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 978-2-10-073831-1

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

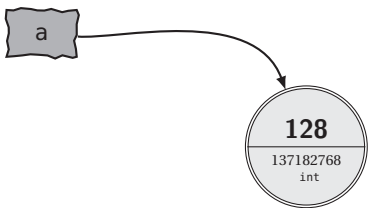
Que se passe-t-il au juste lorsqu'on saisit un nombre (par exemple 128) dans la console **Python** ? Eh bien, disons, en première approximation, que l'interpréteur crée un nouvel « objet » (sans préciser pour l'instant le sens de ce mot) et le garde en mémoire.

```
>>> 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.

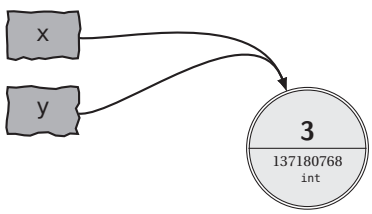
Signalons un raccourci propre à **Python** très utile en pratique et qui a l'avantage d'éviter la confusion avec la manipulation d'équations en algèbre.

```
>>> 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 ====
>>>
3100000000003.1416
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
```