Alexandre Casamayou-Boucau · Pascal Chauvin · Guillaume Connan

Programmation en Python pour les mathématiques

3e édition

DUNOD

Graphisme de couverture : Élisabeth Riba

Illustration de couverture : © Alexvectors - Shutterstock

11 rue Paul Bert, 92240 Malakoff
www.dunod.com
ISBN 978-2-10-083918-6

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 option 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 utiliserons qu'occasionnellement, sans

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.

en faire une présentation détaillée, car cela dépasserait largement le cadre de cet ouvrage. 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. Mais dans les chapitres suivants, nous utiliserons principalement le module `matplotlib` pour produire des graphiques.

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, et simulation de lois de probabilité.

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/fr/3/tutorial/index.html`

Depuis la version 3.7 de **Python**, la documentation est disponible en français à l'adresse :

`http://docs.python.org/fr/3/index.html`

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.
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/`.

```
>>> 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
>>> 2 ** 0.5  # un moyen de calculer la racine carrée de 2
1.4142135623730951
```

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

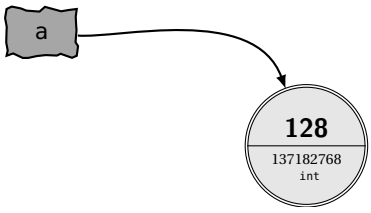
Que se passe-t-il au juste lorsqu'on saisit un nombre (par exemple 128) dans la console **Python** ? Eh bien, disons, en première approximation, que l'interpréteur crée un nouvel « objet » (sans préciser pour l'instant le sens de ce mot) et le garde en mémoire.

```
>>> 128, id(128), type(128)
(128, 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*[3] (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
```



3. La notion de type sera détaillée à la section 11 page 30.

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[4]. 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
```

4. 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.

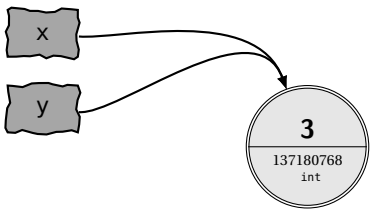
Signalons un raccourci propre à **Python** très utile en pratique et qui a l'avantage d'éviter la confusion avec la manipulation d'équations en algèbre.

```
>>> 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

Parallèlement à la manipulation de variables, un programme informatique nécessite généralement la définition de fonctions. Une fonction au sens informatique reprend la notion mathématique de fonction.
En Python, la définition de la fonction se fait à l'aide du mot `def` suivi du nom de la fonction, des arguments de la fonction entre parenthèses et de deux-points. Elle se poursuit sur les lignes suivantes par les instructions qui doivent être effectuées et se termine par l'instruction `return` suivi de l'expression à renvoyer en fin de programme. À partir de la deuxième ligne, toutes les lignes doivent être indentées, c'est-à-dire qu'elles doivent commencer par quatre espaces.
Voici par exemple une fonction qui prend en argument un nombre et renvoie la racine carrée de ce nombre.