_{k-1} \frac{h^{2k-2}}{2^{2k-2}} + \mathrm{O}\left(h^{2k}\right) \\ \hline \frac{1}{3}\left(4\mathrm{T}\left(\frac{h}{2}\right) - \mathrm{T}(h)\right) = \int_a^b f(t)\,\mathrm{d}t + \underbrace{b_2 h^4 + \cdots + b_{k-1} h^{2k-2} + \mathrm{O}\left(h^{2k}\right)}_{\mathrm{O}(h^4)} \end{array}$$

On remarque qu'ainsi, on a gagné un ordre dans le développement de l'erreur. C'est un bon exercice de vérifier que la nouvelle expression obtenue correspond à la méthode de SIMPSON.
Le procédé précédent se généralise aisément : on utilise des dichotomies successives avec des pas de la forme $h = \frac{b-a}{2^m}$ et on remplit un tableau triangulaire comprenant les nombres :

$$\mathrm{A}_{m,0} = \mathrm{T}\left(\frac{b-a}{2^m}\right) \quad \text{et} \quad \forall n \in [\![1,m]\!], \ \mathrm{A}_{m,n} = \frac{4^n \mathrm{A}_{m,n-1} - \mathrm{A}_{m-1,n-1}}{4^n - 1}$$



Le dernier élément de la diagonale principale $A_{m,m}$ est alors la meilleure approximation de l'intégrale I que l'on puisse obtenir en évaluant f en 2^m points.
En pratique, on commence par remplir la première colonne en se servant de la relation de récurrence :

$$A_{k,0} = \frac{1}{2}A_{k-1,0} + A'_{k,0} \quad \text{où} \quad A'_{k,0} = \frac{b-a}{2^k}\sum_{j=1}^{2^{k-1}} f\left(a + (2j+1)\frac{b-a}{2^k}\right)$$

En effet, en notant $h = \frac{b-a}{2^k}$, on a

$$\begin{aligned} A_{k,0} &= h\left(\frac{1}{2}f(a) + \sum_{j=1}^{2^k-1} f\left(a+jh\right) + \frac{1}{2}f(b)\right) \\ &= h\left(\frac{1}{2}f(a) + \sum_{p=1}^{2^{k-1}-1} f\left(a+2ph\right) + \sum_{p=0}^{2^{k-1}-1} f\left(a+(2p+1)h\right) + \frac{1}{2}f(b)\right) \\ &= \frac{1}{2}A_{k-1,0} + \underbrace{h\sum_{j=0}^{2^{k-1}-1} f\left(a+(2j+1)h\right)}_{A'_{k,0}} \end{aligned}$$

Ceci permet de calculer une fois et une fois seulement les valeurs de f nécessaires.
Ensuite, on remplit les autres colonnes en utilisant la formule

$$\forall n \in [\![1, m]\!],\ A_{m,n} = \frac{4^n A_{m,n-1} - A_{m-1,n-1}}{4^n - 1}$$

La programmation de l'algorithme peut alors s'écrire comme suit :

integration.py

```
class Romberg(object):
    def __init__(self, a, b, m):
        self.a, self.b, self.m = a, b, m

    def integrale(self, f):
        a, b, m = self.a, self.b, self.m
        A = [[(b-a) * (f(a)+f(b)) * 0.5]]
        for k in range(1, m+1):
            h = (b-a) / 2**k
            Ap = h * sum([f(a + j*h) for j in range(1, 2**k, 2)])
            A.append([0.5 * A[k-1][0] + Ap])
        for t in range(1, m+1):
            for k in range(t, m+1):
                A[k].append((4**t * A[k][t-1] - A[k-1][t-1]) / (4**t-1))
        return A[m][m]
```

Exercice 50. Conjecturer l'ordre de la partie principale du développement asymptotique de la suite $u_n = \int_0^1 \ln\left(1+t^n\right)\mathrm{d}t$

Solution. Si u_n possède un développement de la forme $u_n = c_0 n^\alpha + c_1 n^{\alpha-1} + o\left(n^{\alpha-1}\right)$, alors nécessairement,

$$\alpha = \lim_n n\left(\frac{u_{n+1}}{u_n} - 1\right)$$

On peut donc conjecturer la valeur de α avec le script suivant :

integration.py

```
from math import *
m = Romberg(0, 1, 16)
def u(n): return m.integrale(lambda x:log(1+x**n))
def v(n): return n * (u(n+1)/u(n) - 1)
n, liste = 2**8, [1, 0]

while n <= 10**4 and abs(liste[0] - liste[1]) > 1e-10:
    liste = [v(n), v(n+1)]
    print("n={:4d} | v(n)={: .10f}".format(n, liste[0]))
    n *= 2
print("n*u(n)={:f}".format(n*u(n)))
```

```
n= 256 | v(n)=-0.9918769362          n=2048 | v(n)=-0.9989775146
n= 512 | v(n)=-0.9959222822          n=4096 | v(n)=-0.9994885018
n=1024 | v(n)=-0.9979570706          n=8192 | v(n)=-0.9997441893
n*u(n)=0.822412
```

Donc vraisemblablement, $\alpha = -1$. En fait, en utilisant un théorème d'intégration terme à terme, on peut prouver que $u_n \sim \frac{\pi^2}{12n}$. (Remarquer que $\pi^2/12 \approx 0.822467033424$).

6.3 Méthodes de Gauss

Comme nous l'avons dit en début de section, les méthodes de quadrature de GAUSS consistent à choisir non seulement les poids, mais aussi les pivots, pour que la méthode soit aussi précise que possible.
Considérons le cas particulier de la méthode de GAUSS-LEGENDRE qui permet de calculer une intégrale sur le segment $[-1, 1]$. Pour trouver la répartition optimale des pivots, il faut faire le détour par les polynômes orthogonaux ; pour l'exposé théorique, nous renvoyons une fois de plus à [Dem96], où il est notamment montré le fait suivant.
Si on cherche à rendre l'approximation

$$\mathrm{I} = \int_{-1}^{1} f(x)\,\mathrm{d}x = \sum_{j=1}^{n} w_j f(x_j) + \mathrm{E}$$

exacte pour tous les polynômes de degré inférieur à $2n-1$, alors il faut prendre

- comme pivots les n racines x_j du n-ième polynôme de LEGENDRE P_n ;
- comme poids les valeurs

$$w_j = \frac{2}{\left(1 - x_j^2\right)\left[\mathrm{P}_n'(x_j)\right]^2}$$

On rappelle que les polynômes de LEGENDRE peuvent être définis par la formule de Rodrigues :

$$P_n(x) = \frac{1}{2^n n!}\frac{d^n}{dx^n}\left[(x^2-1)^n\right]$$

ou par récurrence :

$$P_0(x) = 1, \quad P_1(x) = x \quad \text{et} \quad \forall n \in \mathbb{N}, \quad (n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x)$$

Pour le calcul de P'_n, on dispose également de la relation de récurrence :

$$\frac{x^2-1}{n}\frac{d}{dx}P_n(x) = xP_n(x) - P_{n-1}(x)$$

Le calcul des poids et des pivots s'en déduit :

Nbre de points n	Pivots x_i	Poids w_i
1	0	2
2	$\pm 1/\sqrt{3}$	1
3	0 $\pm\sqrt{3/5}$	$8/9$ $5/9$
4	$\pm\sqrt{\left(3-2\sqrt{6/5}\right)/7}$ $\pm\sqrt{\left(3+2\sqrt{6/5}\right)/7}$	$\frac{18+\sqrt{30}}{36}$ $\frac{18-\sqrt{30}}{36}$
5	0 $\pm\frac{1}{3}\sqrt{5-2\sqrt{10/7}}$ $\pm\frac{1}{3}\sqrt{5+2\sqrt{10/7}}$	$\frac{128}{225}$ $\frac{322+13\sqrt{70}}{900}$ $\frac{322-13\sqrt{70}}{900}$

Pour appliquer cette méthode au calcul d'une intégrale sur un segment quelconque, on effectue un changement de variable affine :

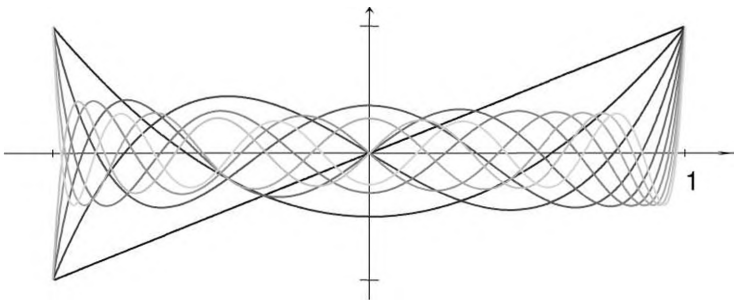
$$\int_a^b f(x)\,dx = \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{b-a}{2}x + \frac{a+b}{2}\right)dx$$

La formule de quadrature de GAUSS devient alors

$$\int_a^b f(x)\,dx \approx \frac{b-a}{2}\sum_{j=1}^{n} w_j f\left(\frac{b-a}{2}x_j + \frac{a+b}{2}\right)$$

Pour programmer cette méthode, nous proposons de calculer numériquement les poids et les pivots.

Dans un premier temps, on définit une fonction `legendre` qui calcule le m-ième polynôme de LEGENDRE, ainsi que son polynôme dérivé.



integration.py

```
def legendre(t, m):
    if m == 0:
        return 1
    elif m==1:
        return t
    else:
        p0, p1 = 1, t
        for k in range(1, m):
            p = ((2*k+1)*t*p1 - k*p0) / (1+k)
            p0, p1 = p1, p
        return p

from math import *
from PostScript import plot, nrange
numpoints, a, b, fichier, zoom = 1000, -1 , 1, "legendre", 120
boite = [-1.1, -1.1, 1.1, 1.1] # xmin, ymin, xmax, ymax
liste = [[[[x, legendre(x, n)] for x in nrange(a, b, numpoints)],
        (n/10, 0.32, 1 - n/16)] for n in range(1, 11)]
plot(fichier, boite, zoom, liste, gradH=.03, ratioY=.4)
```

Dans un deuxième temps, on utilise l'algorithme de NEWTON pour le calcul des racines positives de P_m, sachant que P_m a même parité que m. Pour calculer la i-ième racine positive, il faut une valeur approchée pour initialiser l'algorithme de NEWTON ; il est possible de montrer que la valeur $\cos\left(\frac{4i+3}{2m+1}\frac{\pi}{2}\right)$ convient.
Voici donc une programmation de la méthode de GAUSS-LEGENDRE :

integration.py

```
from math import pi, cos

class GaussLegendre(Integration):
    def gaussPivots(self, m, tol=1e-14):
        def legendre(t,m):
            p0 = 1; p1 = t
            for k in range(1, m):
                p = ((2*k+1)*t*p1 - k*p0) / (1+k)
                p0, p1 = p1, p
            dp = m*(p0 - t*p1) / (1 - t**2)
            return p, dp

        w = [0 for i in range(m)]    # poids
        x = [0 for i in range(m)]    # pivots
        nRoots = (m + 1)//2          # nombre de racines positives
        for i in range(nRoots):
            t = cos(pi*(i + 0.75)/(m + 0.5))    # racine approchée
            for j in range(30):
                p, dp = legendre(t, m)
                dt = -p / dp; t = t + dt        # méthode de Newton
                if abs(dt) < tol:
                    x[i] = t; x[m-i-1] = -t
```

```
                    w[i] = 2 / (1 - t**2) / (dp**2)
                    w[m-i-1] = w[i]
                    break
        return zip(w, x)

    def quadrature(self):
        a, b, n = self.a, self.b, self.n
        return [[(b-a)*w*0.5, (a+b)*0.5 + (a-b)*0.5*x]
                for w, x in self.gaussPivots(n)]
```

6.4 Comparatif des différentes méthodes

Il ne nous reste plus qu'à comparer la méthode de GAUSS-LEGENDRE aux méthodes précédentes, ce que nous faisons en cherchant à calculer une valeur approchée du nombre π à l'aide de la formule

$$\pi = 12\left(\int_0^{\frac{1}{2}} \sqrt{1-t^2}\,\mathrm{d}t - \frac{\sqrt{3}}{8}\right)$$

integration.py

```
print("{:>3} | {:^10} | {:^10} | {:^9} | {:^9} | {:^9}".format("n",
      "rectangles", "trapèzes", "Simpson", "Romberg", "Gauss"))
for i in range(1, 8):
    r, t = Rectangles(0, 0.5, 2**i), Trapezes(0, 0.5, 2**i)
    s, ro = Simpson(0, 0.5, 2**i), Romberg(0, 0.5, i)
    g = GaussLegendre(0, 0.5, 2**i)
    comp = [eval(m).integrale(lambda x:sqrt(1-x**2))
            for m in ['r', 't', 's', 'ro', 'g']]
    comp = [pi - 12*(c - sqrt (3)/8) for c in comp]
    print("{0:>3d} |  {1[0]: .2e} |  {1[1]: .2e} | {1[2]: .2e} "
          "| {1[3]: .2e} | {1[4]: .2e}".format(2**i, comp))
```

```
  n | rectangles |  trapèzes  |  Simpson  |  Romberg  |   Gauss
  2 |  -1.65e-01 |  -1.79e-02 |  9.63e-06 |  6.60e-04 | -4.34e-04
  4 |  -9.15e-02 |  -4.50e-03 |  1.27e-06 |  6.36e-06 | -1.08e-07
  8 |  -4.80e-02 |  -1.13e-03 |  1.22e-07 |  4.11e-08 |  7.99e-15
 16 |  -2.46e-02 |  -2.82e-04 |  9.59e-09 |  1.28e-10 |  3.20e-14
 32 |  -1.24e-02 |  -7.05e-05 |  6.76e-10 |  1.62e-13 |  1.55e-14
 64 |  -6.24e-03 |  -1.76e-05 |  4.49e-11 | -8.88e-16 |  2.66e-14
128 |  -3.13e-03 |  -4.40e-06 |  2.90e-12 |  0.00e+00 |  3.64e-14
```

L'accélération de convergence avec la méthode de ROMBERG est impressionnante : avec seulement 64 pivots, et au prix de quelques opérations arithmétiques, la précision est passée de $3 \cdot 10^{-2}$ (dans le cas de la méthode des trapèzes) à 10^{-10}.
On remarque également que la précision de la méthode de GAUSS-LEGENDRE est excellente avec seulement 8 points de quadrature, mais c'est au prix d'une programmation plus ardue. Au-delà de 8 points, les erreurs d'arrondi dans la détermination des pivots et des points en gâtent la précision (en effet, nous avons choisi une tolérance de 10^{-14} pour la détermination numérique de ces paramètres).

6.5 Méthode adaptative

Si on cherche à calculer une valeur approchée de π à l'aide de l'égalité $\pi = 4 \cdot \int_0^1 \sqrt{1-t^2}\,dt$, on remarque que lorsque la fonction à intégrer possède une dérivée infinie en $x = 1$: dans ce cas, les méthodes classiques sont peu efficaces. Dans ce genre de situation, il faut recourir à des méthodes adaptatives, c'est-à-dire des méthodes où le pas de la subdivision augmente dans les régions où la fonction à intégrer devient irrégulière.
Parmi ces méthodes, nous allons présenter la méthode adaptative de SIMPSON [4] qui fut proposée par G.F. KUNCIR en 1962. Cette méthode utilise une estimation de l'erreur obtenue à partir d'un calcul d'intégrale par la méthode de SIMPSON. Si l'erreur est supérieure à une valeur de seuil spécifiée par l'utilisateur, l'algorithme subdivise l'intervalle d'intégration en deux et applique cette méthode récursivement à chacun des sous-intervalles. Cet algorithme s'avère généralement beaucoup plus efficace que la méthode de SIMPSON composée car elle utilise moins d'évaluations de la fonction f dans les endroits où f est suffisamment régulière pour être raisonnablement remplacée par un trinôme. La condition d'arrêt suggérée par J.N. LYNESS est :

$$\frac{1}{15}|S(a,c)+S(c,b)-S(a,b)| < \varepsilon$$

où $[a,b]$ est un intervalle de milieu c ; $S(a,b)$, $S(a,c)$, et $S(c,b)$ sont les estimations fournies par la règle de SIMPSON sur les intervalles correspondants ; enfin ε la valeur de seuil souhaitée. En utilisant le procédé d'extrapolation de RICHARDSON, il est possible de montrer que la meilleure approximation à 3 pivots de l'intégrale sur $[a,b]$ est donnée par

$$S(a,c)+S(c,b)+\frac{S(a,c)+S(c,b)-S(a,b)}{15}$$

L'estimation ainsi obtenue est exacte pour des polynômes de degré au plus cinq.
Voici une programmation possible de cet algorithme.

integration.py

```
def simpson(f, a, b):
    c = (a+b) * 0.5
    return (f(a) + 4*f(c) + f(b)) * abs(b-a) / 6

def simpson_adaptative(f, a, b, eps):
    c = (a+b) * 0.5
    reunion, gauche, droite = simpson(f,a,b), simpson(f,a,c), simpson(f,c,b)
    if abs(gauche + droite - reunion) <= 15 * eps:
        return gauche + droite + (gauche + droite - reunion)/15
    else:
        return (simpson_adaptative(f, a, c, eps/2)
                + simpson_adaptative(f, c, b, eps/2))

from math import *
s = 4 * simpson_adaptative(lambda x:sqrt(1-x**2), 0, 1, 1e-9)
print("{:20} ---> {: .15f}".format("pi", pi))
print("{:20} ---> {: .15f} => erreur = {: .2e}"
        .format("Simpson adaptative", s, pi-s))
```

```
pi                   --->  3.141592653589793
Simpson adaptative   --->  3.141592653587836 => erreur =  1.96e-12
```

4. On s'inspire ici de http://en.wikipedia.org/wiki/Adaptive_Simpson%27s_method

7 Exercices d'entraînement

Les corrigés sont disponibles sur le site dunod.com à partir de la page d'accueil de l'ouvrage.

Exercice 51. (Modification de la classe ODE pour la rendre compatible avec les systèmes)

a) Modifier deux lignes de la classe `ODE` présentée dans la section 3 (p. 196) pour permettre la vectorisation des données : la liste des points calculés doit être à présent un tableau du type `numpy.array` (cf. p. 74).

b) Pour tester cette nouvelle classe `ODE`, tracer (avec `PostScript.py` puis avec `matplotlib`) les courbes intégrales du système autonome de van der Pol en utilisant non plus la méthode `RK4_systeme` mais `RK4`.

Les exercices suivants se proposent de reprendre quelques schémas numériques présentés dans ce chapitre en utilisant cette fois-ci les primitives disponibles dans le module `SciPy`[5].

Exercice 52. (Point fixe du cosinus) En important la fonction `optimize` du module `scipy`, trouver la valeur du point fixe de la fonction cosinus, c.-à-d. la solution de l'équation $\cos x = x$.

Solution. Pour des variantes de la solution ci-dessous, consulter le fichier de corrigés en ligne.

```
>>> import numpy as np
>>> from scipy import optimize
>>> optimize.bisect(lambda x:np.cos(x)-x, 0, 1) # rép.: 0.7390851332156672
```

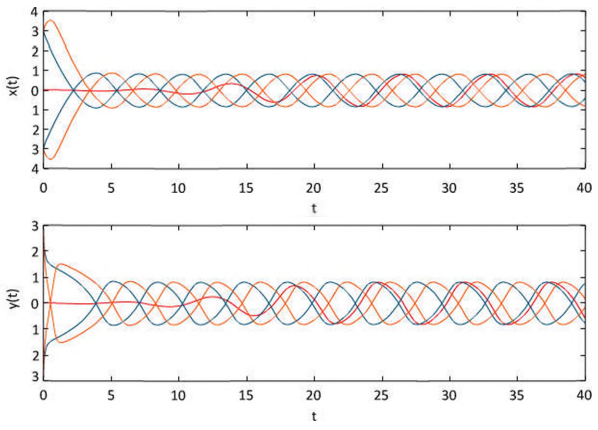
Exercice 53. (Intégration numérique) Utiliser la fonction `integrate` du module `scipy` pour tirer des formules suivantes une approximation de π :

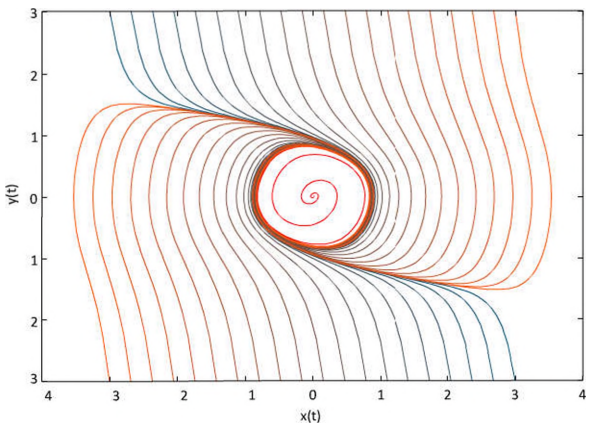
$$\pi = 4\cdot \mathrm{I} \text{ avec } \mathrm{I} = \int_0^1 \sqrt{1-t^2}\,\mathrm{d}t \qquad \text{et} \qquad \pi = 12\left(\mathrm{I} - \frac{\sqrt{3}}{8}\right) \text{ avec } \mathrm{J} = \int_0^{\frac{1}{2}} \sqrt{1-t^2}\,\mathrm{d}t$$

Comparer les résultats avec la valeur de π fournie par `scipy`.

Exercice 54. (Équations différentielles) En important la fonction `integrate` du module `scipy`, tracer les solutions des équations différentielles suivantes rencontrées à la section 3 :

a) $x'(t) = 1 + t^2 \cdot x(t)^2$ *b)* $\begin{cases} \frac{\mathrm{d}x}{\mathrm{d}t} = y \\ \frac{\mathrm{d}y}{\mathrm{d}t} = \frac{y}{2} - x - y^3 \end{cases}$ *c)* $t^2 x''(t) + t x'(t) + \left(t^2 - \alpha^2\right) x(t) = 0$





5. Pour l'installation et la documentation du module `SciPy`, consulter http://scipy.org/.

5 Récursivité

Sommaire

1 Quelques exemples

1.1 PGCD d'une liste

Exercice 55. Écrire une fonction calculant le PGCD d'une liste de nombres entiers naturels.

Solution. Le programme se décompose en deux fonctions : une fonction de calcul du PGCD de deux entiers, et une fonction de calcul du PGCD d'une liste qui s'appuie sur la fonction précédente. Le procédé est simplement récursif.

pgcdListe.py

```
def pgcd(a, b):
    """ pgcd de deux entiers naturels """
    if b == 0:
        return a
    else:
        return pgcd(b, (a % b))

def pgcd_liste(l):
    """ pgcd d'une liste """
    if len(l) == 1:
```

```
        return l[0]
    else:
        return pgcd(l[0], pgcd_liste(l[1:]))

print(pgcd_liste([4*5*6*37, 3*5*6*37, 3*4*6*37, 3*4*5*37]))
```

1.2 Compléments sur la récursivité

Lorsque la *dernière* instruction effectuée avant de quitter une procédure (ou une fonction) est un appel récursif isolé — la valeur de retour de l'appel ne pouvant constituer un argument pour un autre appel ou un calcul — à cette même procédure, on dit que la procédure est *récursive terminale.*

Comme illustration, nous calculerons récursivement (et naïvement !) de deux manières différentes la somme $\sum_{i=1}^{10} i$.

somme1.py

```
def somme(n):
    if n > 0:
        return n + somme(n-1)
    else:
        return 0

print(somme(10))
```

Dans tous les cas où n est non nul, la dernière instruction exécutée par la procédure `somme()` est une addition, dont l'un des termes est la valeur rendue par l'appel récursif de `sommme()`. Le fait de quitter le corps de la fonction en terminant par une addition induit un empilement des termes avant de réaliser l'opération, c'est-à-dire un agrandissement de la pile.

Notre programme effectue la sommation de la manière suivante :

$$\text{somme}(10) = 10 + \text{somme}(9 + \text{somme}(8 + \text{somme}(7 + \cdots)))$$

C'est-à-dire :

$$10 + (9 + (8 + (7 + \cdots)))$$

Selon les historiens, l'une des premières apparitions de la récursivité est la programmation du langage ALGOL60 par Edsger DIJKSTRA, lorsqu'il met en œuvre la notion de pile.

Si l'on songe au calcul de cette expression postfixée depuis une pile, on aura donc à évaluer la séquence :

10	9	8	...	1	0	+	+	...	+

Selon le principe consistant à dépiler deux opérandes puis empiler le résultat de l'opération, l'évaluation conduira à une hauteur maximale de pile égale à 12 dans notre exemple. On note

aussi que le premier calcul a lieu *tardivement*. Le calcul effectif de la somme ne peut commencer tant que la variable `n` n'a pas atteint la valeur 0 : l'évaluation de `somme(10)` attend l'évaluation de `somme(9)`, qui elle-même est suspendue à l'évaluation de `somme(8)`, qui... Puis les additions effectives sont réalisées en remontant.

On peut procéder différemment et faire en sorte que les calculs soient finis dès qu'on atteint la valeur 0 pour `n`. On utilise pour cela une variable auxiliaire, jouant le rôle d'un *accumulateur* : dès la fin de la récursion atteinte, celui-ci contient la somme recherchée.

somme2.py

```
def somme_rec(acc, n):
    """ somme récursive terminale (avec accumulateur) """
    if n > 0:
        return somme_rec(acc + n, n - 1)
    else:
        return acc

def somme(n):
    """ somme (fonction d'enveloppe) """
    return somme_rec(0, n)

print(somme(10))
```

Examinons alors la suite des appels pour le calcul de `somme(10)` :

$$\begin{aligned}
\text{somme}(10) &= \text{somme_rec}(0;10)\\
&= \text{somme_rec}(10;9)\\
&= \text{somme_rec}(19;8)\\
&= \text{somme_rec}(27;7)\\
&\dots\\
&= \text{somme_rec}(54;1)\\
&= \text{somme_rec}(55;0)\\
&= 55
\end{aligned}$$

Cette fois, nous avons évalué `somme(10)` de la manière suivante :

$$\sum_{i=1}^{10} i = ((((((((((10+9)+8)+7)+\cdots)+1$$

Reprenons notre expression postfixée équivalente pour son évaluation sur une pile : le calcul nécessitera cette fois une pile dont la hauteur ne dépassera pas 2 !

10	9	+	8	+	7	+	...	1	+	0

Le second programme montre une version récursive terminale de notre fonction de calcul de la somme. L'utilisation des ressources est bien moins importante (dans notre exemple, la taille maximale de la pile nécessaire au calcul est atteinte dès la première opération, qui survient d'ailleurs sans être différée).

Exercice 56. Écrire une fonction qui élève un nombre à une puissance donnée, en utilisant une récursivité terminale.

Solution. L'utilisation d'une fonction « enveloppe » exonère l'utilisateur des détails de programmation. Pour lui, savoir qu'on utilise un accumulateur ne présente aucun intérêt. On a inclus un affichage pour observer l'évolution des calculs.

puissRecTerm.py

```
def puissance_rec_term(a, n, p):
    print "appel avec a =", a, "; n =", n, "; p =", p
    if n == 0:
        return p
    else:
        return puissance_rec_term(a, n-1, p*a)

def puissance(a, n):
    return puissance_rec_term(a, n, 1)

print(puissance(2, 10))
```

Exercice 57. Donner une version récursive et une version récursive terminale d'une procédure de calculs des termes de la suite de FIBONACCI.

Solution. On donne d'abord la version récursive classique.

fibRec.py

```
def fibo(n):
    if n < 2:
        return 1
    else:
        return fibo(n - 1) + fibo(n -2)

for n in range(12):
    print(fibo(n))
print(fibo(40)) # donne 165580141
```

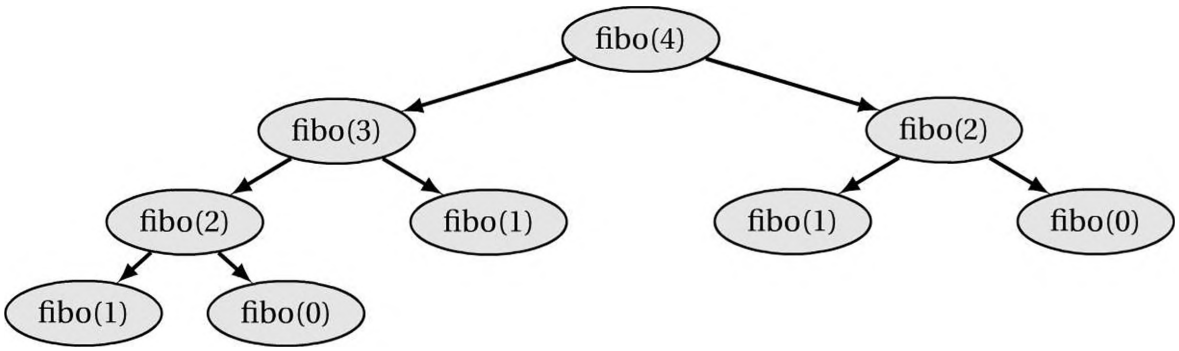
La version récursive terminale suit :

fibRecTerm.py

```
def fibo_rec_term(n, i, a, b):
    if i == n:
        return b
    else:
        return fibo_rec_term(n, i+1, a+b, a)

def fibo(n):
    """ fonction d'enveloppe """
    return fibo_rec_term(n, 0, 1, 1)

for n in range(12):
    print(fibo(n))
print(fibo(40))
```

La solution récursive simple sature rapidement les ressources de la machine, qui exige un temps non négligeable (de l'ordre 3 minutes sur un ordinateur avec processeur 32 bits d'1 Ghz sous Linux) avant de délivrer une réponse pour `fibo(40)`. En revanche, le second programme donne presque immédiatement la réponse.

Revenons un instant sur la version récursive non terminale présentée ci-dessus. Si le programme est conforme au plus près à la définition mathématique de cette suite, il n'est pas efficace en pratique. On peut s'en convaincre en représentant sous forme d'arbre binaire les calculs nécessaires à l'obtention de `fibo(4)` :



Hormis les feuilles (toujours égales à 1 d'après la définition mathématique de la suite de FIBONACCI), le calcul de chaque nœud demande l'addition de deux sous-arbres. On note immédiatement que certains sous-arbres identiques apparaissent rapidement à plusieurs reprises. De tels sous-arbres vont grossir avec l'augmentation du rang du terme à calculer ; le gaspillage de ressources qui en résulte est considérable.

Ces exercices sont l'occasion d'évoquer l'efficacité et la complexité d'un algorithme. L'époque n'est pas si éloignée, où les ordinateurs étaient largement moins véloces, avec des unités de calcul opérant sur des données de plus petites tailles, et dotés d'une mémoire de travail très restreinte (seulement quelques mega-octets de mémoire vive et des processeurs dont la vitesse était mesurée en MHz).

1.2.1 Dérécursification

Dans certains cas, on peut dérécursifier un programme. Prenons l'exemple d'un programme énumérant la liste des entiers de 0 à 9 :

```
def boucle(n):
    if n > 0:
        boucle(n - 1)
    print(n)

boucle(9)
```

```
i = 0
while i < 10:
    print(i)
    i = i + 1
```

Les deux programmes produisent la liste attendue. Le second programme est itératif, ce qui nous contraint à l'emploi des affectations de variables.
Nous avons substitué une boucle itérative à la définition récursive.

Exercice 58. Donner la version itérative d'une procédure de calcul des termes de la suite de FIBONACCI.

Solution. Le premier programme est la version générique, applicable à tout langage de programmation ; la version itérative à la mode « **Python** » met en œuvre l'affectation parallèle, ce qui économise l'emploi d'une variable locale.

figIterGen.py

```
def fibo(n):
    a = 1
    b = 1
    for i in range(n):
        t = a
        a += b
        b = t
    return b

for n in range(12):
    print(fibo(n))
```

fibIterPy.py

```
def fibo(n):
    a = b = 1
    for i in range(n):
        a, b = a + b, a
    return b

for n in range(12):
    print(fibo(n))
```

1.3 Complexité

1.3.1 Motivations : ce qui est attendu d'un algorithme

En algorithmique comme en programmation, on aborde les questions suivantes afin d'apprécier la validité et la qualité d'un algorithme ou d'un programme.

- preuve de terminaison (l'exécution s'achève et l'algorithme se termine en un nombre fini d'étapes) ;
- efficience (l'algorithme réalise bien ce pourquoi il est conçu) ;
- spécification complète : de même que les hypothèses d'un énoncé ou le domaine de définition d'une fonction sont bien établis, les étapes et instructions qui composent l'algorithme sont toutes parfaitement déterminées, par leur nature et par les données sur lesquelles elles s'appliquent.

1.3.2 Que mesure-t-on ?

Mesurer la complexité et l'efficacité d'un programme exige de se doter de protocoles et outils permettant d'établir des comparaisons, entre algorithmes d'une part, et entre différentes programmations d'un même algorithme d'autre part.

1.3.3 Mesures expérimentales sur un programme réel

On peut effectuer des mesures lors du déroulement d'un calcul sur une machine. Cela n'est pas toujours possible. Cette approche n'est pas souhaitable pour des programmes critiques, comme ceux qui gouvernent un calculateur embarqué pour la gestion d'un vol aéronautique, ou le contrôle des équipements d'une centrale nucléaire.

Ces mesures, lorsqu'elles sont possibles, sont relatives à l'environnement matériel d'exécution du programme : architecture de l'ordinateur hôte, performances des processeurs, disponibilité de la mémoire en quantité suffisante...

Empreinte en mémoire. On étudie ici l'empreinte mémoire du programme pendant sa durée de vie : « comment la consommation de mémoire évolue-t-elle entre le début et la fin de l'exécution du programme ? » est une question à examiner si l'on souhaite éviter de saturer un calculateur et provoquer un arrêt brutal.

Coût en temps de calcul. Il s'agit dans ce cas d'étudier le temps nécessaire à l'exécution d'un programme. Évaluer la complexité en temps d'un algorithme est une prévision théorique, que l'on peut ensuite décliner pour un programme et une architecture. Afin d'éliminer la question du coût d'une instruction qui est variable selon l'environnement logiciel, on pourra évaluer la complexité en termes d'instructions élémentaires.

Complexité de conception. Le programmeur est conduit à choisir les algorithmes et structures de données de la façon la plus pertinente pour l'algorithme qu'il doit mettre en œuvre sous forme de programme. De fait, le seul outil dont dispose le programmeur est son propre raisonnement.

1.3.4 Avec quels outils ?

Les calculs sur ordinateur sont souvent l'occasion des « nombres réels machine », dont la maîtrise des arrondis dans les calculs approchés doit être envisagée d'emblée avec précaution. Néanmoins, nous nous limiterons ici à l'évocation de la complexité temporelle que nous noterons $t(n)$.

$t(n)$ est une fonction de $\mathbb{N}$ dans $\mathbb{R}^+$ pour laquelle nous devons préciser la nature de n :

- dans le cas du tri d'une liste, n peut désigner la longueur de la liste ;
- dans le cas de la factorisation d'un entier, n pourra être la valeur cet entier ;
- n peut désigner le nombre d'instructions élémentaires...

1.3.5 Un exemple : coût en temps d'une fonction itérative

Nous reprendrons l'exemple de la somme $\sum_{i=1}^{10} i$.

somme.py

```
def somme(n):          # (0)
    s = 0              # (1)
    i = 1              # (2)
    while i <= n:      # (3)
        s = s + i      # (4)
        i = i + 1      # (5)
    return s           # (6)

print(somme(10))       # (7)
```

Chacune des instructions (i) possède un coût d'exécution en temps que nous noterons c_i $(c_i \in \mathbb{R}^+)$:

- $c_1 = c_2$ si on suppose que toutes les affectations élémentaires ont la même durée ;
- le coût c_3 se décompose en un coût fixe pour le test, puis selon le cas en un coût de branchement après la fin de la boucle, ou en n fois le coût du corps de la boucle ;
- $c_4 = c_5 = c_+ + c_1$ sont les coûts d'instructions composées d'une addition et d'une affectation, et c_+ désigne le coût d'une instruction d'addition entre deux nombres ;
- c_6 est le coût du retour de fonction ;
- c_7 est le coût composé du coût d'appel de fonction (empilement des paramètres, branchement) et du coût d'affichage de la valeur calculée (dépilement).

Selon que l'on considère que certaines opérations sont élémentaires ou non, on influe sur la granularité et l'estimation du coût.
Nous pouvons maintenant effectuer notre calcul du coût d'exécution $\mathrm{C}(n)$:

$$\begin{aligned} \mathrm{C}(n) &= c_1 + c_2 + c_3 + n \times (c_4 + c_5) + c_6 + c_7 \\ &= 2c_1 + c_3 + 2n\,(c_1 + c_+) + c_6 + c_7 \\ &= (2\,(c_1 + c_+))\,n + (2c_1 + c_3 + c_6 + c_7) \\ &= \mathrm{K}_1 n + \mathrm{K}_2 \end{aligned}$$

Nous constatons un coût affine en n. Selon la notation de LANDAU, nous avons : $\mathrm{C}(n) = \mathrm{O}(n)$. Cette égalité exprime (nous sommes en présence de fonctions positives) :

$$\exists \mathrm{K} \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geqslant n_0, \mathrm{C}(n) \leqslant \mathrm{K} \times n.$$

Le coût d'exécution de notre fonction est majoré par une fonction affine de n, elle-même majorée par une fonction linéaire. Le programmeur, par l'étude mathématique du problème, et l'existence d'une formule élégante y répondant, gagnera énormément à remplacer le calcul de la somme $\sum_{i=1}^{n} i$ par celui de

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

car le programme correspondant est lui en $\mathrm{O}(1)$, c'est-à-dire en temps constant, quelle que soit la valeur de n.

1.4 Complexité des algorithmes de tri

Dans ce paragraphe, on étudie la complexité des algorithmes de tri présentés dans la section 13 en fin du premier chapitre.

a) Complexité du tri par sélection :
La relative simplicité de cet algorithme fait qu'on lui attribue le qualificatif d'« algorithme naïf ». Cela signifie que même si l'algorithme est correct, il est trop naïf pour être réellement efficace.
À la première itération, sont effectuées $(n-1)$ comparaisons. À la p-ième itération, sont effectuées $(n-p)$ comparaisons. Soit une complexité en $\mathrm{O}(n^2)$ dont on peut obtenir une représentation en donnant un coût unitaire aux opérations atomiques :

$$\sum_{p=1}^{n-1} (n-p) = \frac{1}{2}(n^2 - n).$$

b) *Complexité du tri par insertion :*
Soit un tableau de n éléments. Pour mettre en œuvre le tri par insertion, il faut insérer un à un $(n-1)$ éléments dans un tableau préalablement trié. Dans le pire des cas, c'est-à-dire quand la liste à trier de manière croissante est classée de manière décroissante, pour insérer le p^{e} élément, il faut décaler les $(p-1)$ éléments qui le précédent. Soit un coût total, dans le pire des cas, donnée par la formule suivante :

$$\sum_{p=2}^{n}(p-1)=\frac{1}{2}(n^2-n).$$

Cependant, en moyenne, seule la moitié des éléments est décalée. Soit un décalage de $(p-1)/2$ éléments et donc une complexité en moyenne qui se calcule ainsi :

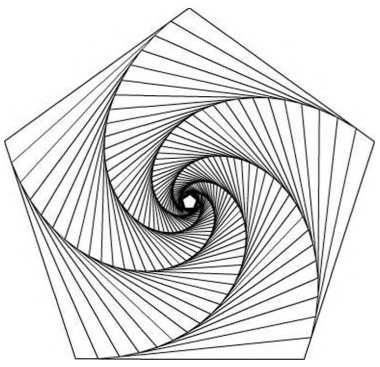
$$\sum_{p=2}^{n}\frac{1}{2}(p-1)=\frac{1}{4}(n^2-n).$$

Il s'agit d'une complexité en $O(n^2)$ analogue à la complexité du tri par sélection. Cependant, le tri par insertion présente un avantage certain : il peut être facilement mis en œuvre pour insérer de nouvelles données dans un tableau déjà trié ou pour trier des valeurs au fur et à mesure de leur apparition (cas des algorithmes en temps réel où il faut parfois exploiter une série de valeurs triées qui vient s'enrichir, au fil du temps, de nouvelles valeurs). C'est l'un des seuls algorithmes de tri dont la complexité est meilleure quand le tableau initial possède un « certain ordre ».

c) *Complexité du tri rapide :*
La complexité du tri rapide pour trier une liste de n éléments est égale à la complexité pour le tri de p et de q éléments où $p+q+1=n$. Soit $C(n)=C(p)+C(q)+constante$. Dans le meilleur des cas, $p=q=n/2$, soit $C(n)=2C(q)$. On peut alors montrer que la complexité en moyenne est en $O(n\cdot\ln(n))$.

2 Spirale de pentagones

Exercice 59. La spirale régulière ci-dessous est constituée de 50 pentagones (la longueur du plus grand côté étant 370 pixels), emboîtés de sorte qu'entre deux pentagones consécutifs inscrits l'un dans l'autre, le côté du plus petit est une réduction de 10 % du côté du plus grand. Programmer le dessin de cette spirale.



Solution.

pentagones.py

```
from math import *
from turtle import *

def pentagone(cote):
    for n in range(5):
        forward(cote)
        left(72)

def spirale(nombre_triangles, cote):
    coeff = 0.1
    a = cote
    d = coeff * a
    for i in range(nombre_triangles):
        pentagone(a)
        d = coeff * a
        forward(d)
        a_prime = sqrt((a-d)**2 + d**2 -2*(a-d)*d*(1-sqrt(5))/4)
        angle = (180/pi) * acos((a_prime**2 + a**2 - 2*a*d)/(2*a_prime*(a-d)))
        left(angle)
        a = a_prime

up()
goto(-190,-210)
down()
spirale(50, 370)
up()
```

3 Courbe du dragon

La courbe du dragon est une autre application de la récursivité.

dragon1.py

```
from math import sqrt
from turtle import *

def dragon_gauche(taille, niveau):
    color("red")
    if niveau == 0:
        forward(taille)
    else:
        left(45)
        dragon_gauche(taille/sqrt(2), niveau-1)
        right(90)
        dragon_droite(taille/sqrt(2), niveau-1)
        left(45)
```

```
def dragon_droite(taille, niveau):
    color("blue")
    if niveau == 0:
        forward(taille)
    else:
        right(45)
        dragon_gauche(taille/sqrt(2), niveau-1)
        left(90)
        dragon_droite(taille/sqrt(2), niveau-1)
        right(45)

dragon_droite(400, 12)
```

La courbe du dragon pave le plan, comme le montre la figure suivante :

dragon2.py

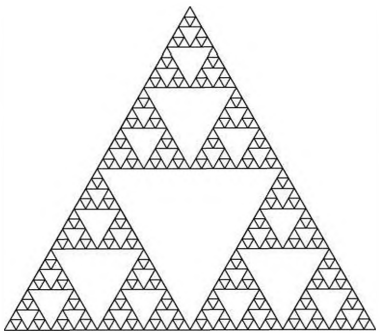
```
from math import sqrt
from turtle import *

def dragon_gauche(taille, niveau):
    if niveau == 0:
        forward(taille)
    else:
        left(45)
        dragon_gauche(taille/sqrt(2), niveau-1)
        right(90)
        dragon_droite(taille/sqrt(2), niveau-1)
        left(45)

def dragon_droite(taille, niveau):
    if niveau == 0:
        forward(taille)
    else:
        right(45)
        dragon_gauche(taille/sqrt(2), niveau-1)
        left(90)
        dragon_droite(taille/sqrt(2), niveau-1)
```

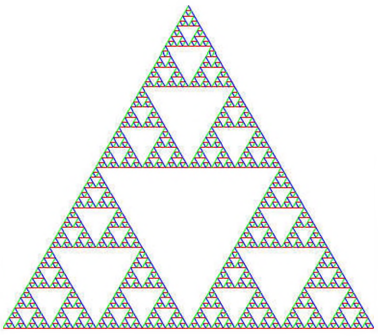
```
        right(45)

def figure():
    color("red")
    dragon_gauche(200, 10)
    up()
    goto(0, 0)
    right(90)
    down()
    color("blue")
    dragon_gauche(200, 10)
    up()
    goto(0, 0)
    right(90)
    down()
    color("green")
    dragon_gauche(200, 10)
    up()
    goto(0, 0)
    right(90)
    down()
    color("gray")
    dragon_gauche(200, 10)
    up()

figure()
```

4 Triangle de Sierpińsky

4.1 Construction récursive

Les deux dessins ci-dessous sont obtenus respectivement par les programmes `sierpinsky1.py` et `sierpinsky2.py` :





sierpinsky1.py

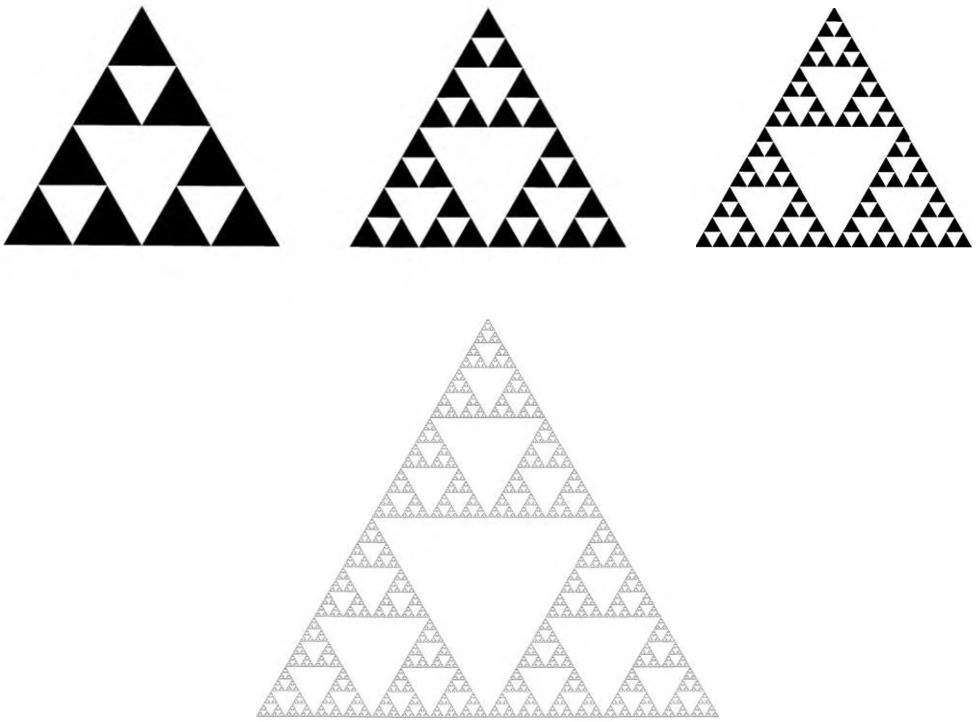
```
from turtle import *
```

```
def drapeau(longueur, niveau):
    if niveau == 0:
        forward(longueur)
    else:
        drapeau(longueur/2, niveau-1)
        left(120)
        drapeau(longueur/2, niveau-1)
        left(120)
        drapeau(longueur/2, niveau-1)
        left(120)
        forward(longueur)

def triangle(longueur, niveau):
    for i in range(3):
        drapeau(longueur, niveau)
        left(120)

triangle(600, 5)
```

Avec un peu de couleur :

sierpinsky2.py

```
from turtle import *

def drapeau(longueur, niveau):
    if niveau == 0:
        down()
        cap = heading()
        if cap == 0 or cap == 180:
            color("red")
        if cap == 60 or cap == 240:
            color("green")
        if cap == 120 or cap == 300:
            color("blue")
        forward(longueur)
        up()
    else:
        drapeau(longueur/2, niveau-1)
        left(120)
        drapeau(longueur/2, niveau-1)
        left(120)
        drapeau(longueur/2, niveau-1)
        left(120)
        forward(longueur)

def triangle(longueur, niveau):
    for i in range(3):
        drapeau(longueur, niveau)
        left(120)

triangle(600, 6)
```



4.2 Construction aléatoire

On peut obtenir aléatoirement le triangle de SIERPIŃSKI par un algorithme très simple :

> ABC est un triangle, M est un point quelconque.
> *a*) Choisir au hasard l'un des trois points A, B et C.
> *b*) Construire le point M_1 milieu du segment liant le point précédent au point M.
> *c*) Recommencer à la première étape en faisant jouer au point M_1 le rôle du point M.

Bien évidemment, il faut poursuivre l'algorithme assez longtemps sur le papier pour observer quelque chose, sans compter les segments $[M_iM_{i+1}]$ uniquement utiles à la construction mais qui encombrent le dessin...

Cet algorithme donne la figure suivante, réalisée avec le programme `sierpinsky3.py`. Ce programme fait appel à un nouveau type objet défini par le programmeur pour représenter un objet « point du plan », repéré par ses coordonnées cartésiennes et doté d'une couleur.

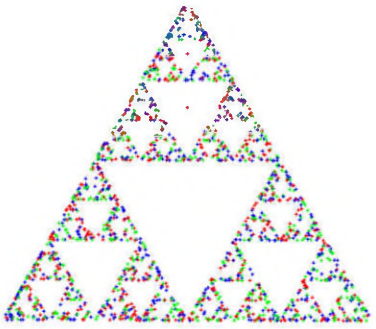
sierpinsky3.py

```
from math import sqrt
import random
from turtle import *

delta = -300
loupe = 600

class point(object):
```

```
    """ Point du plan """

    def __init__(self, x = 0, y = 0, couleur = "black"):
        self.x = x
        self.y = y
        self.couleur = couleur

    def milieu(self, p, couleur = "black"):
        self.x = (self.x + p.x)/2
        self.y = (self.y + p.y)/2
        self.couleur = couleur

    def rouge(self):
        self.couleur = "red"

    def vert(self):
        self.couleur = "green"

    def bleu(self):
        self.couleur = "blue"

    def croix(self):
        down()
        for i in range(4):
            forward(3)
            backward(3)
            left(90)
        up()

    def imprime(self):
        goto(self.x*loupe + delta, self.y*loupe + delta)
        color(self.couleur)
        self.croix()

def figure(points):
    a = point(0.0, 0.0, "red")
    b = point(1.0, 0.0, "green")
    c = point(0.5, sqrt(3)/2, "blue")

    # premier point m, i.e. m(0) choisi ici comme isobarycentre
    m = point(0.5, sqrt(3)/6)

    if points <= 0:
        points = 200

    for i in range(points):
        # obtention aléatoire du point suivant:
        # m(i+1) = milieu [m(i), a ou b ou c]
        alea = random.randrange(3)
        if alea == 0:
```

```
            m.milieu(a)
        if alea == 1:
            m.milieu(b)
        if alea == 2:
            m.milieu(c)
        # obtention aléatoire de la couleur du point
        alea = random.randrange(3)
        if alea == 0:
            m.rouge()
        if alea == 1:
            m.vert()
        if alea == 2:
            m.bleu()
        m.imprime()

up()
speed("fastest")
figure(2000)
```

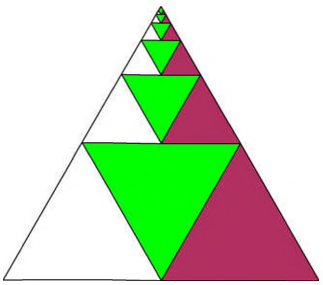
Le programme donne une figure comme la suivante :



5 Sommes de termes d'une suite géométrique

On laisse en exercice l'écriture d'un programme pour l'obtention des sommes partielles de la série $\sum_{i=1}^{+\infty} \left(\frac{1}{4}\right)^i$.

La tortue **LOGO** du module standard « `turtle` » permet de donner aisément une interprétation géométrique de la réponse :



```
from turtle import *

def triangle_vert(c):
    color("green")
    begin_fill()
    for i in range(3):
        forward(c)
        left(120)
    end_fill()
    color("black")
    for i in range(3):
        forward(c)
        left(120)

def triangle_blanc(c):
    color("white")
    begin_fill()
    for i in range(3):
        forward(c)
        left(120)
    end_fill()
    color("black")
    for i in range(3):
        forward(c)
        left(120)

def triangle_marron(c):
    color("maroon")
    begin_fill()
    for i in range(3):
        forward(c)
        left(120)
    end_fill()
    color("black")
    for i in range(3):
        forward(c)
        left(120)

def complet(n, c):
    if n > 0:
        triangle_vert(c)
        triangle_blanc(c/2)
        forward(c/2)
        triangle_marron(c/2)
        backward(c/2)
        left(60)
        forward(c/2)
        right(60)
        complet(n-1, c/2)

up()
goto(-300,-240)
down()
complet(6, 512)

nomFichier = "somme-geom"
ht()
ts = getscreen()
ts.getcanvas().postscript(file =
    "{0}.eps".format(nomFichier))
import os
os.system("epstopdf
    {0}.eps".format(nomFichier))
mainloop()
```

6 Exercices d'entraînement

Les corrigés sont disponibles sur le site dunod.com à partir de la page d'accueil de l'ouvrage.

Pour cette série d'exercices, l'objectif est de concevoir des programmes récursifs faisant appel à la notion de boucle d'itération, sans pour autant utiliser les constructions `for` et `while` définies dans le langage **Python**.

Exercice 60.

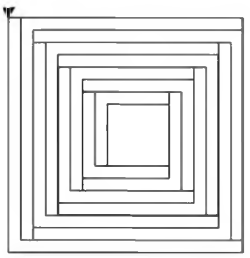
a) Écrire un programme produisant une table de multiplication pour un entier donné. L'utilisateur choisira en entrée l'entier et le nombre de multiples.

b) Proposer une version du programme précédent donnant la même table en ordre inverse (décroissant).

c) Écrire un programme de calcul de l'arrangement $A_n^k = n(n-1)(n-2)\ldots(n-k+1)$, où n et k sont deux entiers naturels vérifiant : $k \leqslant n$.
Pour cela, construire récursivement une boucle itérant un entier naturel.

Exercice 61.
Concevoir deux fonctions mutuellement récursives `pair()` et `impair()` qui indiquent si leur paramètre en appel est un nombre pair ou impair.

Exercice 62.
En employant le module `turtle` de **Python**, écrire un programme réalisant la figure ci-contre, dans laquelle le carré intérieur possède un côté de 50 pixels, et les gnomons sont rectangles isocèles, d'une « épaisseur » de 10 pixels :



Exercice 63.
x désigne un nombre réel.

Écrire, dans une forme récursive et sans recourir aux routines standard du langage, une fonction qui calcule une approximation de $cos(x)$, x étant exprimé en radians. On utilisera le développement limité à l'ordre 6 de la fonction cosinus au voisinage de 0.

On pourra observer à titre de comparaison la sortie avec celle de la fonction mathématique cos de **Python**, disponible dans le module standard `math`.

Exercice 64.
On utilisera *uniquement* le type entier (« int ») de **Python** pour cet exercice. Il s'agit d'écrire une fonction qui rend la partie entière de la racine carrée d'un nombre entier naturel.

Exercice 65. (Fractions continues – 1re partie)

Pour le nombre rationnel $\frac{111}{40}$, on peut donner un développement sous forme de fraction continue (tous les numérateurs sont égaux à 1) :

$$\frac{111}{40} = 2 + \cfrac{1}{1 + \cfrac{1}{3 + \cfrac{1}{2 + \cfrac{1}{4}}}}$$

que l'on écrira sous la forme condensée : $[2,1,3,2,4]$.

Écrire un programme qui donne la représentation en fraction continue d'un nombre rationnel, en forme condensée. On envisagera le cas des nombres négatifs.

On complètera cette étude avec le calcul des réduites, dans un exercice du chapitre 6.

Exercice 66.
On se propose d'écrire quelques outils pour manipuler le type `list` standard de **Python** :

- une fonction « `tete()` » : elle retourne le premier élément de la liste en argument ;
- une fonction « `suite()` » : elle retourne une nouvelle liste, obtenue par suppression de la tête de la liste en argument. La liste passée en argument doit rester intègre après appel de la fonction ;
- une fonction « `coupe()` » : elle retourne la liste qui résulte de la suppression des n premiers éléments, lorsque la suppression est possible ;
- une fonction « `debut()` » : elle retourne la liste des n premiers éléments.

On utilisera les entêtes suivants, le paramètre *liste* désignant une liste native de **Python** :

```
def tete(liste):
    pass # code à écrire ici

def suite(liste):
    pass

def coupe(n, liste):
    pass

def debut(n, liste):
    pass
```

Quelques exemples d'emploi de ces primitives en mode interactif :

```
>>> import random
>>> from outils import *
>>> hasard = [random.randrange(50) for t in range(20)]
>>> hasard
[48, 9, 16, 8, 38, 46, 27, 45, 11, 44, 47, 3, 47, 10, 12, 10, 32, 18, 13, 41]
>>> recherche(tete(hasard), hasard)
True
>>> tete(hasard)
48
>>> suite(hasard)
[9, 16, 8, 38, 46, 27, 45, 11, 44, 47, 3, 47, 10, 12, 10, 32, 18, 13, 41]
>>> debut(2, hasard)
[48, 9]
>>> coupe(2, hasard)
[16, 8, 38, 46, 27, 45, 11, 44, 47, 3, 47, 10, 12, 10, 32, 18, 13, 41]
>>>
```

6 Classes

Sommaire

La programmation est souvent l'occasion de manipuler des informations, dont l'intérêt réside dans le regroupement sous forme d'une collection ; le nombre d'éléments d'une telle collection peut ne pas être connu avant l'utilisation du programme, ou du moins peut être amené à varier au cours de l'exécution du programme. Les objets d'une telle collection peuvent être de nature hétérogène ou non, comme ils peuvent être organisés ou non, selon des critères pertinents définis par le programmeur pour les besoins du problème à résoudre.

Nous traiterons dans ce chapitre de données organisées sous forme séquentielle de listes, ou de données organisées sous forme d'arbres binaires. En première approche, nous distinguerons deux usages différents qui constitueront un critère de choix pour l'une ou l'autre structure : lorsque que l'insertion ou la suppression d'un élément s'effectuent toujours en même position dans la liste, une structure de liste (pile ou file) est souvent appropriée ; en revanche, si un besoin de conservation de l'ordre des éléments est essentiel, après insertion ou suppression, une structure d'arbre binaire devra être préférée.

Nous rappelons, comme nous avons déjà pu le rencontrer lors de chapitres précédents, que langage **Python** fournit nativement les outils et structures de données nécessaires à l'utilisation des listes. Toutefois, la présentation des listes constitue une part importante de l'algorithmique, et il est bon de connaître le fonctionnement interne de ce type classique de structure de données. Enfin, l'étude des listes permettra de mieux appréhender ensuite la notion d'arbre binaire qui, elle, ne fait pas partie du noyau du langage.

Ce chapitre prend place dans un exposé plus vaste sur la notion de classe, inhérente à la programmation orientée objet. En effet, l'implémentation de liste sous forme de classes d'objets trouve sa nécessité dans le souci de fournir au programmeur une structure de données dont l'abstraction lui permet de concentrer son effort sur le problème à résoudre, tout lui en offrant une séparation des concepts et un bon confort d'utilisation et de maintenance.

Lors de la conception de nos classes, nous aurons l'occasion d'examiner et de mettre en œuvre les notions de composition et d'héritage.

Nous prendrons comme fil directeur de cette présentation la réalisation d'un petit système de calcul pour les nombres rationnels d'une part, la mise en œuvre d'un anneau de polynômes à coefficients dans $\mathbb{Z}$ d'autre part.

Notre étude débouchera sur la conception d'un système de calcul manipulant un corps de fractions rationnelles.

Avant d'entrer dans le vif du sujet, nous mentionnons toutefois quelques avertissements :

- Tous les programmes de ce chapitre ont été écrits et testés avec une distribution de la branche 2.6 de **Python**. Ils sont compatibles avec les versions suivantes 3.x mais ils ne tirent nullement avantage des nouvelles fonctionnalités du langage. En particulier, le lecteur notera la conservation de l'instruction « `print` » en lieu et place de la nouvelle syntaxe « `print()` ». Il y a ici une subtilité pour l'interpréteur **Python** qui, en versions antérieures, considère les parenthèses selon le cas comme le signe de la présence d'un *tuple*.
- Enfin, dans un but d'étude, nous n'avons pas cherché à utiliser toutes les potentialités du langage qui en font son originalité. Ce choix est motivé par la volonté de ne pas

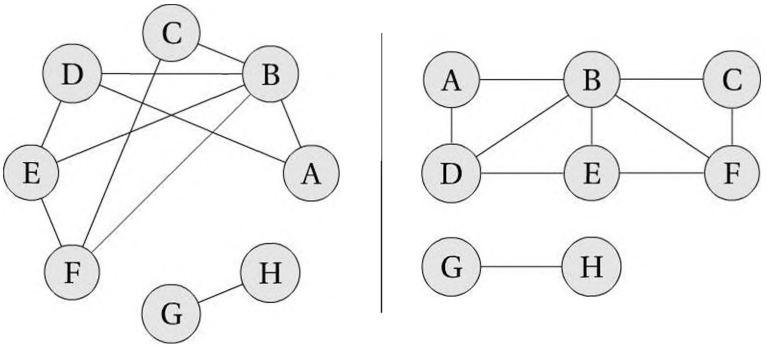


FIGURE 6.1 – Deux représentations du même graphe planaire (non connexe)

masquer des algorithmes ou détails — par l'utilisation de primitives puissantes mais de fait « opaques » — au lecteur qui souhaiterait transférer les concepts présentés vers un autre langage de programmation.

1 Graphes

Avant de nous pencher dans les sections suivantes sur les notions d'arbre et de liste et quelques-unes de leurs applications, nous commençons ce chapitre en abordant la notion de graphe, qui permet notamment une généralisation des deux notions précédentes, au sens où un arbre est un graphe connexe sans cycle.

Si la théorie des graphes constitue un domaine récent au regard de l'histoire des mathématiques, il n'en demeure pas moins vaste et source de nombreux travaux contemporains. Nous limiterons dès lors notre propos à l'étude de questions relatives aux graphes simples, c'est-à-dire sans arête multiple ni boucle. Nous choisissons aussi de resteindre notre étude aux graphes non orientés aux arêtes non pondérées.

1.1 Présentation

Deux types de représentation sont couramment utilisés pour un graphe : une matrice d'adjacence (figure 6.2) et une liste d'adjacence. Une matrice d'adjacence est une matrice carrée symétrique (pour un graphe non orienté), dont les éléments (dans notre cas équivalents à des booléens) indiquent l'existence ou non d'une arête entre deux sommets.

Le programme ci-après illustre une représentation par liste d'adjacence du graphe 6.1 :

lettres.py

```
from graphe import *

g = graphe(
    {
        'A': ['B', 'D'],
        'B': ['A', 'C', 'D', 'E', 'F'],
        'C': ['B', 'F'],
        'D': ['A', 'B', 'E'],
        'E': ['B', 'D', 'F'],
```

```
        'F': ['B', 'C', 'E'],
        'G': ['H'],
        'H': ['G']
    }
)

print("Chaînes(s) élémentaire(s) entre A et F:")
for c in g.recherche_chaines("A", "F"):
    print(c)
print()

print("Chaînes(s) élémentaire(s) entre A et G:")
for c in g.recherche_chaines("A", "G"):
    print(c)
```

Le module `graphe.py` qui fournit le type « graphe » n'est pas reproduit ici dans son intégralité, mais nous pouvons observer son résultat à l'exécution pour le graphe donné en 6.1 :

Terminal

```
$ ./lettres.py
Chaînes(s) élémentaire(s) entre A et F:
['A', 'B', 'C', 'F']
['A', 'B', 'D', 'E', 'F']
['A', 'B', 'E', 'F']
['A', 'B', 'F']
['A', 'D', 'B', 'C', 'F']
['A', 'D', 'B', 'E', 'F']
['A', 'D', 'B', 'F']
['A', 'D', 'E', 'B', 'C', 'F']
['A', 'D', 'E', 'B', 'F']
['A', 'D', 'E', 'F']

Chaînes(s) élémentaire(s) entre A et G:

$
```

Nous programmons notre structure de graphe comme un objet (au sens de la programmation) dont l'unique donnée interne est une liste d'adjacence : une telle liste est constituée de l'ensemble des sommets du graphe, et pour chaque sommet adjacent, la liste des sommets qui lui sont connectés. Nous écartons la représentation par une matrice d'adjacence pour une raison d'économie de moyens (pour les graphes non complets, les coefficients de la matrice d'adjacence sont souvent en majorité nuls) ; le passage d'un modèle à l'autre constitue cependant un excellent exercice de programmation confié au lecteur.

1.2 Quelques questions usuelles

Nous nous proposons par exemple d'aborder — du point de vue de la programmation — les questions suivantes :

- Un graphe est-il connexe ?

$$\begin{array}{cccccccc} A & B & C & D & E & F & G & H \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \left(\begin{array}{cccccccc} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}\right) & \begin{array}{l} \leftarrow A \\ \leftarrow B \\ \leftarrow C \\ \leftarrow D \\ \leftarrow E \\ \leftarrow F \\ \leftarrow G \\ \leftarrow H \end{array} \end{array}$$

FIGURE 6.2 – Matrice d'adjacence du graphe 6.1

- Quelles sont les composantes connexes d'un graphe ?
- Comment établir les chaînes élémentaires entre deux sommets d'un graphe ?
- Quelles sont les chaînes les plus courtes entre deux sommets d'un graphe ?

Nous avons ainsi constitué, dans le but de disposer d'outils capables de manipuler plusieurs graphes indépendamment les uns des autres au sein d'un même programme, une boîte à outils organisée sous la forme d'une classe d'objet. Nous allons en détailler les principales méthodes :

`composante_connexe()` Cette méthode permet de déterminer la composante connexe d'un sommet du graphe. Elle est notamment utilisée pour l'obtention de la liste des composantes du graphe, ainsi que pour tester l'existence d'une chaîne entre deux sommets.

`recherche_chaines()` Cette méthode établit la liste de toutes les chaînes entre deux sommets donnés.

`recherche_cycles()` Cette méthode s'appuye sur la méthode précédente, dans la mesure où un cycle est une chaîne dont les deux extrémités sont le même sommet.

`nombre_aretes()` Cette méthode, basée sur le lemme des poignées de mains, peut sembler superflue. Elle conserve son intérêt si l'on se souvient que le constructeur d'un objet `graphe` ne mentionne que la liste d'adjacence, dont la méthode « déduit » le nombre d'arêtes.

Nous n'en dirons pas plus en ce qui concerne quelques méthodes secondaires, néanmoins fort utiles, ayant trait par exemple à la détermination du nombre de sommets et d'arêtes d'un graphe... dont le lecteur pourra prendre connaissance à la lecture des fichiers disponibles en téléchargement.

Les méthodes évoquées ci-dessus peuvent être à leur tour aisément invoquées pour construire des méthodes fournissant les réponses aux questions posées plus haut.

1.3 Parcours eulériens

Rappelons qu'une chaîne ou un cycle eulériens désignent respectivement une chaîne ou un cycle qui empruntent exactement une fois chacun toutes les arêtes d'un graphe connexe.

L'existence éventuelle de telles chaînes ou cycles est assurée par le théorème d'EULER – HIERHOLZER. Un procédé de construction de parcours eulériens est celui connu sous le nom d'« algorithme de FLEURY » sur lequel nous avons basé notre méthode qui génère un parcours eulérien, chaîne ou cycle suivant le cas.

Nous disposons dans notre objet `graphe` de deux méthodes « `existence_chaine_Euler()` » et « `existence_cycle_Euler()` » pour étudier la question de l'existence. Dans l'affirmative, on utilise alors la méthode « `recherche_Euler()` » pour la génération d'une solution au problème.

graphe.py

```
    # ... le début de la classe n'est pas reproduit ici ...

    def graphe_reduit(self, a, b):
        """ construit un graphe en supprimant [ab] """
        h = dict()
        for s in self.sommets():
            l = []
            for t in self.adjacents(s):
                if (s != a or t != b) and (s != b or t != a):
                    l.append(t)
            if len(l) != 0:
                h[s] = l
        return graphe(h)
```

La méthode « `graphe_reduit()` » construit à partir de l'objet graphe lui-même un nouvel objet graphe, obtenu par suppression de l'arête (réputée existante) entre les sommets a et b.

La méthode « `est_un_pont()` » fournit quant à elle un diagnostic pour la question suivante : Étant donnés a et b deux sommets adjacents d'un graphe connexe, la suppression d'une arête d'extrémités a et b produit-elle ou non un nouveau graphe possédant plusieurs composantes connexes, si l'on fait exception d'un éventuel sommet isolé par suite de la suppression ?

graphe.py

```
    def est_un_pont(self, a, b):
        """ [ab] est-elle un pont ? """
        if not (b in self.adjacents(a)):
            return False
        r = self.graphe_reduit(a, b)
        return (len(r.composantes()) != 1)
```

La traduction en programme **Python** de l'algorithme de FLEURY suit :

graphe.py

```
    def recherche_Euler(self):
        """ algorithme de Fleury """
        g = self
        fin = (g.nombre_aretes() == 1)
        if fin:
            return g.sommets()
```

```
        p = []

        choix = []
        for s in g.sommets():
            if g.degre_sommet(s) % 2 == 1:
                choix.append(s)
        if len(choix) == 0:
            choix = g.sommets()

        a = choix[random.randrange(0, len(choix))]

        while not fin:

            u = g.adjacents(a)
            v = []
            for s in u:
                if not g.est_un_pont(a, s):
                    v.append(s)

            b = v[random.randrange(0, len(v))]

            p.append(a)
            g = g.graphe_reduit(a, b)
            a = b

            fin = (g.nombre_aretes() == 1)

        p.append(a)
        p.append(g.adjacents(a)[0])

        return p
```

Le principe de l'algorithme est le suivant, moyennant la vérification préalable de l'existence d'une solution :

a) Choisir au hasard un sommet.
b) Choisir une arête parmi toutes les arêtes adjacentes (au sommet de départ) qui ne sont pas un pont, et la conserver dans le parcours en construction.
c) Recommencer à la seconde étape, en partant du nouveau graphe réduit obtenu par la suppression de l'arête ci-dessus (et des précédentes), avec pour sommet de départ l'extrémité de cette arête, et ainsi jusqu'à épuisement des arêtes.

La méthode « `recherche_Euler()` » implémente l'algorithme de FLEURY, avec une modification dans l'initialisation : on choisit d'emblée un sommet du graphe de degré impair, lorsqu'il en existe, afin de construire dans ce cas un chemin eulérien. À défaut, un sommet est choisi aléatoirement afin de construire un cycle eulérien.

Remarquons qu'à la seconde étape, lorsque le seul choix possible est celui d'un pont, cela signifie que le sommet de départ est alors de degré 1, auquel cas le graphe réduit qui en résulte doit être amputé de ce sommet isolé, afin de préserver la connexité.

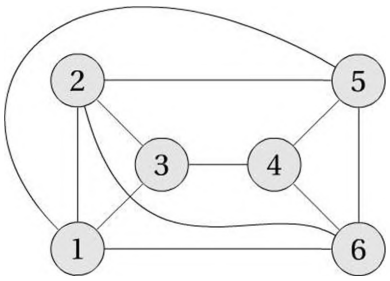


FIGURE 6.3 – Graphe non planaire en illustration de `test11()`

Terminal

```
$ ./test_graphes.py
----- test 7 -----
['F', 'E', 'D', 'C', 'B', 'A', 'C']
----- test 8 -----
['1', '2', '3', '4', '1', '3']
----- test 9 -----
['1', '2']
----- test 10 -----
['c', 'd', 'a', 'e', 'f', 'g', 'a', 'b', 'c']
----- test 11 -----
['4', '5', '1', '3', '2', '6', '1', '2', '5', '6', '4', '3']
$
```

2 Représentation des nombres

Nous allons construire quelques classes numériques destinées à représenter les nombres pour notre projet de système de calcul (quatre opérations et exponentiation) calculant convenablement sur les nombres rationnels.

Mentionnons immédiatement l'existence (depuis la version 2.6 du langage) des modules natifs « `fraction.py` » et « `decimal.py` », présentés plus avant, qui améliorent la gestion des calculs avec ce type de nombres. Les classes présentées ici sont bien sûr plus modestes, mais décrites dans un but d'étude.

Nous utiliserons par ailleurs des instances de classes que nous composerons soigneusement pour inclure une gestion des erreurs qui peuvent intervenir dès que nous effectuons une série de calculs en cascade.

En nous limitant aux décimaux, nous suivrons la démarche axiomatique de construction des nombres traditionnellement adoptée en Mathématiques, puis nous adopterons les idées suivantes :

- un nombre entier est un rationnel de dénominateur 1 ;

- un nombre décimal est un rationnel dont dénominateur peut s'exprimer sous la forme $2^p \times 5^q$, où p et q désignent des entiers naturels.

Ce choix peut paraître déroutant, mais il est motivé par les raisons suivantes : la conception d'un langage et d'un environnement de programmation se heurte toujours aux limites de temps à l'exécution, et surtout de mémoire de travail. Il en résulte des choix inévitables dont celui qui casse notre conception mathématique moderne des nombres : nous ne pouvons représenter qu'un nombre fini de nombres, en nature comme en quantité, dans une mémoire de calculateur. Pire, le calculateur manipule des nombres fondamentalement codés en système binaire : les « nombres machine » ne représentent qu'une infime partie des nombres mathématiques, dont certains n'ont aucune représentation correcte possible sur un nombre fini d'octets, en dehors d'approximations.

Le langage **Python** en souffre comme tout autre langage, même s'il possède un avantage décisif dans sa capacité à manipuler correctement les très grands entiers, là où le langage C et tant d'autres se perdent dès qu'il s'agit d'ajouter 1 et $\left(2^8\right)^4 - 1 = 4\,294\,967\,295$ (selon la norme ANSI C99, pour un entier non signé codé sur quatre octets) ; les librairies (modules logiciels additionnels apportant des extensions au langage de base) spécialisées en calcul scientifique ou les systèmes de calcul formel prennent alors le relais dans une certaine mesure...

Nous examinerons en détail une classe pour la représentation des nombres entiers, ainsi qu'une classe pour les nombres rationnels.

Nous prévoyons pour de tels nombres les quatre opérations et l'exponentiation, ainsi que la possibilité pour le système d'évaluer les expressions où s'enchaînent les différentes opérations. Nous devrons envisager les situations pour lesquelles le système sera placé devant l'évaluation d'un nombre mathématiquement indéfini, et décider d'un comportement du système dans un tel cas.

2.1 Quelques rappels et outils

On profite de l'occasion à venir d'implémenter un type rationnel pour revenir sur des détails qui distinguent nos définitions mathématiques de celles presque similaires disponibles dans les langages de programmation, afin d'éviter des confusions fréquentes et autres subtiles erreurs qui peuvent en découler, et dont la recherche de la cause fait inévitablement perdre un temps précieux.

Nous rappellerons brièvement les termes « troncature à l'unité », « partie entière », « partie fractionnaire » et « partie décimale » pour en donner une version en **Python** :

- les notations $\mathrm{E}(x) = \lfloor x \rfloor$ sont définies par : $\forall x \in \mathbb{R},\ \lfloor x \rfloor = \max\{n | n \in \mathbb{Z} \text{ et } n \leqslant x\}$;
- la notation $\lceil x \rceil$ est définie par : $\forall x \in \mathbb{R},\ \lceil x \rceil = \min\{n | n \in \mathbb{Z} \text{ et } n \geqslant x\}$;
- la notation [1] $\{x\}$ que nous appellerons « partie fractionnaire » du réel x, définie par :

$$\forall x \in \mathbb{R},\ \{x\} = x - \lfloor x \rfloor.$$

1. « Mathématiques concrètes » – GRAHAM, KNUTH, PATASCHNIK.

Avec ces définitions, notons la correspondance suivante entre `int(x)` et la fonction :

$$x \longmapsto \begin{cases} \lfloor x \rfloor & \text{pour tout } x \in \mathbb{R}^+, \\ \lceil x \rceil & \text{sinon} \end{cases}$$

La distinction est importante entre la fonction mathématique « partie entière » et le résultat obtenu par transtypage d'un nombre négatif. En effet, pour x positif, nous avons coïncidence entre $\mathrm{E}(x)$ et `int(x)`, mais lorsque x est négatif, on perd cette égalité.

Nous utiliserons la propriété ci-dessous par la suite, dès lors que nous aurons à passer d'une représentation à une autre pour un même nombre décimal :

$$\forall x \in \mathbb{R},\ x = \text{troncature\_unite}(x) + \text{partie\_decimale}(x)\ .$$

```
def tronc_unite(x):
    """ fonction troncature à l'unité """
    return int(x)
```

```
def E(x):
    """ fonction partie entière (au sens mathématique) """
    if x < 0:
        x = x - 1
    return tronc_unite(x)
```

```
def partie_frac(x):
    """ fonction partie fractionnaire """
    return (x - E(x))
```

```
def partie_dec(x):
    """ fonction partie décimale """
    return (x - tronc_unite(x))
```

Et prenons quelques exemples pour fixer les idées :

Terminal

```
>>> print(tronc_unite(-9.1))
-9
>>> print(E(-9.1))
-10
>>> print(partie_frac(-9.1))
0.9
>>> print(partie_dec(-9.1))
-0.1
>>>
```

Faisons quelques remarques :

a) Pour nos `entier` et `rationnel`, nous avons écarté de notre modèle la gestion de débordement de capacité des entiers machine. Pour intégrer une vérification de telles erreurs,

il suffit de compléter la gestion d'erreur du modèle en fixant, au choix du concepteur de la classe, un intervalle d'entiers machine dans lesquels les entiers machine sont considérés comme valides. Ainsi, chaque opération arithmétique doit contrôler elle-même si son résultat mathématique appartient effectivement ou non à l'intervalle de validité défini.

b) Le langage **Python** offre un module additionnel `decimal` pour la gestion fine des nombres décimaux, avec une précision de calcul bien supérieure à celle offerte par le type `float` standard.

2.2 Une première tentative et un échec

Nous allons nous baser sur la décomposition d'un décimal x, sous la forme usuelle construite à partir de la division euclidienne des entiers :

$$x = \epsilon\left(a + \frac{b}{10^n}\right)$$

avec $(a, b, n) \in \mathbb{N}^3$, et dans laquelle ϵ sera égal à 1 ou -1 selon que notre décimal est respectivement positif ou négatif.

```
def forme_frac(valeur =0):
    """ retourne le décimal sous forme p/q """
    if valeur == 0:
        return 0

    epsilon = 1
    v = valeur
    if v < 0:
        epsilon = -1
        v = -v

    a = E(v)           # cf. plus haut
    v = partie_dec(v)  # idem

    b = 0
    c = 1
    while v > 0:
        v *= 10; t = E(v); v -= t
        b = b*10 + t
        c *= 10

    return str(epsilon*(a*c + b)) + "/" + str(c)

print(forme_frac(-23))
print(forme_frac(-2.3))
```

Observons notre magnifique programme à la manœuvre :

Terminal

```
-23/1
-22999999999999998223643160599749535322189331054687 5/10000000000\
0000000000000000000000000000000000000000
```

L'échec est cuisant : l'algorithme est correct, mais inutilisable ! Sans entrer ici dans le détail, disons simplement que l'erreur provient de la représentation interne des nombres en **Python** (ce défaut existe dans la majorité des langages, pour des raisons de conception des unités logiques de calcul). Nous reviendrons sur ces questions dans un autre chapitre.

2.3 Une seconde tentative

Nous sommes donc contraints de contourner une difficulté des langages de programmation inhérente à la représentation des nombres flottants.

On va partir de la forme textuelle du nombre décimal (la forme lisible par les humains) et construire notre nombre décimal à l'aide des outils offerts par **Python** pour le traitement des chaînes de caractères et des listes. On fait l'hypothèse raisonnable (et c'est effectivement le cas et un point fort de **Python**) que le langage interprète correctement notre notation usuelle des nombres décimaux, puis on le décompose en une liste d'une ou deux séries de chiffres, séparés le cas échéant par la « virgule » (en fait, le point décimal dans les langages conçus outre-mer pour la plupart) :

```
def forme_frac(valeur =0):
    """ retourne le décimal sous forme p/q (version 2) """
    if valeur == 0:
        return 0

    negatif = (valeur < 0)
    v = valeur
    if negatif:
        v = -v

    chiffres = str(v).split('.') # merci Python!

    a = int(chiffres[0])
    b = 0
    n = 0

    if len(chiffres) == 2:
        b = int(chiffres[1])
        n = len(chiffres[1])

    q = 10 ** n
    p = a * q + b
    if negatif:
        p = -p

    return str(p) + "/" + str(q)
```

```
print(forme_frac(-23))
print(forme_frac(-2.3))
```

Le programme ci-dessus donne cette fois le résultat escompté.

2.4 Une classe pour les entiers

Nous commençerons par une classe simple qui implémente les nombres entiers, et leurs opérations.

Pour définir le modèle de notre type `entier`, nous choisissons deux variables d'instance :

- un champ « `self.__valeur` » contenant la valeur numérique de notre nombre entier ;
- un champ « `self.__valide` », valeur booléenne indiquant l'état de validité de notre nombre.

Nous définissons ci-dessous le constructeur de notre classe de représentation d'un nombre entier : il s'agit en fait d'une classe enveloppe, qui embarque un nombre entier (natif du langage **Python**) agrégé avec un descripteur d'état, indiquant la validité de ce nombre.

On notera le fait que, même en cas d'erreur, on veille à ne pas rompre le flot d'exécution du programme : une fonction est censée toujours rendre un résultat ; il appartient au programmeur de toujours contrôler la qualité (validité) du résultat retourné par l'appel de la fonction.

D'un point de vue conceptuel, on pourrait aller bien plus loin qu'un simple indicateur de validité, en élaborant une structure mathématique. Cette structure donnerait pour nos programmes une classe « super-nombre » par l'adjonction, à nos nombres mathématiques, d'un élément « pas-un-nombre », et en définissant de nouvelles opérations :

$$\begin{aligned}&\text{super-nombre} \equiv \text{nombre ou pas-un-nombre}\\&\text{nombre} \oplus \text{nombre} = \text{nombre}\\&\text{nombre} \oplus \text{pas-un-nombre} = \text{pas-un-nombre}\\&\ldots\end{aligned}$$

Voici le code :

```
class entier:

    def __init__(self, valeur =0, valide =True):
        self.__valide = valide
        if self.__valide:
            self.__valeur = valeur
        else:
            self.__valeur = 0

    def __str__(self):
        if self.__valide:
            return str(self.__valeur)
        else:
            return "(entier invalide)"
```

Il nous faut discuter un choix surprenant : la décision de construire à chaque fois un nouveau nombre `entier` invalide pour forcer une erreur. Une méthode plus satisfaisante met en œuvre des concepts moins simples, mais nous entraînerait bien plus loin que le cadre de ce livre. Nous pourrions ainsi définir ce qu'on appelle — selon la terminologie consacrée des « motifs de conception » — un singleton, c'est-à-dire un objet qui ne sera instancié qu'une fois seulement pour l'ensemble du programme. Un singleton fait office de variable globale pour le programme : le recours aux variables globales demeure une question non tranchée.

En l'espèce, nous pourrions créer un singleton `entier` construit sur les valeurs `(0, False)`, sur lequel nous retournerions une référence à chaque fois que nous avons besoin d'utiliser un « `entier` non défini », par exemple issu d'un calcul erroné.

Poursuivons l'étude de notre classe `entier` par quelques méthodes :

```
    def est_valide(self):
        return self.__valide

    def valeur(self):
        return int(self.__valeur)
```

Pour ce qui concerne la gestion des erreurs, nous avons simplement pris le parti d'utiliser une variable booléenne. Pour obtenir un diagnostic plus fin des erreurs (par exemple, après tentative de créer un rationnel à partir d'un nombre nul au cours d'une succession de calculs...), un gestionnaire d'erreur devrait être implémenté.

Nous présentons ensuite les méthodes pour les opérations mathématiques :

```
    def __add__(self, autre):
        if self.__valide and autre.__valide:
            return entier(self.__valeur + autre.__valeur)
        else:
            return entier(0, False)

    def __neg__(self):
        if self.__valide:
            return entier(-self.__valeur)
        else:
            return entier(0, False)

    def __sub__(self, autre):
        return self.__add__(autre.__neg__())
```

On remarquera que la seconde ligne de la méthode « `__add__()` » ne fait curieusement pas appel à la méthode (publique) `est_valide()` définie au sein de la classe, mais elle utilise directement la variable d'instance `self.__valide`. Cette méthode n'est pourtant pas inutile. C'est ici qu'il convient de distinguer deux utilisations de la valeur de validité :

- l'instance de classe inspecte elle-même son état de validité. L'instance a accès à toutes ses variables internes, sans déroger au principe d'encapsulation des données. De surcroît, passer une méthode alourdirait et ralentirait inutilement le programme ;

– par ailleurs, un objet *extérieur* utilisant la classe peut avoir besoin de contrôler la validité de l'`entier` qu'il manipule. La protection des données le prive de l'accès aux membres internes de la classe `entier`. C'est le rôle de la fonction `est_valide()` que de permettre de connaître l'état de validité d'une instance, tout en respectant le caractère privé de l'information. Ce type de fonction destiné à apporter, si besoin et selon le choix du programmeur, une visibilité à l'information est appelé « accesseur » en terminologie objet.

Lors de la conception d'une classe d'objets, il est particulièrement important d'être vigilant quant à l'utilisation de l'information, car plusieurs utilisateurs (un programmeur qui utilise un module, qu'il en connaisse ou non les détails ; une méthode invoquée depuis une classe ou depuis l'extérieur...) peuvent avoir ou non accès aux données, avec des privilèges différents. C'est le concepteur de la classe qui doit veiller à toutes les utilisations possibles (légitimes) de son code.

Pour l'ensemble de nos programmes, nous réduirons souvent notre travail en considérant la soustraction comme addition de l'opposé.

```
    def __mul__(self, autre):
        if self.__valide and autre.__valide:
            return entier(self.__valeur * autre.__valeur)
        else:
            return entier(0, False)

    def __truediv__(self, autre):
        if not(self.__valide and autre.__valide):
            return entier(0, False)
        if autre.__valeur == 0:
            return entier(0, False)
        if (self.__valeur % autre.__valeur) != 0:
            return entier(0, False)
        return entier(self.__valeur / autre.__valeur)
```

Nous donnons pour finir l'opérateur d'exponentiation pour la classe `entier` :

```
    def __pow__(self, autre):
        if not(self.__valide and autre.__valide):
            return entier(0, False)
        a = self.__valeur
        n = autre.__valeur
        if a == 0:
            if n <= 0:
                return entier(0, False)
            else:
                return entier()
        if n < 0:
            return entier(0, False)
        p = 1
        while n > 0:
            if n % 2 == 1:
```

```
            p *= a
        n //= 2
        a *= a
    return entier(p)
```

On peut observer pour l'exponentiation, un algorithme efficace basé sur la décomposition de l'exposant en base 2. On devrait en principe préférer l'exponentiation native fournie par le langage. La présentation de cet algorithme donne un exemple que l'on pourra réutiliser cette fois pour de nouveaux types définis par l'utilisateur (le type `polynome` en est un exemple présenté plus loin).

Nous pouvons effectuer quelques tests :

Terminal

```
>>> from entier import *
>>> a = entier(-232)
>>> print(a)
-232
>>> print(a.valeur())
-232
>>> print(a.est_valide())
True
>>> b = entier(4)
>>> x = a - b
>>> print(x)
-236
>>> x = a / b
>>> print(x)
-58
>>> x = a / (b - b)
>>> print(x)
(entier invalide)
>>> a = entier(23)
>>> b = entier(9)
>>> x = a ** b
>>> print(x)
1801152661463
>>> b = entier(-4)
>>> x = a ** b
>>> print(x)
(entier invalide)
>>>
```

On observera ici le fonctionnement du mécanisme de la surcharge d'opérateur : en raison de la syntaxe particulière employée pour nommer nos opérateurs arithmétiques, **Python** incorpore en quelque sorte à son propre langage ces nouvelles définitions. Ce mécanisme permet donc d'écrire de façon équivalente mais bien plus commode :

Terminal

```
>>> x = a.__add__(b)
>>> x = a + b
```

Enfin nous complétons le fichier `entier.py` par deux fonctions, externes à la classe elle-même, que nous utiliserons pour notre classe `rationnel` :

```
def pgcd_naturels(a, b):
    if b == 0:
        return a
    else:
        return pgcd_naturels(b, a % b)

def pgcd_entiers(a, b):
    if a.est_valide() and b.est_valide():
        return entier(pgcd_naturels(abs(a.valeur()), abs(b.valeur())))
    else:
        return entier(0, False)
```

2.5 Une classe pour les rationnels

Nous pouvons maintenant construire notre classe `rationnel` à partir de la classe `entier`.

Nous soulignons ici un choix de conception : la structure de l'objet représentant un nombre rationnel contient trois informations (variables d'instance), alors que deux entiers suffisent mathématiquement à décrire un nombre rationnel. Chaque instance de notre objet `rationnel` comportera deux champs de type `entier` pour le numérateur et le dénominateur, ainsi qu'un champ pour la validité. Au passage, le champ de validité aurait pu être considéré comme un type entier au lieu du type booléen, pour contenir par exemple un entier représentant un code d'erreur à l'intention de l'utilisateur.

L'extrait de code ci-dessous commence par une fonction utilitaire `verif_entier()`, utilisée ensuite dans le constructeur de notre classe `rationnel`. Cette fonction détermine si un nombre entier peut s'exprimer sous la forme $2^p \times 5^q$, où p et q désignent des entiers naturels. Cette fonction appliquée au dénominateur nous indique si le nombre rationnel est un nombre décimal, ce qui autorise alors une représentation sous la forme ordinaire plutôt que fractionnaire.

rationnel.py

```
from entier import *

def verif_entier(n):
    if n == 1:
        return True
    else:
        if n % 2 == 0:
            return verif_entier(n/2)
        if n % 5 == 0:
            return verif_entier(n/5)
        return False
```

```
class rationnel:

    def __init__(self, num =0, denom =1, valide =True):
        valide = valide and (type(num) == int or isinstance(num, entier))
        valide = valide and (type(denom) == int or isinstance(denom, entier))

        if valide:
            if type(num) == int:
                num = entier(num)
            if type(denom) == int:
                denom = entier(denom)

        valide = valide and num.est_valide() and denom.est_valide()

        self.__valide = valide and (denom.valeur() != 0)

        if self.__valide:
            if denom.valeur() < 0:
                num, denom = -num, -denom
            d = pgcd_entiers(num, denom)
            self.__num, self.__denom = num / d, denom / d
        else:
            self.__num, self.__denom = entier(0), entier(1)
```

La méthode suivante permet d'obtenir une représentation de l'objet `rationnel` sous la forme d'une chaîne alphanumérique :

```
    def __str__(self):
        """ représentation sous forme de chaîne """
        if self.__valide:
            if self.__denom.valeur() == 1:
                return str(self.__num)
            if verif_entier(self.__denom.valeur()):
                return str(self.__float__())
            else:
                return str(self.__num) + "/" + str(self.__denom)
        return "(rationnel invalide)"

    def __float__(self):
        """ représentation sous forme de nombre réel machine """
        if self.__valide:
            a, b = self.__num.valeur(), self.__denom.valeur()
            return (float(a)/float(b))
        else:
            return float(0)
```

On remarquera l'utilisation de valeurs par défaut dans le constructeur (`=0` et `=1`). Cette facilité nous permet sans effort supplémentaire de construire par la suite des nombres rationnels particuliers comme les nombres entiers (sans besoin cette fois de préciser le dénominateur) ; l'appel du constructeur sans paramètre construira donc une instance représentant notre nombre 0 ordinaire.

On observera aussi que le constructeur possède une taille cette fois bien plus imposante : nous devons d'abord vérifier les états de validité, puis réduire les numérateur et dénominateur, normaliser le nombre rationnel en imposant un dénominateur positif. Plus important, notre constructeur — la méthode `__init__()` — est en mesure de construire un nombre `rationnel` depuis le type entier natif de **Python**, comme depuis notre propre type `entier`.

Nous mettons à disposition de l'utilisateur de la classe une méthode permettant la consultation, sous forme textuelle, de la fraction considérée, ainsi qu'une méthode `__float__()` qui délivre une valeur numérique du rationnel, au besoin sous la forme d'une approximation réalisée par **Python** lui-même.

En procédant ainsi, nous avons éliminé la possibilité de construire un nombre `rationnel` valide à partir d'un nombre `entier` invalide comme numérateur ou dénominateur.

Nous pouvons procéder à l'écriture de nos opérateurs mathématiques pour l'addition et la soustraction :

```
    def __add__(self, autre):
        """ somme de deux rationnels """
        if self.__valide and autre.__valide:
            a, b = self.__num, self.__denom
            p, q = autre.__num, autre.__denom
            return rationnel(a*q + b*p, b*q)
        else:
            return rationnel(0, 1, False)

    def __neg__(self):
        if self.__valide:
            a, b = self.__num, self.__denom
            return rationnel(-a, b)
        else:
            return rationnel(0, 1, False)

    def __sub__(self, autre):
        return self.__add__(autre.__neg__())
```

Nous donnons pour la forme le produit et le quotient de deux objets de type `rationnel` :

```
    def __mul__(self, autre):
        if self.__valide and autre.__valide:
            a, b = self.__num, self.__denom
            p, q = autre.__num, autre.__denom
            return rationnel(a*p, b*q)
        else:
            return rationnel(0, 1, False)

    def __truediv__(self, autre):
        if self.__valide and autre.__valide:
            a, b = self.__num, self.__denom
            p, q = autre.__num, autre.__denom
            return rationnel(a*q, b*p)
        else:
```

```
            return rationnel(0, 1, False)
```

Il reste possible de choisir la définition du quotient comme produit par l'inverse ; il convient de s'interroger sur la consommation de ressources, par exemple en étudiant par une série de tests quelle est l'implémentation la plus efficace.

```
    def __pow__(self, autre):
        if self.__valide and autre.__valide:
            a, b = self.__num, self.__denom
            p, q = autre.__num, autre.__denom
            if (a.valeur() == 0) and (p.valeur() <= 0):
                return rationnel(0, 1, False)

            if p.valeur() < 0:
                p, a, b = -p, b, a

            if b.valeur() < 0:
                a, b = -a, -b

            if q.valeur() == 1:
                return rationnel(a**p, b**p)

        return rationnel(0, 1, False)
```

Le lecteur pourra compléter la liste des opérateurs par les méthodes qui construisent la comparaison de deux rationnels ainsi que que d'une nouvelle instance donnant le nombre inverse ; cette dernière méthode rend possible la division comme multiplication par l'inverse.

Les quelques fonctions qui suivent n'ont pas d'intérêt mathématique. Leur présence est seulement motivée par le respect du très important principe d'encapsulation des données. Après tout, le conducteur n'a aucun besoin de connaître le fonctionnement intime et dans le détail du moteur de son véhicule, et il s'en accommode parfaitement dans son utilisation normale. De la même façon, un utilisateur de la classe `rationnel` se satisfait de savoir que la somme de deux rationnels est un rationnel, avec la possibilité de contrôler que la somme calculée est correcte. Cela peut paraître banal, mais cela le devient un peu moins lorsqu'on considère que l'utilisateur de la classe n'est plus un humain (en principe apte au contrôle du bon déroulement des calculs), mais un calculateur, qui effectuera avec docilité tout ce qui lui est demandé, nonobstant l'absurdité des résultats intermédiaires.

Un utilisateur de la classe doit à tout prix éviter de manipuler les données internes de la classe, sous peine d'en casser l'intégrité : le concepteur de la classe en a pris la responsabilité lors de la conception de la classe.

En dernier lieu, le concepteur de la classe peut, à son initiative seulement, mettre à disposition de l'utilisateur de la classe des accesseurs (il peut s'avérer pratique de consulter la valeur du dénominateur) ou même des mutateurs, bien que ces derniers portent en eux les dangers mentionnés précédemment.

```
    def est_valide(self):
        return self.__valide
```

```
    def num(self):
        return self.__num

    def denom(self):
        return self.__denom
```

Les quelques tests ci-dessous montrent en fait la partie « visible » depuis l'extérieur de la classe, par l'utilisateur de la classe `rationnel` :

```
if __name__ == "__main__":
    a = rationnel(-23)
    print(a)

    b = rationnel(entier(-23))
    print(b)

    c = rationnel(4, 0)
    print(c)

    d = rationnel(-400, -3760)
    print(d)
    print(float(d))

    d = rationnel(entier(-400), entier(-3760))
    print(d)
    print(float(d))

    x = a + d
    print(x)

    x = c + d
    print(x)

    x = d ** rationnel(-5)
    print(x)
```

Terminal

```
-23
-23
(rationnel invalide)
5/47
0.106382978723
5/47
0.106382978723
-1076/47
(rationnel invalide)
73390.40224
```

On notera le dernier paragraphe du fichier `rationnel.py` commençant par la ligne de l'instruction `if` : notre programme ci-dessus est complet au sens où il peut en effet être considéré

comme une unité autonome pour implémenter le type `rationnel` (moyennant l'inclusion du fichier définissant le type `entier`). Ce paragraphe permet d'utiliser le fichier source comme un programme autonome (par exemple pour des tests) ou comme module au service d'un programme plus conséquent.

Exercice 67. Étendre la classe `entier` afin que le type effectue lui-même le contrôle du débordement de capacité, étant fixées par le programmeur les valeurs d'un « plus petit » et d'un « plus grand » entier. Entre ces deux entiers, l'arithmétique des quatre opérations, et l'exponentiation seront garanties par le type.

Exercice 68. Pour l'écriture des nombres décimaux sous la forme usuelle, nous nous sommes reposés sur le fait que **Python** effectue convenablement cette tâche. Écrire un programme **Python** qui délivre cette forme, à partir de la donnée de deux nombres entiers.

3 Listes

Nous traiterons principalement deux types particuliers de listes :

- les files (ou *queues*, encore appelées FIFO pour *first in, first out*) : cette structure est une liste séquentielle appropriée au traitement de ses éléments selon le protocole « premier entré, premier sorti » ;
- les piles (ou *stacks* ou LIFO) : comme le nom l'indique, ce type de liste est la métaphore de la pile d'assiettes « dernier entré, premier sorti ».

Remarque. Afin d'éviter toute confusion, le terme anglo-saxon *file* désignant le plus souvent en langage informatique un fichier de données, nous parlerons de *queue* en lieu et place de *file* dans la suite du chapitre.

3.1 Trois versions du tri récursif d'une liste

Avant d'étudier la création des deux types évoqués ci-dessus, prenons le temps d'observer quelque peu le type liste et les facilités proposées par **Python**, notamment d'un point de vue « programmation orientée objet ».
On propose ici trois versions successives, en termes d'organisation du code source, de programme de tri récursif et d'obtention du maximum d'une liste de nombres.
Commençons par notre algorithme basique, volontairement maladroit :

maxiRecListe1.py

```
from random import *

def initialiser(l):
    """ remplissage de la liste avec des nombres aléatoires """
    for i in range(len(l)):
        l[i] = randrange(1, 100)

def maxi_rec(l):
```

```
    """ recherche récursive du maximum """
    if len(l) > 1:
        maxi = maxi_rec(l[:-1])
        if maxi > l[-1]:
            return maxi
        else:
            return l[-1]
    else:
        return l[0]

if __name__ == "__main__": # programme principal

    # une liste de dix nombres 0
    liste = [0 for i in range(10)]
    initialiser(liste)

    print(liste)
    print(maxi_rec(liste))
```

Comme on le constate, l'utilisateur possède tous les détails d'implémentation sous les yeux. En particulier, le dimensionnement correct de la liste et son initialisation dépendent de lui. On peut toutefois déléguer certains traitements fastidieux au programme, et permettre à l'utilisateur de concentrer son attention sur le problème qu'il cherche à résoudre, non sur des détails externes.

On observe de plus que les données sont séparées de leur traitement.

Dans une seconde version, le programme est amélioré par la gestion des listes en **Python**. La liste est initialement vide, puis remplie à hauteur de la longueur désirée. Le programme comporte néanmoins toujours le défaut de devoir demander explicitement l'initialisation de la liste d'entiers.

maxiRecListe2.py

```
from random import *

def initialiser(l, n):
    """ remplissage (récursif) de la liste avec des nombres aléatoires """
    if n > 0:
        l.append(randrange(1, 100))
        initialiser(l, n-1)

def maxi_rec(l):
    """ recherche récursive du maximum """
    if len(l) > 1:
        maxi = maxi_rec(l[:-1])
        if maxi > l[-1]:
            return maxi
        else:
            return l[-1]
    else:
        return l[0]
```

```
if __name__ == "__main__":

    liste = [] # la liste est initialement vide
    initialiser(liste, 10)

    print(liste)
    print(maxi_rec(liste))
```

Nous donnons enfin une version « orientée objet » de ce programme : son intérêt réside cette fois dans l'encapsulation des données. Cette notion réalise le regroupement logique des données et leur traitement. Du point de vue de l'utilisateur de la classe, celui-ci n'a aucune connaissance des détails d'implémentation de la liste, et peut l'utiliser telle quelle. On a ainsi ajouté une couche d'abstraction – une protection – puisque l'utilisateur n'intervient pas dans le fonctionnement interne de l'objet. Lors de la création de la liste, l'utilisateur se contente de spécifier la taille de la liste des nombres aléatoires.

maxiRecListe3.py

```
from random import *

def maxi_rec(l):
    """ recherche récursive du maximum """
    if len(l) > 1:
        maxi = maxi_rec(l[:-1])
        if maxi > l[-1]:
            return maxi
        else:
            return l[-1]
    else:
        return l[0]

class table(object):

    def __init__(self, taille):
        self.__liste = [randrange(1, 100) for n in range(taille)]

    def __str__(self):
        return str(self.__liste)

    def maximum(self):
        return maxi_rec(self.__liste)

if __name__ == "__main__":
    liste = table(10)
    print(liste)
    print(liste.maximum())
```

Dans cette forme du programme, la liste est automatiquement et complètement construite dès sa création (placée dans un état conforme à celui attendu, c'est-à-dire une liste de dix entiers quelconques) ; l'information est parfaitement protégée de toute mauvaise manipulation par l'utilisateur.

3.2 Une classe pour encapsuler l'information

Nous envisageons de constituer des listes dont les éléments sont de natures variées, et *a priori* non déterminées lors de la conception de la classe. Pour ces listes, notre unité d'information est le nœud, qui est une enveloppe destinée à recevoir l'information qui nous intéresse (un nombre, un monôme...) et surtout un lien (un pointeur) sur un autre membre de la liste, que l'on nommera `self.__suivant`. Par pointeur, nous entendons une variable dont le rôle consiste à désigner (pointer) une autre variable.

Note. Lorsque nous parlons de pointeur, nous désignons une variable qui indique l'adresse d'une autre variable. Le langage **Python** n'encourage pas la manipulation directe d'un type particulier « pointeur » comme par exemple le langage C. Les langages qui offrent au programmeur le concept de pointeur permettent de manipuler arithmétiquement les variables et leurs adresses physiques en mémoire.

noeud.py

```
class noeud(object):

    def __init__(self, donnee =None, suivant =None):
        self.__donnee = donnee
        self.__suivant = suivant

    def __str__(self):
        return str(self.__donnee)
```

Donnons quelques commentaires sur la structure de la classe. Comme pour les listes natives de **Python**, nous pourrons stocker dans un nœud tout type d'information, quelle qu'en soit sa nature. Cette liberté est précieuse et souvent absente des autres langages.

Terminal

```
>>> from noeud import *
>>> a = noeud(7)
>>> print(a)
7
>>> from rationnel import *
>>> b = noeud(rationnel(-50, -480), a)
>>> print(b)
5/48
>>> c = noeud("chouette")
>>> print(c)
chouette
>>>
```

En plus du constructeur habituel pour la création du nœud, nous avons respecté l'habitude fort pratique de créer une méthode `__str__()` qui retournera une description textuelle de l'information contenue dans le nœud. Comme on peut le constater sur cet exemple, notre liste est à même d'accueillir des données hétérogènes. D'autre part, le nœud b pointe sur le nœud a lors de sa création, au contraire des deux autres nœuds a et c qui ne pointent rien (la valeur `None`).

Les deux types de listes que nous allons considérer posséderont les fonctionnalités suivantes :

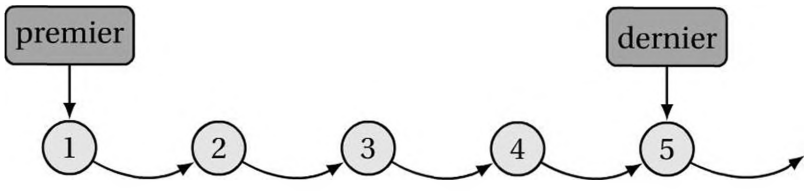


FIGURE 6.4 – Une structure de queue

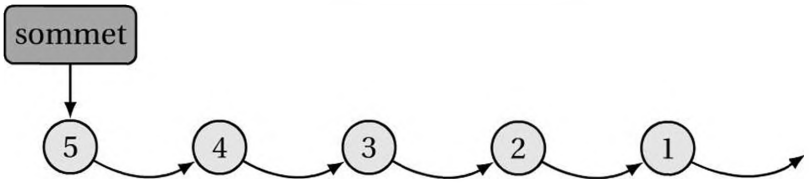


FIGURE 6.5 – Une structure de pile (le sommet apparaît à gauche)

- nous voulons pouvoir compléter la liste par adjonction d'un nouvel élément, à droite ou en fin de liste pour une queue, et pour une pile le placer au sommet (empiler) ;
- pour une queue, nous souhaitons pouvoir retirer le premier élément introduit dans la queue (le plus à gauche), ou pour une structure de pile, retirer l'élément au sommet de la pile (dépiler).
- enfin, nous souhaitons pouvoir consulter, sans modifier les éléments d'une queue ou d'une pile, le premier élément à gauche ou l'élément au sommet de la pile.

Dans les deux listes représentées en figures 6.4 et 6.5, nous avons inséré dans le même ordre séquentiel les entiers 1, 2, 3, 4 et 5. Néanmoins, le premier élément lu (retiré) de la file sera 1 (premier élément historiquement inscrit) ; le premier élément lu dans la pile sera le dernier élément inscrit.

Nous ne manipulerons nos objets de type queue ou pile que par les « poignées » que constituent les pointeurs `premier`, `dernier` et `sommet` :

- `premier` nous permet de lire (et retirer) le premier élément de la queue ;
- `dernier` nous permet d'inscrire immédiatement (sans recherche) après le dernier élément un nouveau dernier élément ;
- `sommet` permet indifféremment d'empiler ou de dépiler l'élément en sommet de pile.

On notera qu'en théorie un seul pointeur suffit à gérer une queue, mais au risque de dégrader les performances dans l'utilisation de la queue. Le rôle du pointeur `dernier` est d'éviter pour chaque insertion de parcourir la file en intégralité pour retrouver son dernier élément.

Signalons enfin que nous nous interdisons l'insertion et la suppression en milieu de pile ou de queue ; bien que cela soit possible, ces fonctionnalités présentent peu d'intérêt en pratique pour une liste séquentielle. Pour ce type de manœuvre, l'utilisation d'un arbre binaire est plus

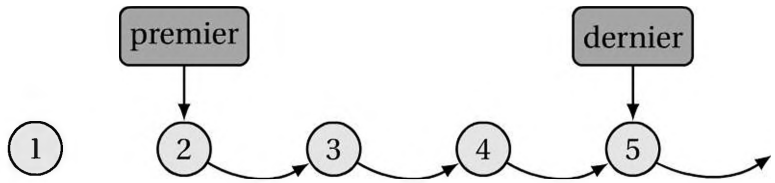


FIGURE 6.6 – Retrait dans une queue

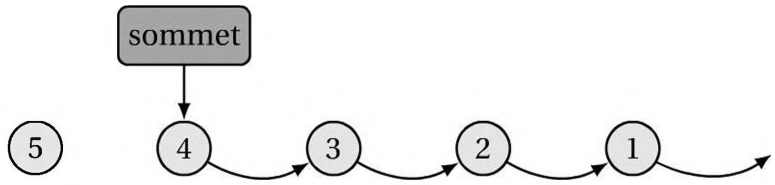


FIGURE 6.7 – Retrait dans une pile

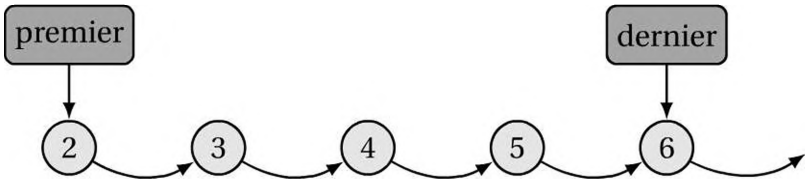


FIGURE 6.8 – Insertion dans une queue

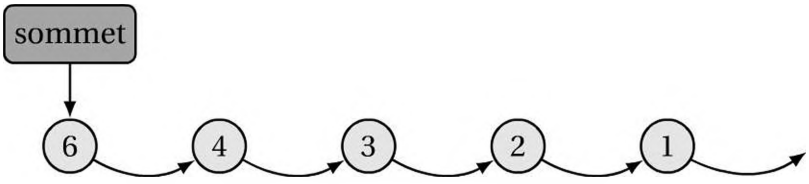


FIGURE 6.9 – Insertion dans une pile

judicieuse. Nous reviendrons sur cette question dans la section réservée aux arbres binaires de recherche.

Munis d'un objet de type noeud, nous allons maintenant constituer les modèles d'objet pour les listes FIFO et LIFO. Ces objets peuvent en fait ne comporter qu'une seule variable d'instance, pointant un nœud et suffisante pour la manipulation de la queue ou de la pile. Par exemple, nous choisissons un pointeur désignant l'élément le plus à gauche pour la queue, ou le sommet pour une pile. Cependant, nous donnerons deux variantes d'implémentation :

- pour une queue : un pointeur sur le premier élément de la queue, ainsi qu'un pointeur sur le second élément de la queue. Ce choix s'explique par le fait qu'on agit aux deux extrémités de la queue (lecture au début, écriture à la fin) ;
- pour une pile : uniquement un pointeur sur le sommet de la pile. En effet, dans une pile, les écritures et lectures s'effectuent toujours sur le sommet.

3.3 Une classe pour la structure de queue

Avant de présenter la classe queue, complétons la classe noeud par quelques méthodes accesseurs et mutateurs :

noeud.py

```
class noeud(object):
    """ noeud (liste simplement chaînée) """

    def __init__(self, donnee =None, suivant =None):
        """ constructeur """
        self.__donnee = donnee
        self.__suivant = suivant

    def __str__(self):
        return str(self.__donnee)

    def get_donnee(self):
        """ accesseur """
        return self.__donnee

    def set_donnee(self, d):
        """ mutateur """
        self.__donnee = d

    def get_suivant(self):
        return self.__suivant

    def set_suivant(self, s):
        self.__suivant = s
```

Le code source de la classe queue suit :

queue.py

```
from noeud import *
```

```
class queue(object):
    """ queue """

    def __init__(self):
        self.__premier = None
        self.__dernier = None

    def __str__(self):
        s = []
        n = self.__premier
        while not (n is None):
            s.append(n.get_donnee())
            n = n.get_suivant()
        return str(s)

    def est_vide(self):
        return (self.__premier is None)

    def ecrire_dernier(self, d):
        n = noeud(d)
        if self.__premier is None:
            self.__premier = n
        else:
            self.__dernier.set_suivant(n)
        self.__dernier = n

    def lire_premier(self):
        if self.__premier is None:
            return None
        n = self.__premier
        d = n.get_donnee()
        self.__premier = n.get_suivant()
        n = None
        return d
```

On peut ajouter les lignes suivantes au fichier queue.py, pour les tests :

```
if __name__ == "__main__":
    q = queue()
    q.ecrire_dernier("lapin")
    q.ecrire_dernier("chasseur")
    q.ecrire_dernier(2)
    q.ecrire_dernier("le")
    q.ecrire_dernier("retour")

    print(q)
    while not q.est_vide():
        print(q.lire_premier())
```

3.4 Une classe pour la structure de pile

pile.py

```
from noeud import *

class pile(object):
    """ pile """

    def __init__(self):
        self.__sommet = None

    def __str__(self):
        s = []
        n = self.__sommet
        while not (n is None):
            s.append(n.get_donnee())
            n = n.get_suivant()
        return str(s)

    def est_vide(self):
        return (self.__sommet is None)

    def empiler(self, d):
        n = noeud(d)
        n.set_suivant(self.__sommet)
        self.__sommet = n

    def depiler(self):
        if self.__sommet is None:
            return None
        n = self.__sommet
        d = n.get_donnee()
        self.__sommet = n.get_suivant()
        n = None
        return d

if __name__ == "__main__":
    p = pile()
    p.empiler("GNU")
    p.empiler("is")
    p.empiler("not")
    p.empiler("Unix")

    print(p)
    while not p.est_vide():
        print(p.depiler())
```

Comme on le constate, l'acronyme récursif du *GNU* est restitué à l'envers :

Terminal

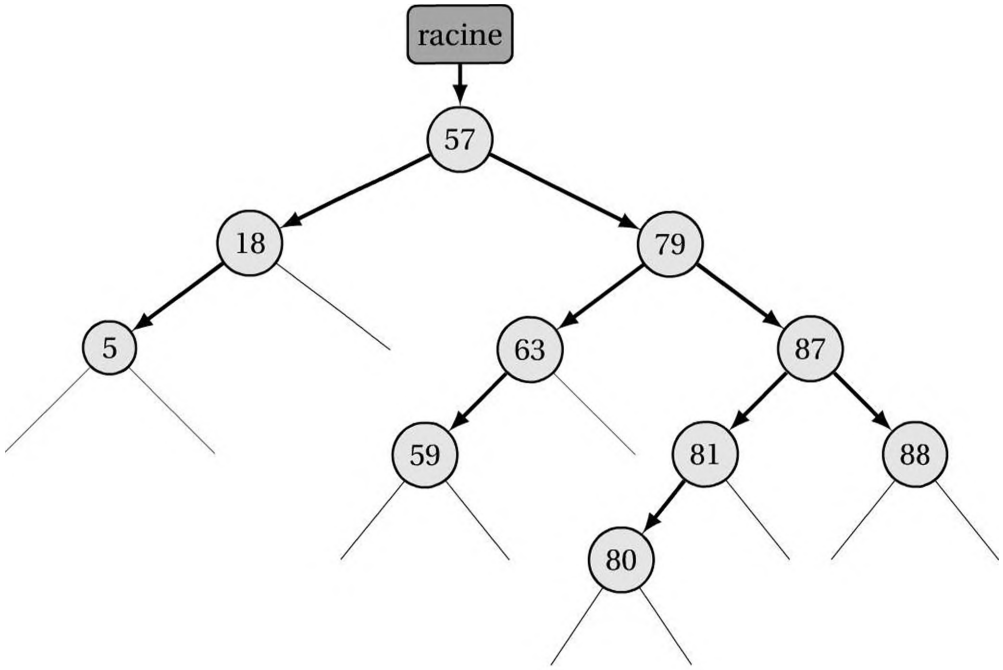


FIGURE 6.10 – Un arbre binaire de recherche

```
$ ./queue.py
['Unix', 'not', 'is', 'GNU']
Unix
not
is
GNU
$
```

4 Arbres binaires

Nous nous intéressons ici à un type particulier d'arbre binaire appelé *arbre binaire de recherche* (voir figure 6.10), dont la structure peut évoluer lors des adjonctions ou suppressions de nouveaux éléments, tout en restant stable selon un ordre défini sur les nœuds.

4.1 Une classe pour la structure de nœud double

La propriété principale pour un arbre binaire de recherche est la garantie que, pour tout nœud n de l'arbre, tout membre du sous-arbre gauche de n est inférieur à n'importe quel nœud membre de n'importe quel sous-arbre du sous-arbre droit de n.

Une autre garantie non moins importante réside dans l'unicité de chaque valeur dans un arbre binaire de recherche.

De telles garanties confèrent une structure dichotomique à l'arbre. Ce type de structure est particulièrement bien adapté à la recherche, à l'insertion et à la suppression de nœuds. En

effet, la construction d'un arbre binaire s'opère en respectant immédiatement pour chaque nouvelle donnée la conservation de l'ordre sur l'arbre. Il en va de même pour la suppression.

noeud.py

```
class noeud(object):
    """ noeud double (deux fils) """

    def __init__(self, num, gauche =None, droite =None):
        """ constructeur """
        self.__num = num
        self.__gauche = gauche
        self.__droite = droite

    def montrer(self, decal_horiz =0):
        """ imprimer le noeud avec ses fils """
        if self.__droite:
            self.__droite.montrer(decal_horiz + 3)
        print(" " * decal_horiz, self.__num)
        if self.__gauche:
            self.__gauche.montrer(decal_horiz + 3)

    def chercher(self, num):
        """ chercher une valeur """
        if num == self.__num:
            return True

        if num < self.__num:
            if self.__gauche:
                return self.__gauche.chercher(num)
            else:
                return False
        else:
            if self.__droite:
                return self.__droite.chercher(num)
            else:
                return False

    def inserer(self, num):
        """ insérer une valeur """
        if num != self.__num:
            if num < self.__num:
                if self.__gauche:
                    self.__gauche.inserer(num)
                else:
                    self.__gauche = noeud(num)
            else:
                if self.__droite:
                    self.__droite.inserer(num)
                else:
                    self.__droite = noeud(num)
```

```
        else:
            print "(pas de doublon dans un arbre binaire de recherche)"

    def plus_grand(self):
        if self.__droite:
            return self.__droite.plus_grand()
        else:
            return self.__num

    # ... méthode plus_petit() ...

    def get_num(self):
        return self.__num

    def set_num(self, num):
        self.__num = num

    def fils_gauche(self):
        """ accesseur fils gauche """
        return self.__gauche

    # ... méthode fils_droite() ...
```

La méthode récursive suivante permet le comptage des nœuds des deux sous-arbres :

noeud.py

```
    def noeuds(self):
        n = 1
        if self.__gauche:
            n = n + self.__gauche.noeuds()
        if self.__droite:
            n = n + self.__droite.noeuds()
        return n
```

La recherche et l'insertion d'un élément dans un arbre binaire de recherche ne posent pas de difficulté particulière. En revanche, la suppression est moins immédiate. C'est ici que les garanties évoquées en début de section jouent tout leur rôle. Lorsque nous devons supprimer un nœud (dont on aura préalablement vérifié l'existence), on adoptera la stratégie suivante :

- si le nœud est une feuille, il est supprimé sans autre forme de procès ;
- si le nœud est un fils unique, on le supprime en lui substituant son fils ;
- l'affaire est plus délicate lorsque le nœud comporte deux fils. Dans ce cas, nous savons que toute donnée contenue dans le sous-arbre gauche de notre nœud est réputée plus petite que la valeur de notre nœud. Nous pouvons alors échanger la valeur de notre nœud soit avec la plus grande valeur contenue dans le sous-arbre gauche, soit avec la plus petite valeur contenue dans le sous-arbre droit. Puis nous supprimons la feuille qui contenait cette valeur ainsi dupliquée. Cet échange conserve les garanties sur l'ordre des nœuds de l'arbre.
  On relèvera que cette suppression offre un choix au programmeur. On peut laisser ce choix aléatoire ou, au prix d'efforts supplémentaires de conception, faire en sorte de

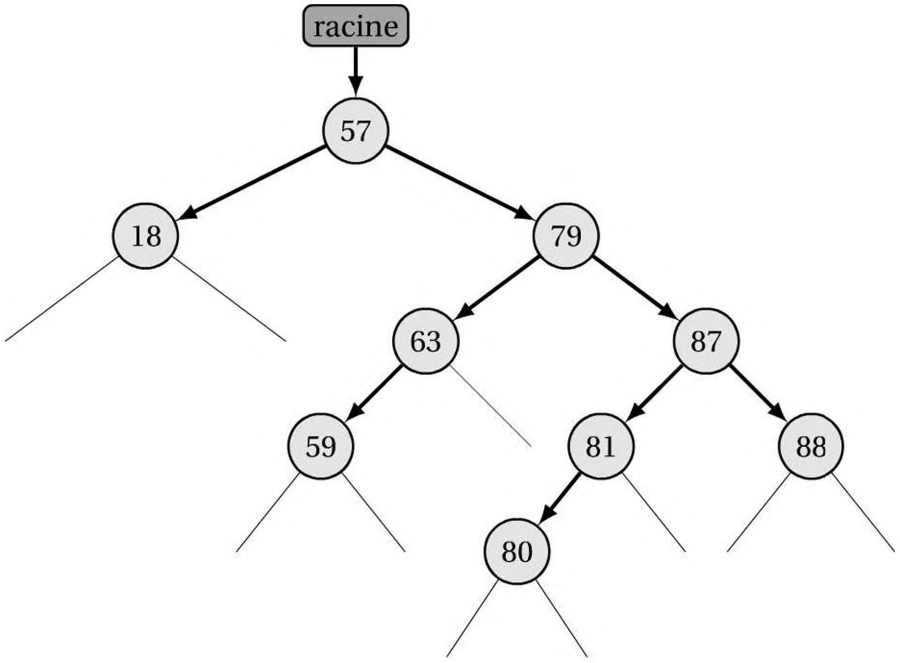


FIGURE 6.11 – Après suppression de la feuille 5 (dans l'arbre 6.10)

conserver un équilibre sur les hauteurs respectives des deux sous-arbres de chaque nœud (on agrémente alors chaque nœud d'un compteur pour compenser la différence des hauteurs des deux sous-arbres). Une telle solution sera privilégiée si on souhaite préserver un arbre binaire d'une dégénérescence en liste, pour des raisons de performance.

noeud.py

```
    def chercher_parent(self, num):
        if num == self.__num:
            return None

        if num < self.__num:
            if self.__gauche.__num == num:
                return self
            else:
                return self.__gauche.chercher_parent(num)
        else:
            if self.__droite.__num == num:
                return self
            else:
                return self.__droite.chercher_parent(num)

    def supprimer_noeud(self, num, parent):
        if num < self.__num:
            self.__gauche.supprimer_noeud(num, parent)
        else:
            if num > self.__num:
                self.__droite.supprimer_noeud(num, parent)
```

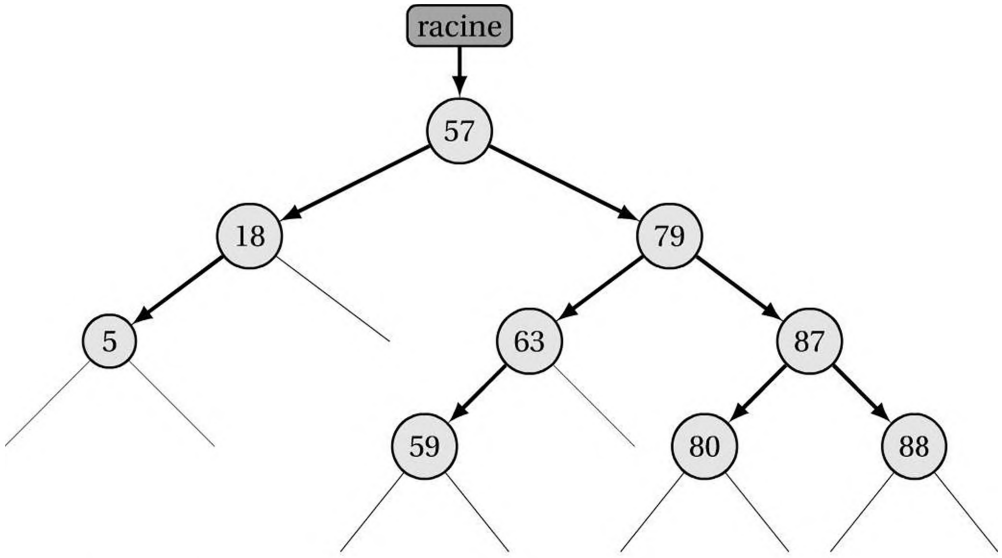


FIGURE 6.12 – Après suppression du nœud 81 (dans l'arbre 6.10)

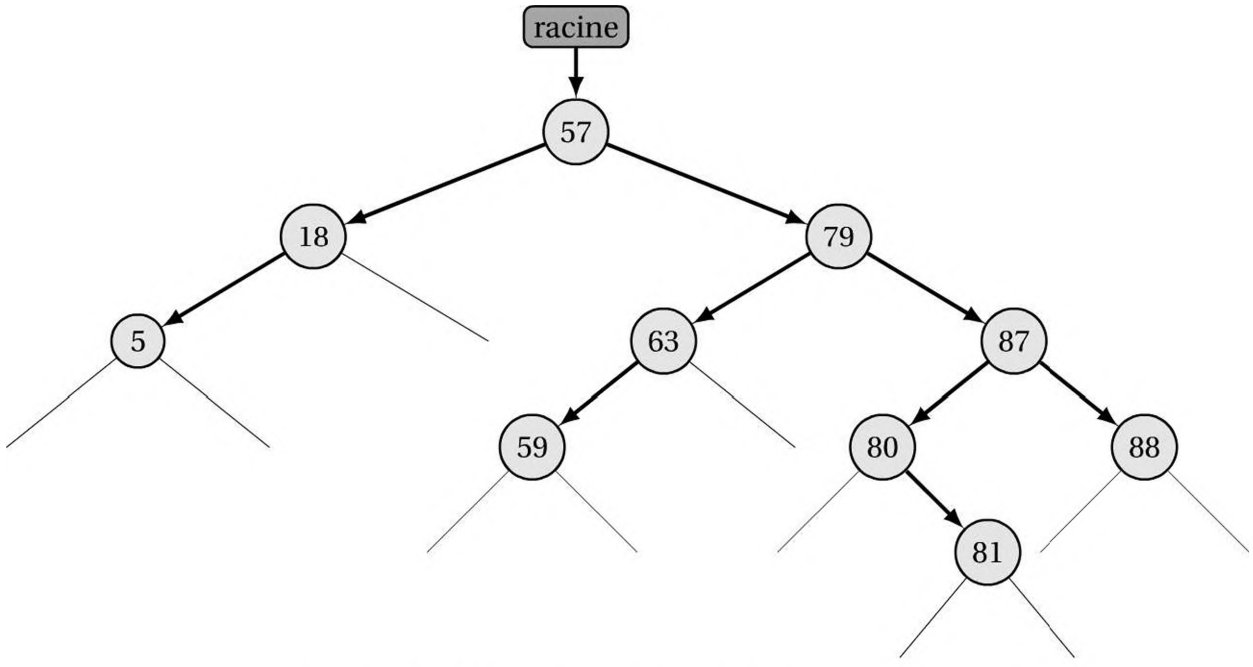


FIGURE 6.13 – Après insertion de la feuille 81 (dans l'arbre 6.10)

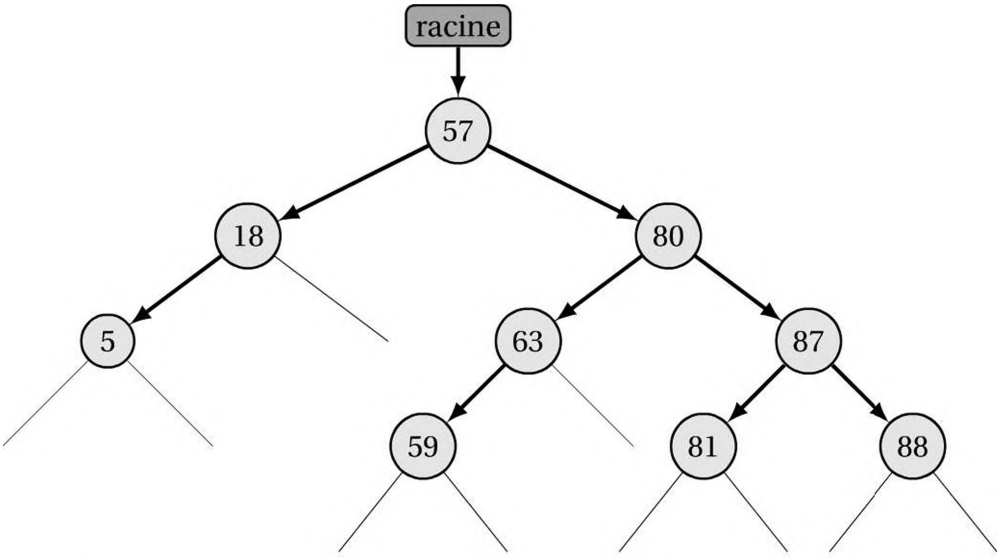


FIGURE 6.14 – Après suppression du nœud 79 (dans l'arbre 6.10)

```
            else:
                if self.__gauche is None and self.__droite is None: # aucun fils
                    if parent.__gauche is self:
                        parent.__gauche = None
                    else:
                        parent.__droite = None
                else:
                    if self.__gauche is None or self.__droite is None: # fils
                        unique

                        if self.__gauche:
                            t = self.__gauche
                        else:
                            t = self.__droite

                        if parent.__gauche is self:
                            s = parent.__gauche
                            parent.__gauche = t
                        else:
                            s = parent.__droite
                            parent.__droite = t
                        s.__gauche = None
                        s.__droite = None

                    else: # deux fils
                        if self.__gauche.noeuds() > self.__droite.noeuds():
                            x = self.__gauche.plus_grand()
                        else:
                            x =  self.__droite.plus_petit()
                        self.supprimer(x)
                        self.__num = x
```

```
    def supprimer(self, num):
        """ supprimer une valeur (réputée existante) """
        parent = self.chercher_parent(num)
        self.supprimer_noeud(num, parent)
```

Les méthodes suivantes permettent de présenter les nœuds sous une forme ordonnée :

noeud.py

```
    def montrer_ordre_croissant(self):
        if self.__gauche:
            self.__gauche.montrer_ordre_croissant()
        print(self.__num)
        if self.__droite:
            self.__droite.montrer_ordre_croissant()

    # ... méthode montrer_ordre_decroissant() ...

    def __liste_ordre_croissant(self, liste):
        if self.__gauche:
            self.__gauche.__liste_ordre_croissant(liste)
        liste.append(self.__num)
        if self.__droite:
            self.__droite.__liste_ordre_croissant(liste)

    def liste_croissante(self):
        liste = []
        self.__liste_ordre_croissant(liste)
        return liste

    # ... méthode __liste_ordre_decroissant() ...

    # ... méthode liste_decroissante() ...
```

4.2 Une classe pour la structure d'arbre binaire de recherche

arbre.py

```
from string import *
import random

from noeud import *

class arbre(object):
    """ arbre binaire de recherche """

    def __init__(self):
        """ constructeur """
        self.__racine = None

    def montrer(self):
```

```
        """ imprimer l'arbre """
        if self.__racine:
            self.__racine.montrer()
        else:
            print("(arbre vide)")

    def chercher(self, num):
        """ rechercher une valeur """
        if self.__racine:
            return self.__racine.chercher(num)
        else:
            return False

    def inserer(self, num):
        """ insérer une nouvelle valeur """
        if self.__racine:
            self.__racine.inserer(num)
        else:
            self.__racine = noeud(num)

    def supprimer(self, num):
        """ supprimer une valeur """
        if self.chercher(num):
            if self.__racine.get_num() == num:
                if self.__racine.fils_gauche():
                    fils = self.__racine.fils_gauche()
                else:
                    fils = self.__racine.fils_droite()
                if fils is None:
                    self.__racine = None
                else:
                    if self.__racine.fils_gauche():
                        x = fils.plus_grand()
                    else:
                        x = fils.plus_petit()
                    self.__racine.supprimer(x)
                    self.__racine.set_num(x)
            else:
                self.__racine.supprimer(num)
        else:
            print("(valeur inexistante)")

    def montrer_ordre_croissant(self):
        """ montrer la liste en ordre croissant """
        if self.__racine:
            self.__racine.montrer_ordre_croissant()
        else:
            print("(liste vide)")

    # ... méthode montrer_ordre_decroissant() ...
```

```
    def liste_croissante(self):
        """ liste en ordre croissant """
        if self.__racine:
            return self.__racine.liste_croissante()
        else:
            return []

    # ... méthode liste_decroissante() ...

    def noeuds(self):
        """ nombre de noeuds de l'arbre """
        if self.__racine:
            return self.__racine.noeuds()
        else:
            return 0
```

Le programme principal suivant permettra de tester l'adjonction et la suppression d'éléments dans un arbre binaire de recherche :

arbre.py

```
if __name__ == "__main__":
    a = arbre()
    for n in range(10):
        a.inserer(random.randrange(100))

    a.montrer()

    print("(" + str(a.noeuds()) + " noeud(s) dans l'arbre)")

    fin = False
    while fin == False:
        print("**************************************")
        print("(a)jouter ou (c)roissant ou (d)écroissant")
        print("ou (s)upprimer ou (q)uitter ?")
        ch = str(input())
        if ch == 'a' or ch == 's':
            n = int(input())
            print "--------------------------------------"
            if ch == 'a':
                a.inserer(n)
            if ch == 's':
                a.supprimer(n)
            a.montrer()
        if ch == 'c':
            print(a.liste_croissante())
        if ch == 'd':
            print(a.liste_decroissante())
        if ch == 'q':
            fin = True
        if not (ch in "acdsq"):
```

```
            print("touches a, c, d, s, q seulement !")
```

5 Calculateur

5.1 Des choix de conception : composition, héritage, polymorphisme

Ces notions sont capitales car elles déterminent les structures des données du programme. Ces données dépendent généralement les unes des autres dans la résolution d'un problème. Cela introduit un couplage de l'information, avec une conséquence importante : tout changement dans la définition d'un type ou d'une structure de donnée peut entraîner par effet de bord des dysfonctionnements importants pouvant compromettre les structures qui en dépendent.

La conception de classes pour la traduction d'algorithme nécessite donc une définition soigneuse de hiérarchies d'objets. On rappelle brièvement les notions de composition, héritage et polymorphisme :

- la composition (ou l'agrégation) d'objets est une relation entre objets du type « possède un » ;
- l'héritage entre objets est une relation du type « est un ». L'héritage permet le polymorphisme, à savoir la manipulation d'un objet sous diverses identités.

Il est souvent délicat de déterminer quelle est la plus judicieuse des deux possibilités pour définir un nouveau type d'objet. Il est important d'avoir à l'esprit que l'héritage maintient des dépendances fortes au sein d'une hiérarchie : un changement à un niveau possède des conséquences sur toute la chaîne des objets, alors que la composition maintient une certaine séparation des rôles et donc favorise l'indépendance des types. Cet aspect doit être pris en compte dans la réalisation de programmes importants.

Pour fixer les idées, nous prendrons l'exemple d'une hiérarchie de quadrilatères, avec les classes suivantes :

```
class quadrilatere(object):
    """ quadrilatère quelconque """

    def __init__(self, a =0, b =0, c =0 , d=0):
        self.a = a
        self.b = b
        self.c = c
        self.d = d
        self.type = "quadrilatere"

    def __str__(self):
        return self.type

    def perimetre(self):
        return (self.a + self.b + self.c + self.d)

if __name__ == "__main__":
    q = quadrilatere(5, 7, 10, 3.4)
```

```
    print(q)
    print(q.perimetre())
```

La classe `quadrilatere` est notre classe mère, à partir de laquelle nous construisons les classes filles `losange` et `rectangle`. Nous parlons d'héritage simple : chacune des deux classes filles possède une classe mère unique.

```
from quadrilatere import *

class losange(quadrilatere):
    """ losange """

    def __init__(self, a =0):
        quadrilatere.__init__(self, a, a, a, a)
        self.type = "losange"

    def perimetre(self):
        return self.a*4

if __name__ == "__main__":
    l = losange(5)
    print(l)
    print(l.perimetre())
```

Le constructeur de la classe `losange` fait appel au constructeur de la classe mère. Nous pouvons noter la réécriture de la méthode `perimetre()` : cette dernière est spécialisée, elle se substitue à la définition de la méthode de même définie dans la classe mère.

```
from quadrilatere import *

class rectangle(quadrilatere):
    """ rectangle """

    def __init__(self, a =0, b =0):
        quadrilatere.__init__(self, a, b, a, b)
        self.type = "rectangle"

#   def perimetre(self):
#       return (self.a + self.b)*2

if __name__ == "__main__":
    r = rectangle(4, 3)
    print(r)
    print(r.perimetre())
```

Le lecteur aura remarqué les lignes commentées pour la méthode de calcul du périmètre : le périmètre est calculé en invoquant, soit une méthode spécialisée, soit une méthode présente plus haut dans la hiérarchie de classe.

Nous construisons maintenant une classe héritant à la fois de la classe `losange` et de la classe `rectangle`, la classe `carre` :

```
from losange import *
from rectangle import *

#class carre(losange, rectangle):
class carre(rectangle, losange):
    """ carré """

    def __init__(self, a =0):
#       quadrilatere.__init__(self, a, a, a, a) # fonctionne aussi
        losange.__init__(self, a)

if __name__ == "__main__":
    c = carre(10)
    print(c)
    print(c.perimetre())
```

On remarquera ici les deux possibilités d'écriture du constructeur : en faisant appel à une méthode de la classe `quadrilatere` ou de la classe `losange`.

D'autre part, nous avons ici typiquement un cas d'héritage « en diamant », du fait de la déclaration de la classe qui mentionne les deux classes `losange` et `rectangle` : quelle est la variable d'instance retournée lorsque nous invoquons la méthode `carre.__str__()` ? Pour s'en convaincre, il suffit de commenter/décommenter les deux lignes commençant par le mot-clé `class`.

L'intérêt pour un objet d'hériter d'un autre objet est de bénéficier de ses informations et méthodes, en ayant la possibilité notamment de les compléter ou de les altérer.

Un objet peut hériter des informations ou des méthodes de plusieurs objets, on parle alors d'héritage multiple ou polymorphisme.

Le constructeur pour instancier la classe fille se contente d'invoquer le constructeur de la classe mère. Il n'est pas nécessaire d'initialiser explicitement les champs de l'objet, qui sont automatiquement appelés lors de l'appel du constructeur. On peut bien évidemment adjoindre de nouveaux champs pour l'objet fille, ou même, sous certaines précautions, altérer les champs hérités de la classe mère.

Remarque. En programmation orientée objet, on accorde un soin particulier au caractère public ou privé de l'information. Ainsi les deux champs `self.__donnee` et `self.donnee` ne sont pas du tout interprétés de la même façon par **Python**.
Dans le premier cas, le champ est privé (c'est-à-dire inaccessible directement depuis l'extérieur de la classe), dans le second cas le champ est public, visible depuis l'extérieur de la classe, en consultation comme en modification. Il revient au programmeur de décider de ce qui doit être public, en privilégiant le caractère privé par défaut, afin d'éviter d'altérer l'objet depuis de nombreux points du programme.

5.2 Le projet calculateur

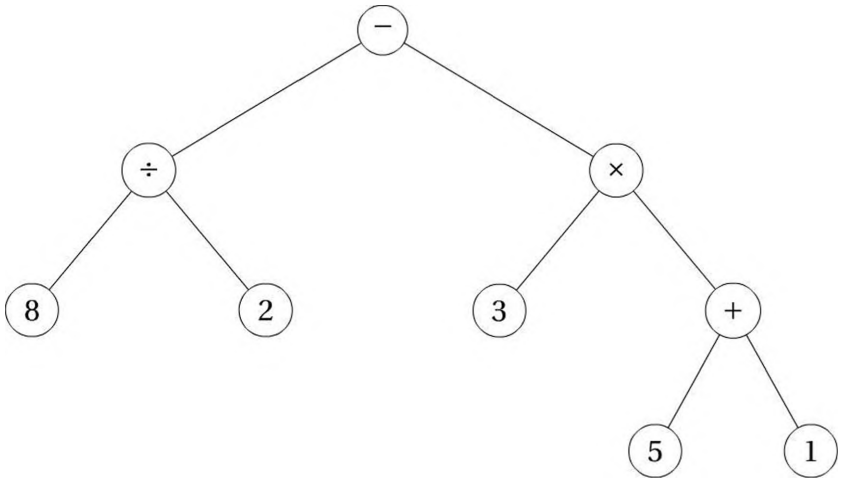
Nous ferons ici la synthèse des thèmes étudiés depuis le début de cette partie, sous la forme d'un petit projet de calculateur sur les nombres rationnels.

Pour simplifier, nous nous baserons sur une version légère de notre calculateur dans laquelle les nombres et les opérations s'écrivent tous sous la forme d'un unique caractère ASCII. Nous faisons cette réduction avec l'intention de faciliter le découpage en unités lexicales (unités insécables mathématiquement pertinentes) de l'expression mathématique à évaluer. L'hypothèse de considérer des lexèmes constitués de plusieurs caractères relève de l'analyse lexicale, domaine non étudié ici.

Nous allons considérer une expression A :

$$A = 8 \div 2 - 3 \times (5 + 1)$$

que nous allons transformer en un arbre pour son évaluation :



L'arbre binaire est parfaitement adapté à la représentation de la structure opératoire de l'expression ; en particulier, il est conforme aux règles usuelles de priorité entre opérations.

Si l'on souhaite évaluer mécaniquement l'expression A, deux questions se posent :

- Comment crée-t-on l'arbre binaire à partir de la forme traditionnelle de l'expression A ?
- Comment évalue-t-on mécaniquement l'expression A à l'aide de l'arbre binaire ?

Nous utiliserons les classes et structures de données rencontrées précédemment :

- les classes numériques (entiers, rationnels) ;
- une structure de queue pour un parcours horizontal de l'expression ;
- une structure de pile pour les opérateurs, puis une structure de pile pour l'évaluation de l'expression ;
- une structure d'arbre binaire.

Nous mettrons en œuvre deux stratégies pour l'évaluation : une approche par les piles d'opérateurs, puis une approche basée sur la récursivité des grammaires, utilisée dans la construction des logiciels interprètes (**Python** en est un bon exemple) ou compilateurs (le langage C est un langage en général compilé, mais on peut en trouver quelques interprètes).

5.3 Notations infixée, préfixée et postfixée

On peut donner trois représentations de l'expression A :

- la forme préfixée [− ; ÷ ; 8 ; 2 ; × ; 3 ; + ; 5 ; 1]. Elle correspond littéralement à « soustraire du quotient de 8 par 2 le produit de 3 par la somme de 5 et 1 ». On indique au calculateur quelle opération il doit effectuer avec quelles opérandes ;
- la forme infixée $A = 8 \div 2 - 3 \times (5 + 1)$, l'écriture mathématique usuelle ;
- la forme postfixée [8 ; 2 ; ÷ ; 3 ; 5 ; 1 ; + ; × ; −]. Cette forme est la forme privilégiée par la plupart des systèmes de calcul, qui évaluent une expression en faisant appel à une pile d'évaluation. Une célèbre gamme de calculatrices en notation polonaise inverse (RPN ou *Reverse Polish Notation*) en est l'exemple bien connu.

On observera que, le nombre d'opérandes étant parfaitement déterminé pour un opérateur donné, les parenthèses deviennent totalement inutiles en notations préfixées et postfixées. La présence de parenthèses est le signe du caractère récursif de notre manière d'écrire les expressions algébriques.

Le diagramme suivant illustre l'évolution de l'état de la pile de calcul au fur et à mesure de la lecture de l'expression postfixe :

unité lex.	8	2	÷	3	5	1	+	×	−
pile	8	2 8	**4**	3 4	5 3 4	1 5 3 4	**6** 3 4	**18** 4	**−14**

Chacun des nombres en caractères gras est obtenu par l'application de l'opérateur arithmétique lu sur les deux opérandes dépilées (pour l'opérateur, l'ordre des opérandes est l'ordre inverse du dépilement) ; le résultat est alors empilé.

Lorsque l'évaluation s'est déroulée correctement, la pile contient à terme un seul nombre, qui est la valeur de l'expression.

5.3.1 Une première version du calculateur

Le programme `calc1.py` est constitué de sept fichiers présentés ci-après (à l'exception des fichiers `entier.py` et `rationnel.py` présentés précédemment et non reproduits ici).

Cette version du calculateur est construite sur le mode d'évaluation décrit plus haut, dans laquelle l'expression est transformée en arbre binaire. Nous avons en fait simplifié notre problème en considérant que toutes nos opérations sont des opérateurs binaires. Ce choix entraîne une difficulté pour la notation de nombre négatif ou de nombre opposé.

Ce problème est contourné par une transformation d'écriture lors de la phase d'analyse lexicale de l'expression. Pour cela, nous utilisons la propriété : « $-a = (-1) \times a$ ». Ainsi, l'expression « $-(x+5)+7$ » sera transformée en la liste d'unités lexicales « $[(-1);\times;(;x;+;5;);+;7;]$ » dans laquelle tous les opérateurs sont binaires.

analex.py

```
from os import *
from re import *
from string import *

def analyse_lexicale(s):
    s = s.replace('[', '(')
    s = s.replace(']', ')')
    s = s.replace('{', '(')
    s = s.replace('}', ')')
    s = s.replace('/', ':')
    t = s.split()
    s = ''.join(t)
    s = '{' + s + '}'
    s = s.replace('-', '+-1*')
    s = s.replace('{+', '{')
    s = s.replace('(+', '(')

    separateurs = compile(r'([+\-*:^(){}])')
    jetons = separateurs.split(s)

    while jetons.count('') > 0:
        jetons.remove('')
    l = []
    for t in jetons:
        if len(t) >= 1:
            l.append(t)
        else:
            if t.isdigit():
                l.append(t)
            else:
                n = len(t)
                for i in range(n):
                    if l[-1] in ascii_letters:
                        l.append('*')
                    l.append(t[i])

    r = []
    r.append(l[0])
    moins = False
    for t in l[1:]:
        if t == '-':
            moins = True
            continue
        if t == '1' and moins == True:
            moins = False
            t = "-1"
        r.append(t)
```

```
    del r[0]
    del r[-1]

    return r
```

La procédure `analyse_lexicale()` effectue le découpage de l'expression en une suite de lexèmes, en vue de la constitution de l'arbre binaire de l'expression. Au passage certains caractères sont remplacés de façon équivalente par d'autres.

Comme annoncé, notre modèle de nœud pour un arbre binaire contient, en plus de l'information utile, deux pointeurs vers deux sous-arbres (fils gauche et fils droit).

noeud2.py

```
from string import *
from rationnel import *

class noeud2(object):
    """ noeud double (pour un arbre binaire) """

    def __init__(self, donnee =None, gauche =None, droite =None):
        self.__donnee = donnee
        self.__gauche = gauche
        self.__droite = droite

    def en_chaine_prefixe(self):
        if self.__donnee[-1] in digits:
            return str(self.__donnee)
        if isinstance(self.__gauche, rationnel):
            operande_gauche = str(self.__gauche)
        else:
            operande_gauche = self.__gauche.en_chaine_prefixe()
        if isinstance(self.__droite, rationnel):
            operande_droite = str(self.__droite)
        else:
            operande_droite = self.__droite.en_chaine_prefixe()
        op = self.__donnee
        return str(op) + '|' + str(operande_gauche) + '|' + str(operande_droite)

    # ... méthode en_chaine_infixe() ...

    # ... méthode en_chaine_postfixe() ...

    def donnee(self):
        return self.__donnee
```

On pourra apprécier la puissance d'expressivité de la récursivité dans l'écriture des procédures `en_chaine_infixe()` et `en_chaine_postfixe()`. La différence fondamentale entre les deux méthodes de parcours se situe dans l'inspection récursive du nœud : en parcours infixe, on inspectera le fils gauche, puis la donnée, puis le fils droit. En parcours postfixe, on inspectera le fils gauche, puis le fils droit, puis la donnée.

arbre.py

```
from string import *

from analex import *
from noeud1 import *
from noeud2 import *
from rationnel import *

def simplifier_parentheses(s):
    if s.startswith('(') and s.endswith(')'):
        s = s[1:-1]
    return s

class arbre(object):
    """ arbre binaire de l'expression """

    def __init__(self, expression_infixe):
        self.__operateurs = []
        self.__pile = []
        e = analyse_lexicale(expression_infixe)
        for jeton in e:
            if jeton[-1] in digits:
                self.__pile.append(noeud2(jeton))
                continue
            if jeton == '(':
                self.__operateurs.append(jeton)
                continue
            if jeton == ')':
                while len(self.__operateurs) > 0 \
                    and self.__operateurs[-1] != '(':
                    self.nouvelle_operation()
                self.__operateurs.pop(-1)
                continue
            if jeton in "+-*:/^":
                if jeton != '^':
                    while len(self.__operateurs) > 0 \
                        and self.priorite(self.__operateurs[-1]) >=
                            self.priorite(jeton):
                        self.nouvelle_operation()
                self.__operateurs.append(jeton)
                continue
        while len(self.__operateurs) > 0:
            self.nouvelle_operation()
        if len(self.__pile) == 1:
            self.__racine = self.__pile.pop(-1)

    def nouvelle_operation(self):
        sommet = self.__operateurs.pop(-1)
        droite = self.__pile.pop(-1)
```

```
        gauche = self.__pile.pop(-1)
        noeud = noeud2(sommet, gauche, droite)
        self.__pile.append(noeud)

    def priorite(self, op):
        precedence = 0
        if op == '+' or op == '-':
            precedence = 1
        if op == '*' or op == ':' or op == '/':
            precedence = 2
        if op == '^':
            precedence = 3
        return precedence

    def prefixe(self):
        return self.__racine.en_chaine_prefixe()

    def infixe(self):
        return simplifier_parentheses(self.__racine.en_chaine_infixe())

    def postfixe(self):
        return self.__racine.en_chaine_postfixe()

    def evaluer(self):
        expr = self.__racine.en_chaine_infixe()
        val = []
        p = self.__racine.en_chaine_postfixe()
        q = p.split('|')
        for n in q:
            if n[-1] in digits:
                val.append(rationnel(int(n)))
                continue
            if n == '+':
                b = val.pop(-1)
                a = val.pop(-1)
                r = a + b
                val.append(r)
                continue
            if n == '-':
                b = val.pop(-1)
                a = val.pop(-1)
                r = a - b
                val.append(r)
                continue
            if n == '*':
                b = val.pop(-1)
                a = val.pop(-1)
                r = a * b
                val.append(r)
                continue
```

```
            if n == ':' or n == '/':
                b = val.pop(-1)
                a = val.pop(-1)
                r = a / b
                val.append(r)
                continue
            if n == '^':
                b = val.pop(-1)
                a = val.pop(-1)
                r = a ** b
                val.append(r)
                continue
        return val.pop(-1)
```

Pour la classe `arbre`, le constructeur est responsable de la construction de l'arbre binaire de l'expression ; la méthode `arbre.evaluer()` est en charge de l'évaluation de l'arbre à l'aide d'une pile.

calcul.py

```
from rationnel import *

class calcul(object):
    """ évaluation expression postfixée """

    def __init__(self):
        self.__pile = []

    def empiler(self, arg):
        self.__pile.append(arg)

    def depiler(self):
        if len(self.__pile) > 0:
            return self.__pile.pop(-1)
        else:
            return rationnel(0, 1, False)

    def additionner(self):
        b = self.depiler()
        a = self.depiler()
        r = a + b
        self.empiler(r)

    # ... méthode soustraire() ...

    # ... méthode multiplier() ...

    # ... méthode diviser() ...

    # ... méthode elever_puissance() ...
```

```
    def resultat(self):
        return self.depiler()

    def reponse(self):
        return self.resultat()
```

Dans le code des opérations arithmétiques présenté ci-dessus, on relèvera l'absence de tout code spécifique à la nature des opérandes. Ce contrôle, grâce à la définition de nos objets, est effectué au sein de chaque opérande. Nous avons ici encore un exemple typique d'abstraction, notre code fonctionne également sans modification sur n'importe quel type natif supportant ces opérateurs arithmétiques.

calc1.py

```
from arbre import *

def calculer(expression):
    e = arbre(expression)
    print("- forme initiale ........ :", expression)
    print("- expression préfixe .... :", e.prefixe())
    print("- expression infixe ..... :", e.infixe())
    print("- expression postfixe ... :", e.postfixe())
    print("- valeur obtenue ........ :", e.evaluer())
    print()

def tests_automatiques():
    # donne 3 4 + 5 1 1 + ^ * 7 - = 168
    print("Exemple de calcul :")
    calculer("(3 + 4) * 5 ^ (1 + 1) - 7")

    # donne 2 3 2 2 ^ ^ ^ = 24178516392292583494123 52
    print("Exemple de calcul :")
    calculer("2 ^ 3 ^ 2 ^ 2")

    # donne 9 1 + 7 2 5 * + *
    print("Exemple de calcul :")
    calculer(" ( 9 + 1 ) * ( 7 + 2 * 5 ) ")

    # donne 3 7 : 2 7 : 5 14 : : -
    print("Exemple de calcul :")
    calculer(" 3/7 - 2/7 : ( 5 : 14 ) ")

def lecture_expr():
    print("---")
    print("Calcul suivant (laisser vide et valider pour quitter):")
    return input()

def boucle():
    expr = lecture_expr()
    while len(expr) > 0:
        calculer(expr)
```

```
        expr = lecture_expr()

if __name__ == "__main__":
    tests_automatiques()
    boucle()
```

5.4 Grammaire et notation BNF

Nous utiliserons cette fois une grammaire, sous la forme BNF (notation BACKUS-NAUR FORM) étendue, pour représenter nos expressions numériques :

```
expr ::= expr1 "+" expr1 |
         expr1 "-" expr1 |
         expr1

expr1 ::= expr2 "*" expr2 |
          expr2 "/" expr2 |
          expr2

expr2 ::= "-" expr3 |
          expr3

expr3 ::= expr4 "^" expr2 |
          expr4

expr4 ::= <naturel> |
          "(" expr ")"

naturel ::= ('0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9')*
```

Le terminal `naturel` est construit sur la répétition éventuelle d'un chiffre décimal.

Cette grammaire fonctionne en quelque sorte comme un véritable pilote pour la lecture de l'expression à évaluer.

Cette grammaire est volontairement non récursive à gauche : son intérêt est une traduction immédiate en termes de procédures dans un langage de programmation.

Nous aurions pu choisir une grammaire récursive *à gauche* décrivant le même langage, avec :

```
expr ::= expr "+" terme |
         expr "-" terme |
         terme

terme ::= terme "*" unaire |
          terme "/" unaire |
          unaire

(...)
```

Malheureusement, la traduction d'une telle grammaire en programme impose d'écrire une procédure `expr` qui commence par s'invoquer elle-même. On entre alors dans une boucle infinie : le programme ne peut pas fonctionner.

5.4.1 Une seconde version du calculateur

Les fichiers `entier.py` et `rationnel.py` sont ceux présentés précédemment et ne sont pas reproduits ici. Le programme se décompose cette en seulement deux fichiers (seule l'évaluation est effectuée, sans aucune présentation de l'arbre de l'expression).

Dans cette version, nous avons choisi de créer un objet `expression` qui évalue une expression selon la grammaire présentée plus haut. Chaque non-terminal de la grammaire donne donc lieu à une méthode pour son évaluation.

expression.py

```
from rationnel import *

class expression(object):
    """ expression à évaluer """

    def __init__(self, s):
        self.__source = s
        self.__n = len(s)
        self.__i = 0
        self.__err = 0
        self.__val = 0

    def __str__(self):
        s = ""
        for ch in self.__source:
            if not str.isspace(ch):
                s += ch
        return s

    def erreur(self, e):
        if self.__err == 0:
            self.__err = e

    def erreur_existe(self):
        if self.__err == 0:
            return False
        else:
            return True

    def aucune_erreur(self):
        return not self.erreur_existe()

    def valeur(self):
        self.evaluer()
        return self.__val
```

```
    def evaluer(self):
        self.__ch = self.suivant()
        if self.__ch == '{':
            self.__ch = self.suivant()
            self.__val = self.expr()
            self.__ch = self.suivant()
            if self.__ch != '}':
                self.erreur(2) # manque une "}"
        else:
            self.erreur(1) # manque une "{"

    def suivant(self):
        while self.__i < self.__n:
            t = self.__source[self.__i]
            self.__i += 1
            if str.isspace(t):
                continue
            else:
                return t
        return '\0'

    def prochain_lu(self):
        while self.__i < self.__n:
            t = self.__source[self.__i]
            if str.isspace(t):
                self.__i += 1
            else:
                return t
        return '\0'

    def suivant_est(self, t):
        if self.prochain_lu() == t:
            return True
        else:
            return False

# expr ::= expr1 '+' expr1 | expr1 '-' expr1 | expr1
    def expr(self):
        if self.erreur_existe():
            return 0
        t = self.expr1()
        while self.suivant_est('+') or self.suivant_est('-'):
            self.__ch = self.suivant()
            if self.__ch == '+':
                self.__ch = self.suivant()
                t += self.expr1()
            else:
                if self.__ch == '-':
                    self.__ch = self.suivant()
```

```
                    t -= self.expr1()
        return t

# expr1 ::= expr2 '*' expr2 | expr2 '/' expr2 | expr2
    def expr1(self):
        if self.erreur_existe():
            return 0
        t = self.expr2()
        while self.suivant_est('*') or self.suivant_est('/') or
            self.suivant_est(':'):
            self.__ch = self.suivant()
            if self.__ch == '*':
                self.__ch = self.suivant()
                t *= self.expr2()
            else:
                if self.__ch == '/' or self.__ch == ':':
                    self.__ch = self.suivant()
                    t /= self.expr2()
                    if not t.est_valide():
                        self.erreur(7) # tentative de diviser par 0
        return t

# expr2 ::= '-' expr3 | expr3
    def expr2(self):
        if self.erreur_existe():
            return 0
        negate = False
        while self.__ch == '-':
            negate = not negate
            self.__ch = self.suivant()
        t = self.expr3()
        if negate:
            return -t
        else:
            return t

# expr3 ::= expr4 '^' expr2 | expr4
    def expr3(self):
        if self.erreur_existe():
            return 0
        t = self.expr4()
        if self.suivant_est('^'):
            self.__ch = self.suivant()
            self.__ch = self.suivant()
            k = self.expr2()
            if not k.est_valide():
                self.erreur(8) # exposant invalide
            t = t ** k
        return t
```

```
# expr4 ::= <naturel> | '(' expr ')'
    def expr4(self):
        if self.erreur_existe():
            return 0
        if str.isdigit(self.__ch):
            t = self.naturel()
            return rationnel(int(t))
        if self.__ch == '(':
            self.__ch = self.suivant()
            t = self.expr()
            self.__ch = self.suivant()
            if self.__ch == ')':
                return t
            else:
                self.erreur(10) # manque une ")"
        else:
            self.erreur(9) # manque une "("
        return 0

# naturel ::= ('0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9')*
    def naturel(self):
        n = int(self.__ch)
        x = self.prochain_lu()
        while str.isdigit(x):
            n = n * 10 + int(x)
            self.__ch = self.suivant()
            x = self.prochain_lu()
        return n
```

Le programme `calc2.py`

Ce programme est pratiquement identique au programme `calc1.py`, avec une différence notable dans la manière d'invoquer l'évaluation de l'expression. Il suffit ici de créer une créer une instance d'un objet `expression`, ce qui entraîne dès la construction son évaluation.

calc2.py

```
from expression import *

def tests_automatiques():
    e = expression("{ (3 + 4) * 5 ^ (1 + 1) - 7 }")
    print("Exemple de calcul :", e)
    print(e.valeur(), '\n') # donne 19

    e = expression("{ 2 ^ 3 ^ 2 ^ 2 }")
    print("Exemple de calcul :", e)
    print(e.valeur(), '\n') # donne 2417851639229258349412352

    e = expression("{ ( 9 + 1 ) * ( 7 + 2 * 5 ) }")
    print("Exemple de calcul :", e)
    print(e.valeur(), '\n') # donne 170
```

```
    e = expression("{ 3/7 - 2/7 : ( 5 : 14 ) }")
    print("Exemple de calcul :", e)
    print(e.valeur(), '\n') # donne -13/35

def lecture_expr():
    print("---")
    print("Calcul suivant (laisser vide et valider pour quitter):")
    return input()

def boucle():
    expr = lecture_expr()
    while len(expr) > 0:
        e = expression("{" + expr + "}")
        print(e.valeur())
        print()
        expr = lecture_expr()

if __name__ == "__main__":
    tests_automatiques()
    boucle()
```

6 Polynômes et fractions rationnelles

6.1 Présentation du projet calc3

On aborde les polynômes et les fractions rationnelles sous l'angle du calcul formel : notre but est de construire un petit calculateur formel, devant être en mesure d'évaluer par exemple des expressions telles que $\frac{1}{3}\left(\frac{1}{2}+\frac{1}{3}+\frac{1}{4}\right)$ ou $(x+y)^3-(x-y)^3$.

Nous avons choisi d'écarter d'emblée les nombres décimaux (plus exactement les nombres réels représentables en machine, à savoir du type `float`). Dans la reconnaissance d'une forme à évaluer, notre calculateur n'admet comme nombres que les nombres entiers relatifs en écriture usuelle (hormis d'éventuels chiffres 0 à gauche). En revanche, le calculateur utilisera la forme « à virgule » pour exprimer un nombre décimal — comme le veut la typographie française — le cas échéant.

D'autre part, toutes les opérations doivent être explicitement écrites (à la différence de la syntaxe usuelle de **Python**, nous avons retenu le symbole d'accent circonflexe seul ∧ pour l'élévation en puissance).

Développons par exemple la fraction :

$$A=\left(\frac{\frac{2}{3}x+\frac{1}{7}}{\frac{4}{15}y-\frac{1}{24}}\right)^4$$

Le programme « `calc3` » donne comme réponse (au format près de la sortie [2]) :

$$A = \frac{98344960000x^4 + 84295680000x^3 + 27095040000x^2 + 3870720000x + 207360000}{2517630976y^4 - 1573519360y^3 + 368793600y^2 - 38416000y + 1500625}$$

À titre de comparaison, voici la réponse proposée par le logiciel « `maxima` » [3] :

$$A = \frac{207360000}{2517630976y^4 - 1573519360y^3 + 368793600y^2 - 38416000y + 1500625} + \frac{552960000x}{359661568y^4 - 224788480y^3 + 52684800y^2 - 5488000y + 214375} + \frac{552960000x^2}{51380224y^4 - 32112640y^3 + 7526400y^2 - 784000y + 30625} + \frac{245760000x^3}{7340032y^4 - 4587520y^3 + 1075200y^2 - 112000y + 4375} + \frac{40960000x^4}{1048576y^4 - 655360y^3 + 153600y^2 - 16000y + 625}$$

6.2 Exemples d'utilisation de calc3

6.2.1 Utilisation directe comme programme depuis la console

Nous avons ci-dessous reproduit en partie un exemple d'utilisation du programme `calc3.py` depuis une console :

Terminal

```
$ ./calc3.py
calc3: 1> 1/3*(1/2 + 1/3 + 1/4)

13/36

calc3: 2> (x+y)^3 - (x - y)^3

6 * x^2*y + 2 * y^3

calc3: 3>
$
```

6.2.2 Utilisation sous forme de module depuis un programme

Comme la souplesse du langage **Python** nous le permet, nous pouvons utiliser directement (à l'importation près) notre module `expression.py` présent dans le programme « `calc3` » pour effectuer divers calculs.

Considérons par exemple l'énoncé suivant :

2. la sortie est celle délivrée par le programme`calc3` dans sa version en **Python 2.7**.
3. `maxima` est un logiciel libre, disponible à l'adresse : `http://maxima.sourceforge.net`.

Exercice 69. Soient $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ deux suites réelles données par les définitions mutuellement récurrentes :

$$\begin{cases} u_0 = 2 \text{ et } v_0 = -3 \\ u_{n+1} = -u_n - v_n, & \forall n \geqslant 0 \\ v_{n+1} = \dfrac{4}{3}u_n + \dfrac{5}{3}v_n, & \forall n \geqslant 0 \end{cases}$$

On pose de plus : $\forall n \in \mathbb{N}, t_n^k = u_n + kv_n$, où k désigne un réel.

a) Montrer que, pour certaines valeurs particulières de k que l'on précisera, la suite correspondante $\left(t_n^k\right)_{n\in\mathbb{N}}$ est une suite géométrique.
b) En déduire une expression indépendante de chacun des termes u_n et v_n.

On laisse en exercice au lecteur le fait de démontrer que la suite $\left(t_n^k\right)_{n\in\mathbb{N}}$ est géométrique pour les deux valeurs $k_1 = \dfrac{1}{2}$ et $k_2 = \dfrac{3}{2}$.

Afin d'illustrer l'utilisation du module « `expression.py` », nous allons voir comment nous pouvons observer le comportement des trois suites. On peut par exemple calculer les dix premiers termes de chacune des trois suites, et afficher les premiers quotients $\dfrac{t_{n+1}}{t_n}$:

Solution.

exemple1.py

```
from modules_calc3 import *

#
# suite u (définie par récurrence)
#
def u_suivant(u, v):
    return (-u - v)

#
# suite v (définie par récurrence)
#
a = ex_t.expression("4/3").lire_valeur()
b = ex_t.expression("5/3").lire_valeur()

def v_suivant(u, v):
    global a
    global b
    return (a * u + b * v)

#
# suite t (combinaison linéaire de u et v)
#
def t(u, v, k):
    return (u + k * v)
```

```
#
# premiers termes
#
u_0 = ex_t.expression("2").lire_valeur()
v_0 = ex_t.expression("-3").lire_valeur()


#
# calcul et impression des dix premiers termes de chaque suite
#
def calculs(k):
    u, v = u_0, v_0
    for n in range(10):
        print("u_" + str(n), "=", u)
        print("v_" + str(n), "=", v)

        d = t(u, v, k)
        print("k_" + str(n), "=", d)
        if n > 0:
            print("quotient =", d/m)
        m = d
        print()

        u, v = u_suivant(u, v), v_suivant(u, v)

#
# calculs
#
print("-----")
k = ex_t.expression("k").lire_valeur()
calculs(k)

print("-----")
k = ex_t.expression("1/2").lire_valeur()
calculs(k)

print("-----")
k = ex_t.expression("3/2").lire_valeur()
calculs(k)
```

On pourra au passage observer la forme générale inattendue prise par la raison de la suite $\left(t_n^k\right)_{n\in\mathbb{N}}$.

Les connaissances d'algèbre linéaire permettent d'examiner la situation du point de vue matriciel. Le traitement du problème fait intervenir la matrice A :

$$A = \begin{pmatrix} -1 & -1 \\ \frac{4}{3} & \frac{5}{3} \end{pmatrix}$$

On montre ci-dessous comment établir avec la librairie `calc3` le polynôme « det (A – λ I) » utile à la diagonalisation de la matrice A. Pour simplifier le code et tenir compte du fait que `calc3` manipule uniquement des entités dont le nom est limité à une lettre unique, nous adopterons la convention $x = \lambda$ dans ce qui suit.

```
from expression import *

#
# la fonction msom() ci-dessous est celle vue au chapitre 3
#
def msom(A,B):
    if lignes(A) != lignes(B) or cols(A) != cols(B):
        return "Tailles non compatibles"
    else:
        return [[A[i][j] + B[i][j] for j in cols(A)] for i in lignes(A)]
#
# impression de la matrice M "à la maxima"
#
def montrer_matrice(M):
    for k in M:
        s = "\t["
        for a in k:
            s += "\t" + str(a)
        print(s + "]")

#
# construction de la matrice A
#
a11 = ex_t.expression("-1").lire_valeur()
a12 = ex_t.expression("-1").lire_valeur()
a21 = ex_t.expression("4/3").lire_valeur()
a22 = ex_t.expression("5/3").lire_valeur()
A = [[a11, a12], [a21, a22]]

montrer_matrice(A)

#
# construction de la matrice B = -x*I
#
x = ex_t.expression("x").lire_valeur()
b11 = -x
b12 = ex_t.expression("0").lire_valeur()
b21 = ex_t.expression("0").lire_valeur()
b22 = -x
B = [[b11, b12], [b21, b22]]

#
# matrice S = (A - x*Id)
#
S = msom(A, B)
```

```
montrer_matrice(S)

#
# déterminant de la matrice S
#
det = S[0][0] * S [1][1] - S[0][1] * S [1][0]
print(det)
```

On observera notamment le fonctionnement et l'utilité d'une bonne abstraction des types de donnée : le respect de cette règle de conception est la garantie que nous avions de pouvoir réutiliser *sans aucune modification* la procédure `msom()` définie auparavant et de façon totalement indépendante au chapitre 3.

Le programme nous donne une vue de la matrice « $A - \lambda I$ » et son déterminant (la sortie du programme est sans fioritures) :

Terminal

```
        [(-1)         (-1)]
        [4/3         5/3]
        [(-1) * x + (-1)         (-1)]
        [4/3         (-1) * x + 5/3]
x^2 + (-2/3) * x + (-1/3)
```

Nous n'irons pas plus loin sur cet exemple car notre propos était de fournir une preuve de fonctionnement du projet « `calc3` ». La boîte à outils du projet ne demande qu'à être complétée par son utilisateur en fonction de ses propres besoins.

Signalons enfin l'existence — pour le calcul numérique et le calcul formel avec **Python** — de nombreux projets, dont les programmes `NumPy` et `SymPy`, conçus pour **Python**, sans oublier le vaste projet `SAGE`.

6.3 Les types et leurs opérations

Dans la définition de notre projet de calculateur, nous faisons l'hypothèse raisonnable que **Python** manipule convenablement les nombres entiers relatifs, y compris les entiers de très grandes valeurs (ce qui n'est hélas pas le cas d'une majorité de langages), sans autres limites que celles imposées par l'environnement d'exécution de **Python** (consulter la documentation de **Python** concernant les types natifs).

Nous donnons dans le tableau suivant les opérations internes selon le type des opérandes :

type	+	−	×	÷	∧
entier rel.	✔	✔	✔	−	exposant dans $\mathbb{N}$
nombre rat.	✔	✔	✔	✔	exposant dans $\mathbb{Z}$
polynôme	✔	✔	✔	−	exposant dans $\mathbb{N}$
fraction rat.	✔	✔	✔	✔	exposant dans $\mathbb{Z}$

Précisons que, pour notre étude, non écartons les exposants non explicites.

Nous traduisons directement ces choix en données structurées sous forme de hiérarchies d'objets.

Pour implémenter ces types, nous utiliserons les structures de données suivantes :

- entier : un objet du type `entier` est basé sur le type natif `long`. Comme les autres types, il comporte un indicateur de validité.
- nombre rationnel : comme on s'en doute, un `rationnel` est une paire de deux `entier`. Lors de sa création, un objet `rationnel` est systématiquement normalisé, c'est-à-dire réduit, avec le dénominateur rendu positif. Les autres modules du programme tiennent pour acquis qu'un `rationnel` est normalisé.
- monôme : un monôme peut être vu comme un nœud dans un arbre binaire de recherche. Plus précisément, un nœud d'arbre binaire représentant un polynôme comporte un membre de type `monome`. Un `monome` est alors l'agrégation d'un coefficient (de type `rationnel`) et d'une chaîne de caractères, développant en extension toutes les indéterminées du monôme.
- polynôme : pour représenter un tel objet, nous avons retenu la structure d'arbre binaire de recherche, et non celle de liste, comme on pourrait y songer à première vue. En effet, nous définissons un « ordre » sur les monômes, d'où l'intérêt d'un arbre binaire de recherche. Ce type de structure facilite en effet considérablement la conservation de l'ordre lors de l'insertion ou de la suppression d'un élément de l'arbre. On peut évidemment obtenir une conservation de l'ordre des monômes sur une liste, mais au prix d'algorithmes plus complexes ou moins efficaces. On notera au passage que nous utilisons, pour la multiplication des polynômes, un parcours de l'arbre en largeur, pour appliquer ensuite la règle usuelle de distributivité.
- fraction rationnelle : un tel objet est constitué de deux polynômes.
- pour la lecture de l'expression, nous utiliserons une structure de queue `LIFO` présentée précédemment. Nous avons privilégié les listes natives de **Python**.
- pour l'évaluation de l'expression, nous avons repris l'implémentation du programme calc2, en la modifiant légèrement afin de calculer d'emblée sur des entiers ou des littéraux, avec cette fois le type `fraction` comme type de base pour les calculs.

Pour ce dernier point, nous avons effectué un nouveau choix d'implémentation. Présentons une première possibilité (les types ci-dessous sont factices, dans le seul but d'illustration) :

Terminal

```
>>> a = polynome("x + 1")
>>> b = fraction("1/x")
>>> c = a + b
>>>
```

Du point de vue de l'utilisateur des classes, nous avons décidé de définir nos objets pour l'utilisation suivante :

Terminal

```
>>> a = expression("x + 1")
>>> b = expression("1/x")
>>> c = a + b
>>>
```

Nous avons choisi d'écarter la première implémentation, afin de conserver des variables a et b déclarées de nature homogène. Au contraire, dans le premier cas, nous aurions eu à gérer les nombreuses situations de conversion de types. Par exemple, pour la seule addition, nous aurions eu à examiner nous-mêmes les situations d'hétérogénéité des arguments (c'est-à-dire en concevoir les méthodes) :

- une méthode d'addition « entier relatif + rationnel » ;
- une méthode « rationnel + entier relatif » (notre addition doit être commutative) ;
- une méthode « entier relatif + polynôme » ;
- une méthode « polynôme + entier relatif » ;
- une méthode « entier relatif + fraction rationnelle » ;
- ...
- sans oublier les opérations internes qui sont codées au sein de chaque type.

Les deux premières lignes ci-dessus illustrent la nécessité de décrire la commutativité de l'addition dans l'écriture des méthodes d'addition des types définis par le concepteur du programme. Par exemple, les appels :

```
addition(entier(2), rationnel(1, 3))
addition(rationnel(1, 3), entier(2))
```

doivent produire la même valeur en retour, alors que les types respectifs des deux arguments sont distincts. Cette approche entraîne un contrôle de type nécessaire pour chaque argument, ainsi qu'une conversion si besoin. L'approche retenue consiste quant à elle à considérer par exemple « 23 » et « x » immédiatement comme deux fractions rationnelles particulières.

Pour terminer la description de nos types de données, signalons que nos types numériques, polynômes et fractions sont en outre munis d'un indicateur d'état de validité, destiné à :

a) vérifier la validité mathématique des opérations (une tentative de division par zéro ; un polynôme est considéré valide si et seulement si chaque monôme est valide...) ;
b) simplifier le flot d'exécution du programme en cas d'erreur dans un calcul (une opération non définie par exemple).

Dans la situation où une instance mathématiquement invalide d'un objet doit être utilisée, nous donnons des valeurs par défaut pour cette instance (par exemple, on choisira l'instance `rationnel(entier(0), entier(1), False)` pour représenter un nombre rationnel

non défini) : de la sorte, un objet représentant une valeur numérique possède toujours une valeur, que cette dernière soit valide ou non.

Cette disposition consistant à accorder la même importance au traitement de l'erreur et aux traitements mathématiques est à rapprocher de l'utilisation du mécanisme d'exception vu dans les chapitres précédents. Nous faisons ici un autre choix, qui accorde une attention égale au traitement des erreurs et des calculs, soit considérer l'erreur mathématique comme une situation banale et pas exceptionnelle en terme de probabilité d'apparition.

Par ailleurs, nous avons parfois choisi d'utiliser nos propres types de données, là où **Python** délivre des types de nature assez proche. Les types natifs doivent être préférés dès qu'ils conviennent parfaitement au problème étudié. S'il est contraire, notre choix est pourtant motivé par la possibilité de transfert vers d'autres langages de programmation dans lesquels certains types ne seraient pas définis au sein du langage (contrairement à beaucoup d'autres, **Python** propose par exemple un type pour la notion de liste).

Signalons enfin qu'une notice accompagne le programme « `calc3` », en présentant les choix retenus pour le développement : elle se trouve dans l'archive des programmes du livre, disponible en ligne sur le site de l'éditeur.

6.4 Pour aller plus loin

Dans le projet `calc3`, nous en sommes restés, en ce qui concerne les exposants, aux nombres entiers relatifs explicites ; ce qui exclut de toute évaluation des constructions telles que $\left(49a^4\right)^{\frac{1}{2}}$ ou $a^n \times a^3$.

En guise de prolongement, on peut faire évoluer l'implémentation de `calc3` de manière à supprimer ces limitations. Il « suffit » pour cela, de considérer que le type de base pour les calculs n'est plus cette fois notre type `fraction` mais un arbre binaire.

L'évaluation de cet arbre peut donner selon le cas un nombre ou une fraction rationnelle au sens large, ou bien en cas d'impossibilité d'expression du résultat par de tels types, l'expression initiale non évaluée. Le projet `calc1` peut constituer un bon point de départ pour cette implémentation d'un type d'objet à vocation calculatoire sous la forme d'arbre binaire.

Si l'on y regarde de plus près, on comprend que nous avons déjà effectué une telle démarche dans l'adaptation du projet `calc2` en `calc3`, par quelques simples modifications dans la traduction de notre grammaire, au niveau du module `expression.py`. On sent alors toute la puissance d'abstraction fournie par les concepts de la programmation orientée objet. Ainsi un bon découpage des objets, et de leurs interactions mutuelles, peut garantir qu'un changement dans l'implémentation d'un objet n'a que peu d'effets de bord sur les autres objets mis en jeu dans le programme.

On pourra également s'intéresser à la dérivation formelle ou au calcul intégral, ainsi qu'au calcul propositionnel, en considérant un type booléen et ses opérations.

Enfin, un autre type de prolongation de ce que nous avons étudié est la réalisation d'un petit langage de programmation. Là encore, on pourra considérer qu'une grammaire étant fixée pour un langage (de programmation), on peut traduire à l'aide des constructions reconnues par ce langage, un programme en un arbre utilisable par un interprète ou un compilateur.

Pour conclure, signalons que le code source complet du projet `calc3` est disponible en ligne sur le site de l'éditeur consacré au livre.

7 Exercices d'entraînement

Les corrigés sont disponibles sur le site dunod.com à partir de la page d'accueil de l'ouvrage.

Exercice 70. On utilise pour cet exercice le type `list` standard de **Python**.
On demande de créer un nouveau type objet `maliste` qui étend les méthodes du type natif `list` avec les fonctions suivantes :

- la méthode `inscrire()` place par défaut un nouvel élément en fin de liste ;
- la méthode `extraire()` extrait par défaut le premier élément de la liste ;
- la méthode `lire()` permet de consulter, sans l'extraire, le premier élément de la liste.

Par défaut le comportement de la classe `maliste` est celui d'une structure de file.

On demande ensuite de créer deux classes filles `mafile` et `mapile` qui héritent de la classe `maliste`, pour la spécialiser respectivement en deux structures de file et de pile.

Exercice 71. On s'intéresse ici à l'évaluation d'une expression mathématique telle que :

$$4 \times (2+3) - 5,$$

préalablement traduite en notation postfixe, c'est-à-dire : « 4 ; 2 ; 3 ; + ; × ; 5 ; − ».

On se limitera pour simplifier aux nombres écrits avec un chiffre décimal unique, et aux quatre opérations, et on supposera que l'expression à évaluer est mathématiquement correcte et bien formée.

En s'appuyant sur les classes `mafile` et `mapile` réalisées dans l'exercice précédent, on demande d'écrire un programme qui évalue l'expression postfixe.

La traduction de l'expression mathématique de la forme usuelle vers la notation postfixe constitue un autre sujet d'étude : pour cela, on pourra par exemple se reporter à l'algorithme « Shunting Yard » (aiguillage ferroviaire) décrit par Edsger DIJKSTRA.

Exercice 72. (Fractions continues – 2e partie) On rappelle la convention adoptée pour représenter les fractions continues sous forme de listes :

$$\frac{111}{40} = 2 + \cfrac{1}{1 + \cfrac{1}{3 + \cfrac{1}{2 + \cfrac{1}{4}}}}$$

c'est-à-dire :

$$\frac{111}{40} = [2\,;1\,;3\,;2\,;4\,]$$

On demande d'écrire un programme qui donne la fraction réduite d'un nombre rationnel désigné par sa représentation sous forme de liste d'entiers.

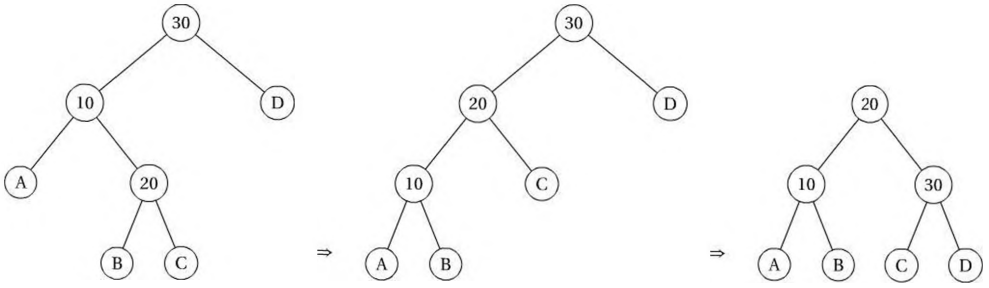


FIGURE 6.15 – Un arbre équilibré en profondeur : une rotation à gauche puis une rotation à droite rééquilibrent l'arbre en hauteur.

Exercice 73. On s'intéresse dans cet exercice à la représentation d'un irrationnel quadratique du type $\sqrt{k}$, k entier naturel non carré parfait, sous la forme d'une fraction continue : on sait dans ce cas que le développement est périodique.

L'algorithme suivant permet de mener à bien le calcul de la période, à partir de trois suites récurrentes $(a_n)_{n\in\mathbb{N}}$, $(d_n)_{n\in\mathbb{N}}$ et $(m_n)_{n\in\mathbb{N}}$ à valeurs dans $\mathbb{N}$, définies par :

$$a_0 = \lfloor k \rfloor\,,\ d_0 = 1\,,\ m_0 = 0$$

puis :

$$m_{n+1} = a_n d_n - m_n\,,\ d_{n+1} = \frac{k - m_{n+1}^2}{d_n}\,,\ a_{n+1} = \left\lfloor \frac{a_0 + m_{n+1}}{d_{n+1}} \right\rfloor$$

Les calculs successifs des triplets (m_n, d_n, a_n) aboutissent à un triplet déjà obtenu, auquel cas la période est atteinte :

$$\sqrt{k} = [a_0, \overline{a_1, a_2, \ldots, a_p}]$$

Par exemple, on a :

$$\sqrt{14} = [3, \overline{1,2,1,6}]$$

(source : « http://en.wikipedia.org/wiki/Methods_of_computing_square_roots »)

On demande de réaliser un programme donnant le développement sous fraction de fraction continue pour un irrationnel quadratique du type $\sqrt{k}$, $k \in \mathbb{N}$.

Exercice 74. On connaît l'existence d'arbres binaires dégénérés, qui perdent leur performance quand il s'agit d'effectuer un parcours. Pour éviter cet inconvénient, on peut utiliser la notion d'arbre équilibré en profondeur, aussi appelé « arbre AVL » (du nom de ses inventeurs Georgii ADELSON-VELSKII et Evguenii LANDIS).

Pour chaque nœud, on calcule la différence de hauteur entre les sous-arbres gauche et droit. Pour insérer un nouveau nœud dans l'arbre, on impose de maintenir une différence des hauteurs valant -1, 0 ou 1, éventuellement au moyen de « rotations » des sous-arbres, comme montré sur la figure 6.15.

On demande de réaliser un programme construisant un arbre AVL.

Bibliographie

[ALSU07] Alfred AHO, Monica LAM, Ravi SETHI et Jeffrey ULLMAN :
Compilateurs : principes, techniques et outils.
Pearson.
2007.

[Amm96] Leendert AMMERAAL :
Algorithmes et structures de données en langage C.
InterÉditions/Masson.
1996.

[Bai08] Yves BAILLY :
Introduction à la programmation avec Python et C++.
Pearson.
2008.

[BM03] Thierry BRUGÈRE et Alain MOLLARD :
Mathématiques à l'usage des informaticiens.
Ellipses.
2003.

[Boo92] Grady BOOCH :
Conception orientée objets et applications.
Addison Wesley.
1992.

[Bop07] Nicole BOPP :
Algorithme cordic pour calculer le logarithme.
Activités mathématiques et scientifiques, N° 62: pp. 31–40, 2007.

[Bri91] Guy BRIAND :
π et la galaxie des arctangentes.
PLOT, N° 54: pp. 23–32, 1991.

[CBP10] Alexandre CASAMAYOU-BOUCAU et François PANTIGNY :
Les maths au collège : démontrer pour comprendre.
Ellipses.
2010.

[CG08] Guillaume CONNAN et Stéphane GROGNET :
Guide du calcul avec les logiciels libres.
Dunod.
2008.

[Con09] Guillaume CONNAN :
Visa pour la prépa MPSI PCSI PTSI.
Dunod.
2009.

[dD14] Florent de DINECHIN :
Page personnelle.
http://perso.citi-lab.fr/fdedinec/.
2014.

[Dem96] Jean-Pierre DEMAILLY :
Analyse numérique et équations différentielles.
EDP Sciences.
1996.

[Dem08] Jean-Pierre DEMAILLY :
Moyenne arithmético-géométrique, intégrales elliptiques et calcul de pi.
Cours, Institut Fourier (CNRS UMR 5582) - Université de Grenoble I.
http://www-fourier.ujf-grenoble.fr/~demailly/manuscripts/pi_ellipt.pdf.
2008.

[Dow08] Gilles DOWEK :
Les principes des langages de programmation.
Éditions de l'École Polytechnique.
2008.

[Dow10] Gilles DOWEK :
Les démonstrations et les algorithmes : introduction à la logique et à la calculabilité.
Informatique. Éd. de l'École polytechnique, Palaiseau.
2010.

[Eng92] Arthur ENGEL :
Mathématiques élémentaires d'un point de vue algorithmique.
Cedic.
1992.

[FC02] Gérard FLEURY et Olivier COGIS :
Graphes à deux voix.
APMEP.
2002.

[Fou06] Laurent FOUSSE :
Intégration numérique avec erreur bornée en précision arbitraire.
PhD in Computer Science, Université Henri Poincaré – Nancy 1.
http://komite.net/laurent/data/pro/publis/these-20070228.pdf.
2006.

[GKP97] Ronald L. GRAHAM, Donald E. KNUTH et Oren PATASHNIK :
Mathématiques Concrètes.
Addison Wesley.
1997.

[Gol91] David GOLDBERG :
What every computer scientist should know about floating point arithmetic.
ACM Computing Surveys, 23(1):5–48, 1991.

[Gou14] Frédéric GOUALARD :
Page personnelle.
http://goualard.frederic.free.fr/.
2014.

[Gra06] Scott GRANNEMAN :
Linux : L'essentiel du code et des commandes.
Campus Press.
2006.

[Gri09] Jean-Philippe GRIVET :
Méthodes numériques appliquées.
EDP Sciences.
2009.

[HW00] Ernst HAIRER et Gerhard WANNER :
L'analyse au fil de l'histoire.
Springer.
2000.

[Kah14] William KAHAN :
Page personnelle.
http://www.cs.berkeley.edu/~wkahan/.
2014.

[KD98] W. KAHAN et Joseph D. DARCY :
How Java's floating-point hurts everyone everywhere.
Technical report, inst-BERKELEY-MATH-EECS, inst-BERKELEY-MATH-EECS :adr.
http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf.
juin 1998.

[Lan09] Hans Peter LANGTANGEN :
A primer on scientific programming with Python.
Springer.
2009.

[LZ04] Vincent LEFÈVRE et Paul ZIMMERMANN :
Arithmétique flottante.
Rapport de Recherche INRIA, N° 5105: pp. 1–63, 2004.
http://hal.inria.fr/inria-00071477/PDF/RR-5105.pdf.

[MBdD+10] Jean-Michel MULLER, Nicolas BRISEBARRE, Florent de DINECHIN, Claude-Pierre JEANNEROD, Vincent LEFÈVRE, Guillaume MELQUIOND, Nathalie REVOL, Damien STEHLÉ et Serge TORRES :
Handbook of Floating-Point Arithmetic.
Birkhäuser Boston.
2010.
ACM G.1.0 ; G.1.2 ; G.4 ; B.2.0 ; B.2.4 ; F.2.1., ISBN 978-0-8176-4704-9.

[Par10] Bernard PARISSE :
Mathématiques assistées par ordinateur.
Cours, Institut Fourier (CNRS UMR 5582) - Université de Grenoble I.
http://www-fourier.ujf-grenoble.fr/~parisse/mat249/mat249.pdf.
2010.

[Pic72] Michèle PICHAT :
Correction d'une somme en arithmétique à virgule flottante.
Numer. Math., 19:400–406, 1972.

[Rod10] Olivier RODOT :
Analyse mathématique, une approche historique.
De Boeck.
2010.

[Ste73] P.H. STERBENZ :
Floating-point computation.
Prentice-Hall series in automatic computation. Prentice-Hall.
http://books.google.fr/books?id=MKpQAAAAMAAJ.
1973.

[Sum08] Mark SUMMERFIELD :
Programming in Python 3 - A Complete Introduction to the Python Language.
Addison Wesley.
2008.

[Swi10] Gérard SWINNEN :
Apprendre à programmer en Python 3.
Eyrolles.
2010.

[Zia09] Tarek ZIADÉ :
Programmation Python : Conception et optimisation.
Eyrolles.
2009.

Index général

Index des commandes