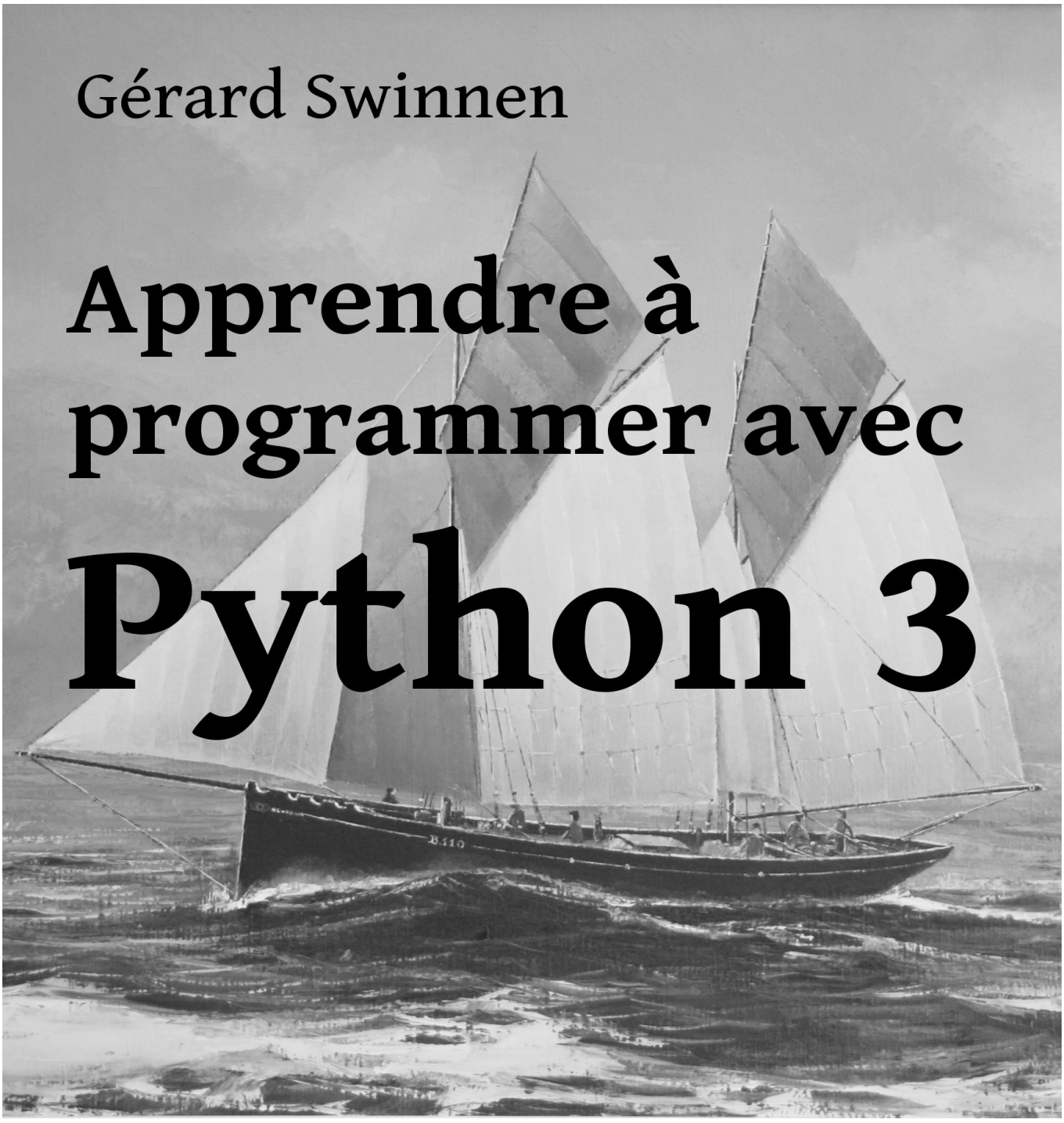
Gérard Swinnen

Apprendre à programmer avec Python 3



La version numérique de ce texte peut être téléchargée librement à partir du site :
http://inforef.be/swi/python.htm

Quelques paragraphes de cet ouvrage ont été adaptés de :
How to think like a coumpter scientist
de *Allen B. Downey, Jeffrey Elkner & Chris Meyers*
disponible sur : http://thinkpython.com
ou : http://www.openbookproject.net/thinkCSpy

La couverture

Choisie délibérément hors propos, l'illustration de couverture est la reproduction d'une œuvre à l'huile réalisée par l'auteur en 1987. Elle met en scène un dundee harenguier de Boulogne, reconstitué d'après des plans anciens et naviguant toutes voiles dehors. Ces bateaux possédaient des mâts articulés pouvant être amenés sur les lieux de pêche, afin de permettre la pêche de dérive.

La capture et la commercialisation du hareng ont été jadis un des principaux piliers de l'économie de l'Europe du Nord.

Grace Hopper, inventeur du compilateur :
« Pour moi, la programmation est plus qu'un art appliqué important. C'est aussi une ambitieuse quête menée dans les tréfonds de la connaissance. »

À Maximilien, Élise, Lucille, Augustin et Alexane.

Préface

En tant que professeur ayant pratiqué l'enseignement de la programmation en parallèle avec d'autres disciplines, je crois pouvoir affirmer qu'il s'agit là d'une forme d'apprentissage extrêmement enrichissante pour la formation intellectuelle d'un jeune, et dont la valeur formative est au moins égale, sinon supérieure, à celle de branches plus classiques telles que le latin.

Excellente idée donc, que celle de proposer cet apprentissage dans certaines filières, y compris de l'enseignement secondaire. Comprenons-nous bien : il ne s'agit pas de former trop précocement de futurs programmeurs professionnels. Nous sommes simplement convaincus que l'apprentissage de la programmation a sa place dans la formation générale des jeunes (ou au moins d'une partie d'entre eux), car c'est une extraordinaire école de logique, de rigueur, et même de courage.

À l'origine, le présent ouvrage a été rédigé à l'intention des élèves qui suivent le cours *Programmation et langages* de l'option *Sciences & informatique* au 3e degré de l'enseignement secondaire belge. Il nous a semblé par la suite que ce cours pouvait également très bien convenir à toute personne n'ayant encore jamais programmé, mais souhaitant s'initier à cette discipline en autodidacte.

Nous y proposons une démarche d'apprentissage non linéaire qui est très certainement critiquable. Nous sommes conscients qu'elle apparaîtra un peu chaotique aux yeux de certains puristes, mais nous l'avons voulue ainsi parce que nous sommes convaincus qu'il existe de nombreuses manières d'apprendre (pas seulement la programmation, d'ailleurs), et qu'il faut accepter d'emblée ce fait établi que des individus différents n'assimilent pas les mêmes concepts dans le même ordre. Nous avons donc cherché avant tout à susciter l'intérêt et à ouvrir un maximum de portes, en nous efforçant tout de même de respecter les principes directeurs suivants :

- L'apprentissage que nous visons se veut généraliste : nous souhaitons mettre en évidence les invariants de la programmation et de l'informatique, sans nous laisser entraîner vers une spécialisation quelconque, ni supposer que le lecteur dispose de capacités intellectuelles hors du commun.
- Les outils utilisés au cours de l'apprentissage doivent être modernes et performants, mais il faut aussi que le lecteur puisse se les procurer en toute légalité à très bas prix pour son usage personnel. Notre texte s'adresse en effet en priorité à des étudiants, et toute notre démarche d'apprentissage vise à leur donner la possibilité de mettre en chantier le plus tôt possible des réalisations personnelles qu'il pourront développer et exploiter à leur guise.
- Nous aborderons très tôt la programmation d'une interface graphique, avant même d'avoir présenté l'ensemble des structures de données disponibles, parce que cette programmation

présente des défis qui apparaissent bien concrets aux yeux d'un programmeur débutant. Nous observons par ailleurs que les jeunes qui arrivent aujourd'hui dans nos classes « baignent » déjà dans une culture informatique à base de fenêtres et autres objets graphiques interactifs. S'ils choisissent d'apprendre la programmation, ils sont forcément impatients de créer par eux-mêmes des applications (peut-être très simples) où l'aspect graphique est déjà bien présent. Nous avons donc choisi cette approche un peu inhabituelle afin de permettre au lecteur de se lancer très tôt dans de petits projets personnels attrayants, par lesquels ils puisse se sentir valorisé. En revanche, nous laisserons délibérément de côté les environnements de programmation sophistiqués qui écrivent automatiquement de nombreuses lignes de code, parce que nous ne voulons pas non plus masquer la complexité sous-jacente.

Certains nous reprocheront que notre démarche n'est pas suffisamment centrée sur l'algorithmique pure et dure. Nous pensons que celle-ci est moins primordiale que par le passé. Il semble en effet que l'apprentissage de la programmation moderne par objets nécessite plutôt une mise en contact aussi précoce que possible de l'apprenant avec des objets et des bibliothèques de classes préexistants. Ainsi il apprend très tôt à raisonner en termes d'interactions entre objets, plutôt qu'en termes de construction de procédures, et cela l'autorise assez vite à tirer profit de concepts avancés, tels que l'instanciation, l'héritage et le polymorphisme.

Nous avons par ailleurs accordé une place assez importante à la manipulation de différents types de structures de données, car nous estimons que c'est la réflexion sur les données qui doit rester la colonne vertébrale de tout développement logiciel.

Choix d'un premier langage de programmation

Il existe un très grand nombre de langages de programmation, chacun avec ses avantages et ses inconvénients. Il faut bien en choisir un. Lorsque nous avons commencé à réfléchir à cette question, durant notre préparation d'un curriculum pour la nouvelle option Sciences & Informatique, nous avions personnellement accumulé une assez longue expérience de la programmation sous *Visual Basic* (*Microsoft*) et sous *Clarion* (*Topspeed*). Nous avions également expérimenté quelque peu sous *Delphi* (*Borland*). Il était donc naturel que nous pensions d'abord exploiter l'un ou l'autre de ces langages. Si nous souhaitions les utiliser comme outils de base pour un apprentissage général de la programmation, ces langages présentaient toutefois deux gros inconvénients :

- Ils sont liés à des environnements de programmation (c'est-à-dire des logiciels) propriétaires.
  Cela signifiait donc, non seulement que l'institution scolaire désireuse de les utiliser devrait acheter une licence de ces logiciels pour chaque poste de travail (ce qui pouvait se révéler coûteux), mais surtout que les élèves souhaitant utiliser leurs compétences de programmation ailleurs qu'à l'école seraient implicitement forcés d'acquérir eux aussi des licences, ce que nous ne pouvions pas accepter. Un autre grave inconvénient de ces produits propriétaires est qu'ils comportent de nombreuses « boîtes noires » dont on ne peut connaître le contenu. Leur documentation est donc incomplète, et leur évolution incertaine.
- Ce sont des langages spécifiquement liés au seul système d'exploitation *Windows*. Ils ne sont pas « portables » sur d'autres systèmes (*Unix*, *Mac OS*, *etc.*). Cela ne cadrait pas avec notre

projet pédagogique qui ambitionne d'inculquer une formation générale (et donc diversifiée) dans laquelle les invariants de l'informatique seraient autant que possible mis en évidence.

Nous avons alors décidé d'examiner l'offre alternative, c'est-à-dire celle qui est proposée gratuitement dans la mouvance de l'informatique libre[1]. Ce que nous avons trouvé nous a enthousiasmés : non seulement il existe dans le monde de *l'Open Source* des interpréteurs et des compilateurs gratuits pour toute une série de langages, mais surtout ces langages sont modernes, performants, portables (c'est-à-dire utilisables sur différents systèmes d'exploitation tels que *Windows*, *Linux*, *Mac OS* ...), et fort bien documentés.

Le langage dominant y est sans conteste *C/C++*. Ce langage s'impose comme une référence absolue, et tout informaticien sérieux doit s'y frotter tôt ou tard. Il est malheureusement très rébarbatif et compliqué, trop proche de la machine. Sa syntaxe est peu lisible et fort contraignante. La mise au point d'un gros logiciel écrit en *C/C++* est longue et pénible. (Les mêmes remarques valent aussi dans une large mesure pour le langage *Java*.)

D'autre part, la pratique moderne de ce langage fait abondamment appel à des générateurs d'applications et autres outils d'assistance très élaborés tels *C++Builder*, *Kdevelop*, etc. Ces environnements de programmation peuvent certainement se révéler très efficaces entre les mains de programmeurs expérimentés, mais ils proposent d'emblée beaucoup trop d'outils complexes, et ils présupposent de la part de l'utilisateur des connaissances qu'un débutant ne maîtrise évidemment pas encore. Ce seront donc aux yeux de celui-ci de véritables « usines à gaz » qui risquent de lui masquer les mécanismes de base du langage lui-même. Nous laisserons donc le *C/C++* pour plus tard.

Pour nos débuts dans l'étude de la programmation, il nous semble préférable d'utiliser un langage de plus haut niveau, moins contraignant, à la syntaxe plus lisible. Après avoir successivement examiné et expérimenté quelque peu les langages *Perl* et *Tcl/Tk* , nous avons finalement décidé d'adopter Python, langage très moderne à la popularité grandissante.

Présentation du langage Python

> *Ce texte de Stéfane Fermigier date un peu, mais il reste d'actualité pour l'essentiel. Il est extrait d'un article paru dans le magazine Programmez! en décembre 1998. Il est également disponible sur http://www.linux-center.org/articles/9812/python.html. Stéfane Fermigier est le co-fondateur de l'AFUL (Association Francophone des Utilisateurs de Linux et des logiciels libres).*

[1] Un logiciel libre (*Free Software*) est avant tout un logiciel dont le code source est accessible à tous (*Open source*). Souvent gratuit (ou presque), copiable et modifiable librement au gré de son acquéreur, il est généralement le produit de la collaboration bénévole de centaines de développeurs enthousiastes dispersés dans le monde entier. Son code source étant « épluché » par de très nombreux spécialistes (étudiants et professeurs universitaires), un logiciel libre se caractérise la plupart du temps par un très haut niveau de qualité technique. Le plus célèbre des logiciels libres est le système d'exploitation ***GNU/Linux***, dont la popularité ne cesse de s'accroître de jour en jour.

Python est un langage portable, dynamique, extensible, gratuit, qui permet (sans l'imposer) une approche modulaire et orientée objet de la programmation. Python est développé depuis 1989 par Guido van Rossum et de nombreux contributeurs bénévoles.

Caractéristiques du langage

Détaillons un peu les principales caractéristiques de Python, plus précisément, du langage et de ses deux implantations actuelles:

- Python est **portable**, non seulement sur les différentes variantes d'*Unix*, mais aussi sur les OS propriétaires : *Mac OS*, *BeOS*, *NeXTStep*, *MS-DOS* et les différentes variantes de *Windows*. Un nouveau compilateur, baptisé *JPython*, est écrit en Java et génère du *bytecode* Java.
- Python est **gratuit**, mais on peut l'utiliser sans restriction dans des projets commerciaux.
- Python convient aussi bien à des **scripts** d'une dizaine de lignes qu'à des **projets complexes** de plusieurs dizaines de milliers de lignes.
- La **syntaxe** de Python est **très simple** et, combinée à des **types de données évolués** (listes, dictionnaires...), conduit à des programmes à la fois très compacts et très lisibles. À fonctionnalités égales, un programme Python (abondamment commenté et présenté selon les canons standards) est souvent de 3 à 5 fois plus court qu'un programme C ou C++ (ou même Java) équivalent, ce qui représente en général un temps de développement de 5 à 10 fois plus court et une facilité de maintenance largement accrue.
- Python gère ses ressources (mémoire, descripteurs de fichiers...) sans intervention du programmeur, par un mécanisme de **comptage de références** (proche, mais différent, d'un *garbage collector*).
- Il n'y a **pas de pointeurs** explicites en Python.
- Python est (optionnellement) **multi-threadé**.
- Python est **orienté-objet**. Il supporte **l'héritage multiple** et **la surcharge des opérateurs**. Dans son modèle objets, et en reprenant la terminologie de C++, toutes les méthodes sont virtuelles.
- Python intègre, comme Java ou les versions récentes de C++, un système **d'exceptions**, qui permettent de simplifier considérablement la gestion des erreurs.
- Python est **dynamique** (l'interpréteur peut évaluer des chaînes de caractères représentant des expressions ou des instructions Python), **orthogonal** (un petit nombre de concepts suffit à engendrer des constructions très riches), **réflectif** (il supporte la métaprogrammation, par exemple la capacité pour un objet de se rajouter ou de s'enlever des attributs ou des méthodes, ou même de changer de classe en cours d'exécution) et **introspectif** (un grand nombre d'outils de développement, comme le *debugger* ou le *profiler*, sont implantés en Python lui-même).
- Comme *Scheme* ou *SmallTalk*, Python est dynamiquement typé. Tout objet manipulable par le programmeur possède un type bien défini à l'exécution, qui n'a pas besoin d'être déclaré à l'avance.

- Python possède actuellement deux implémentations. L'une, **interprétée**, dans laquelle les programmes Python sont compilés en instructions portables, puis exécutés par une machine virtuelle (comme pour Java, avec une différence importante : Java étant statiquement typé, il est beaucoup plus facile d'accélérer l'exécution d'un programme Java que d'un programme Python). L'autre génère directement du *bytecode* Java.
- Python est **extensible** : comme *Tcl* ou *Guile*, on peut facilement l'interfacer avec des bibliothèques C existantes. On peut aussi s'en servir comme d'un langage d'extension pour des systèmes logiciels complexes.
- La **bibliothèque standard** de Python, et les paquetages contribués, donnent accès à une grande variété de services : chaînes de caractères et expressions régulières, services UNIX standards (fichiers, *pipes*, signaux, sockets, threads...), protocoles Internet (Web, News, FTP, CGI, HTML...), persistance et bases de données, interfaces graphiques.
- Python est un langage qui **continue à évoluer**, soutenu par une communauté d'utilisateurs enthousiastes et responsables, dont la plupart sont des supporters du logiciel libre. Parallèlement à l'interpréteur principal, écrit en C et maintenu par le créateur du langage, un deuxième interpréteur, écrit en Java, est en cours de développement.
- Enfin, Python est un langage de choix pour traiter le XML.

Pour le professeur qui souhaite utiliser cet ouvrage comme support de cours

Nous souhaitons avec ces notes ouvrir un maximum de portes. À notre niveau d'études, il nous paraît important de montrer que la programmation d'un ordinateur est un vaste univers de concepts et de méthodes, dans lequel chacun peut trouver son domaine de prédilection. Nous ne pensons pas que tous nos étudiants doivent apprendre exactement les mêmes choses. Nous voudrions plutôt qu'ils arrivent à développer chacun des compétences quelque peu différentes, qui leur permettent de se valoriser à leurs propres yeux ainsi qu'à ceux de leurs condisciples, et également d'apporter leur contribution spécifique lorsqu'on leur proposera de collaborer à des travaux d'envergure.

De toute manière, notre préoccupation primordiale doit être d'arriver à susciter l'intérêt, ce qui est loin d'être acquis d'avance pour un sujet aussi ardu que la programmation d'un ordinateur. Nous ne voulons pas feindre de croire que nos jeunes élèves vont se passionner d'emblée pour la construction de beaux algorithmes. Nous sommes plutôt convaincus qu'un certain intérêt ne pourra durablement s'installer qu'à partir du moment où ils commenceront à réaliser qu'ils sont devenus capables de développer un projet personnel original, dans une certaine autonomie.

Ce sont ces considérations qui nous ont amenés à développer une structure de cours que certains trouveront peut-être un peu chaotique. Nous commençons par une série de chapitres très courts, qui expliquent sommairement ce qu'est l'activité de programmation et posent les quelques bases indispensables à la réalisation de petits programmes. Ceux-ci pourront faire appel très tôt à des bibliothèques d'objets existants, tels ceux de l'interface graphique *tkinter* par exemple, afin que ce concept d'objet devienne rapidement familier. Ils devront être suffisamment attrayants pour que leurs auteurs aient le sentiment d'avoir déjà acquis une certaine maî-

trise. Nous souhaiterions en effet que les élèves puissent déjà réaliser une petite application graphique dès la fin de leur première année d'études.

Très concrètement, cela signifie que nous pensons pouvoir explorer les huit premiers chapitres de ces notes durant la première année de cours. Cela suppose que l'on aborde d'abord toute une série de concepts importants (types de données, variables, instructions de contrôle du flux, fonctions et boucles) d'une manière assez rapide, sans trop se préoccuper de ce que chaque concept soit parfaitement compris avant de passer au suivant, en essayant plutôt d'inculquer le goût de la recherche personnelle et de l'expérimentation. Il sera souvent plus efficace de ré-expliquer les notions et les mécanismes essentiels plus tard, en situation et dans des contextes variés.

Dans notre esprit, c'est surtout en seconde année que l'on cherchera à structurer les connaissances acquises, en les approfondissant. Les algorithmes seront davantage décortiqués et commentés. Les projets, cahiers des charges et méthodes d'analyse seront discutés en concertation. On exigera la tenue régulière d'un cahier de notes et la rédaction de rapports techniques pour certains travaux.

L'objectif ultime sera pour chaque élève de réaliser un projet de programmation original d'une certaine importance. On s'efforcera donc de boucler l'étude théorique des concepts essentiels suffisamment tôt dans l'année scolaire, afin que chacun puisse disposer du temps nécessaire.

Il faut bien comprendre que les nombreuses informations fournies dans ces notes concernant une série de domaines particuliers (gestion des interfaces graphiques, des communications, des bases de données, etc.) sont facultatives. Ce sont seulement une série de suggestions et de repères que nous avons inclus pour aider les étudiants à choisir et à commencer leur projet personnel de fin d'études. Nous ne cherchons en aucune manière à former des spécialistes d'un certain langage ou d'un certain domaine technique : nous voulons simplement donner un petit aperçu des immenses possibilités qui s'offrent à celui qui se donne la peine d'acquérir une compétence de programmeur.

Versions du langage

Python continue à évoluer, mais cette évolution ne vise qu'à améliorer ou perfectionner le produit. Il est donc très rare qu'il faille modifier les programmes afin de les adapter à une nouvelle version qui serait devenue incompatible avec les précédentes. Les exemples de ce livre ont été réalisés les uns après les autres sur une période de temps relativement longue : certains ont été développés sous Python 1.5.2, puis d'autres sous Python 1.6, Python 2.0, 2.1, 2.2, 2.3, 2.4, etc. Ils n'ont guère nécessité de modifications avant l'apparition de Python 3.

Cette nouvelle version du langage a cependant apporté quelques changements de fond qui lui confèrent une plus grande cohérence et même une plus grande facilité d'utilisation, mais qui imposent une petite mise à jour de tous les scripts écrits pour les versions précédentes. La présente édition de ce livre a donc été remaniée, non seulement pour adapter ses exemples à la nouvelle version, mais surtout pour tirer parti de ses améliorations, qui en font probablement le meilleur outil d'apprentissage de la programmation à l'heure actuelle.

Installez donc sur votre système la dernière version disponible (quelques-uns de nos exemples nécessitent désormais la version 3.1 ou une version postérieure), et amusez-vous bien ! Si toute-

fois vous devez analyser des scripts développés pour une version antérieure, sachez que des outils de conversion existent (voir en particulier le script 2to3.py), et que nous maintenons en ligne sur notre site web *http://inforef.be/swi/python.htm* la précédente mouture de ce texte, adaptée aux versions antérieures de Python, et toujours librement téléchargeable.

Distribution de Python et bibliographie

Les différentes versions de Python (pour *Windows*, *Unix*, etc.), son **tutoriel** original, son **manuel de référence**, la **documentation** des bibliothèques de fonctions, etc. sont disponibles en téléchargement gratuit depuis Internet, à partir du site web officiel : *http://www.python.org*

Vous pouvez aussi déjà trouver en ligne et en français, l'excellent cours sur Python 3 de Robert Cordeau, professeur à l'IUT d'Orsay, qui complète excellemment celui-ci. Ce cours est disponible sur le site de l'AFPY, à l'adresse : *http://zope.afpy.org/Members/tarek/initiation-python-3*

Il existe également de très bons ouvrages imprimés concernant Python. La plupart concernent encore Python 2.x, mais vous ne devrez guère éprouver de difficultés à adapter leurs exemples à Python 3. En langue française, vous pourrez très profitablement consulter les manuels ci-après :

- *Programmation Python* , par Tarek Ziadé, Editions Eyrolles, Paris, 2006, 538 p., ISBN 978-2-212-11677-9. C'est l'un des premiers ouvrages édités directement en langue française sur le langage Python. Excellent. Une mine de renseignements essentielle si vous voulez acquérir les meilleures pratiques et vous démarquer des débutants.
- *Au cœur de Python* , volumes 1 et 2, par Wesley J. Chun, traduction de *Python core programming, 2d edition* (Prentice Hall) par Marie-Cécile Baland, Anne Bohy et Luc Carité, Editions CampusPress, Paris, 2007, respectivement 645 et 385 p., ISBN 978-2-7440-2148-0 et 978-2-7440-2195-4. C'est un ouvrage de référence indispensable, très bien écrit.

D'autres excellents ouvrages en français étaient proposés par la succursale française de la maison d'éditions O'Reilly, laquelle a malheureusement disparu. En langue anglaise, le choix est évidemment beaucoup plus vaste. Nous apprécions personnellement beaucoup ***Python : How to program***, par Deitel, Liperi & Wiedermann, Prentice Hall, Upper Saddle River - NJ 07458, 2002, 1300 p., ISBN 0-13-092361-3, très complet, très clair, agréable à lire et qui utilise une méthodologie éprouvée.

Pour aller plus loin, notamment dans l'utilisation de la bibliothèque graphique ***Tkinter***, on pourra utilement consulter ***Python and Tkinter Programming***, par John E. Grayson, Manning publications co., Greenwich (USA), 2000, 658 p., ISBN 1-884777-81-3 , et surtout l'incontournable ***Programming Python*** (second edition) de Mark Lutz, Editions O'Reilly, 2001, 1255 p., ISBN 0-596-00085-5, qui est une extraordinaire mine de renseignements sur de multiples aspects de la programmation moderne (sur tous systèmes).

Si vous savez déjà bien programmer, et que vous souhaitez progresser encore en utilisant les concepts les plus avancés de l'algorithmique Pythonienne, procurez vous ***Python cookbook***, par Alex Martelli et David Ascher, Editions O'Reilly, 2002, 575 p., ISBN 0-596-00167-3 , dont les recettes sont savoureuses.

Exemples du livre

Le code source des exemples de ce livre peut être téléchargé à partir du site de l'auteur :

http://inforef.be/swi/python.htm

ou encore à cette adresse : http://main.pythomium.net/download/cours_python.zip

Remerciements

Ce livre est pour une partie le résultat d'un travail personnel, mais pour une autre – bien plus importante – la compilation d'informations et d'idées mises à la disposition de tous par des professeurs et des chercheurs bénévoles. La source qui a inspiré mes premières ébauches du livre est le cours de A.Downey, J.Elkner & C.Meyers : *How to think like a computer scientist* (Voir : http://greenteapress.com/thinkpython/thinkCSpy). Merci encore à ces professeurs enthousiastes. J'avoue aussi m'être inspiré du tutoriel original écrit par Guido van Rossum lui-même (l'auteur principal de Python), ainsi que d'exemples et de documents divers émanant de la (très active) communauté des utilisateurs de Python. Il ne m'est malheureusement pas possible de préciser davantage les références de tous ces textes, mais je voudrais que leurs auteurs soient assurés de toute ma reconnaissance.

Merci également à tous ceux qui œuvrent au développement de Python, de ses accessoires et de sa documentation, à commencer par Guido van Rossum, bien sûr, mais sans oublier non plus tous les autres ((mal)heureusement trop nombreux pour que je puisse les citer tous ici).

Merci encore à mes collègues Freddy Klich et David Carrera, professeurs à l'Institut Saint-Jean Berchmans de Liège, qui ont accepté de se lancer dans l'aventure de ce nouveau cours avec leurs élèves, et ont également suggéré de nombreuses améliorations. Un merci tout particulier à Christophe Morvan, professeur à l'IUT de Marne-la-Vallée, pour ses avis précieux et ses encouragements. Grand merci aussi à Florence Leroy, mon éditrice chez O'Reilly, qui a corrigé mes incohérences et mes belgicismes avec une compétence sans faille. Merci encore à mes partenaires actuels chez Eyrolles, Sandrine Paniel, Muriel Shan Sei Fan, Matthieu Montaudouin et Taï-marc Le Thanh, qui ont efficacement pris en charge cette nouvelle édition.

Merci enfin à mon épouse Suzel, pour sa patience et sa compréhension.

Table des matières

1

À l'école des sorciers

Apprendre à programmer est une activité déjà très intéressante en elle-même : elle peut stimuler puissamment votre curiosité intellectuelle. Mais ce n'est pas tout. Acquérir cette compétence vous ouvre également la voie menant à la réalisation de projets tout à fait concrets (utiles ou ludiques), ce qui vous procurera certainement beaucoup de fierté et de grandes satisfactions.

Avant de nous lancer dans le vif du sujet, nous allons vous proposer ici quelques réflexions sur la nature de la programmation et le comportement parfois étrange de ceux qui la pratiquent, ainsi que l'explication de quelques concepts fondamentaux. Il n'est pas vraiment difficile d'apprendre à programmer, mais il faut de la méthode et une bonne dose de persévérance, car vous pourrez continuer à progresser sans cesse dans cette science : elle n'a aucune limite.

Boîtes noires et pensée magique

Une caractéristique remarquable de notre société moderne est que nous vivons de plus en plus entourés de nombreuses **boîtes noires**. Les scientifiques ont l'habitude de nommer ainsi les divers dispositifs technologiques que nous utilisons couramment, sans en connaître ni la structure ni le fonctionnement exacts. Tout le monde sait se servir d'un téléphone, par exemple, alors qu'il n'existe qu'un très petit nombre de techniciens hautement spécialisés qui soient capables d'en concevoir un nouveau modèle.

Des boîtes noires existent dans tous les domaines, et pour tout le monde. En général, cela ne nous affecte guère, car nous pouvons nous contenter d'une compréhension sommaire de leur mécanisme pour les utiliser sans état d'âme. Dans la vie courante, par exemple, la composition précise d'une pile électrique ne nous importe guère. Le simple fait de savoir qu'elle produit son électricité à partir d'une réaction chimique nous suffit pour admettre sans difficulté qu'elle sera épuisée après quelque temps d'utilisation, et qu'elle sera alors devenue un objet polluant qu'il ne faudra pas jeter n'importe où. Inutile donc d'en savoir davantage.

Il arrive cependant que certaines boîtes noires deviennent tellement complexes que nous n'arrivons plus à en avoir une compréhension suffisante pour les utiliser tout-à-fait correctement dans n'importe quelle circonstance. Nous pouvons alors être tentés de tenir à leur encontre des raisonnements qui se rattachent à *la pensée magique*, c'est-à-dire à une forme de pensée faisant appel à l'intervention de propriétés ou de pouvoirs surnaturels pour expliquer ce que notre raison n'arrive pas à comprendre. C'est ce qui se passe lorsqu'un magicien nous montre un tour de passe-passe, et que nous sommes enclins à croire qu'il possède un pouvoir particulier, tel un don

de « double vue », ou à accepter l'existence de mécanismes paranormaux (« fluide magnétique », etc.), tant que nous n'avons pas compris le truc utilisé.

Du fait de leur extraordinaire complexité, les ordinateurs constituent bien évidemment l'exemple type de la boîte noire. Même si vous avez l'impression d'avoir toujours vécu entouré de moniteurs vidéo et de claviers, il est fort probable que vous n'ayez qu'une idée très confuse de ce qui se passe réellement dans la machine, par exemple lorsque vous déplacez la souris, et qu'en conséquence de ce geste un petit dessin en forme de flèche se déplace docilement sur votre écran. Qu'est-ce qui se déplace, au juste ? Vous sentez-vous capable de l'expliquer en détail, sans oublier (entre autres) les capteurs, les ports d'interface, les mémoires, les portes et bascules logiques, les transistors, les bits, les octets, les interruptions processeur, les cristaux liquides de l'écran, la micro-programmation, les pixels, le codage des couleurs ... ?

De nos jours, plus personne ne peut prétendre maîtriser absolument toutes les connaissances techniques et scientifiques mises en œuvre dans le fonctionnement d'un ordinateur. Lorsque nous utilisons ces machines, nous sommes donc forcément amenés à les traiter mentalement, en partie tout au moins, comme des objets magiques, sur lesquels nous sommes habilités à exercer un certain pouvoir, magique lui aussi.

Par exemple, nous comprenons tous très bien une instruction telle que : « déplacer la fenêtre d'application en la saisissant par sa barre de titre ». Dans le monde réel, nous savons parfaitement ce qu'il faut faire pour l'exécuter, à savoir manipuler un dispositif technique familier (souris, pavé tactile …) qui va transmettre des impulsions électriques à travers une machinerie d'une complexité prodigieuse, avec pour effet ultime la modification de l'état de transparence ou de luminosité d'une partie des pixels de l'écran. Mais dans notre esprit, il ne sera nullement question d'interactions physiques ni de circuiterie complexe. C'est un objet tout à fait virtuel qui sera activé (la flèche du curseur se déplaçant à l'écran), et qui agira tout à fait comme une baguette magique, pour faire obéir un objet tout aussi virtuel et magique (la fenêtre d'application). L'explication rationnelle de ce qui se passe effectivement dans la machine est donc tout à fait escamotée au profit d'un « raisonnement » figuré, qui nous rassure par sa simplicité, mais qui est bel et bien une illusion.

Si vous vous intéressez à la programmation des ordinateurs, sachez que vous serez constamment confronté à des formes diverses de cette « pensée magique », non seulement chez les autres (par exemple ceux qui vous demanderont de réaliser tel ou tel programme), mais surtout aussi dans vos propres représentations mentales. Vous devrez inlassablement démonter ces pseudo-raisonnements qui ne sont en fait que des spéculations, basées sur des interprétations figuratives simplifiées de la réalité, pour arriver à mettre en lumière (au moins en partie) leurs implications concrètes véritables.

Ce qui est un peu paradoxal, et qui justifie le titre de ce chapitre, c'est qu'en progressant dans cette compétence, vous allez acquérir de plus en plus de pouvoir sur la machine, et de ce fait vous allez vous-même devenir petit à petit aux yeux des autres, une sorte de magicien !

Bienvenue donc, comme le célèbre *Harry Potter*, à l'école des sorciers !

Magie blanche, magie noire

Nous n'avons bien évidemment aucune intention d'assimiler la programmation d'un ordinateur à une science occulte. Si nous vous accueillons ici comme un apprenti sorcier, c'est seulement pour attirer votre attention sur ce qu'implique cette image que vous donnerez probablement de vous-même (involontairement) à vos contemporains. Il peut être intéressant aussi d'emprunter quelques termes au vocabulaire de la magie pour illustrer plaisamment certaines pratiques.

La programmation est l'art d'apprendre à une machine comment elle pourra accomplir des tâches nouvelles, qu'elle n'avait jamais été capable d'effectuer auparavant. C'est par la programmation que vous pourrez acquérir le plus de contrôle, non seulement sur votre machine, mais aussi peut-être sur celles des autres, par l'intermédiaire des réseaux. D'une certaine façon, cette activité peut donc être assimilée à une forme particulière de magie. Elle donne effectivement à celui qui l'exerce un certain **pouvoir**, mystérieux pour le plus grand nombre, voire inquiétant quand on se rend compte qu'il peut être utilisé à des fins malhonnêtes.

Dans le monde de la programmation, on désigne par le terme de ***hackers*** les programmeurs chevronnés qui ont perfectionné les systèmes d'exploitation de type *Unix* et mis au point les techniques de communication qui sont à la base du développement extraordinaire de l'internet. Ce sont eux également qui continuent inlassablement à produire et à améliorer les logiciels libres, dits « *open source* ». Dans notre analogie, les *hackers* sont donc des maîtres-sorciers, qui pratiquent la magie blanche.

Mais il existe aussi un autre groupe de gens que les journalistes mal informés désignent erronément sous le nom de *hackers* (alors qu'ils devraient les appeler plutôt des *crackers*). Ces personnes se prétendent *hackers* parce qu'ils veulent faire croire qu'ils sont très compétents, alors qu'en général ils ne le sont guère. Ils sont cependant très nuisibles, parce qu'ils utilisent leurs quelques connaissances pour rechercher les moindres failles des systèmes informatiques construits par d'autres, afin d'y effectuer toutes sortes d'opérations illicites : vol d'informations confidentielles, escroquerie, diffusion de spam, de virus, de propagande haineuse, de pornographie et de contrefaçons, destruction de sites web, etc. Ces sorciers dépravés s'adonnent bien sûr à une forme grave de magie noire.

Mais il y en a une autre.

Les vrais *hackers* cherchent à promouvoir dans leur domaine une certaine éthique, basée principalement sur l'émulation et le partage des connaissances[2]. La plupart d'entre eux sont des perfectionnistes, qui veillent non seulement à ce que leurs constructions logiques soient efficaces, mais aussi à ce qu'elles soient élégantes, avec une structure parfaitement lisible et documentée. Vous découvrirez rapidement qu'il est aisé de produire à la va-vite des programmes d'ordinateur qui fonctionnent, certes, mais qui sont obscurs et confus, indéchiffrables pour toute autre personne que leur auteur (et encore !). Cette forme de programmation abstruse et ingérable est souvent aussi qualifiée de « magie noire » par les *hackers*.

[2]Voir à ce sujet le texte de *Eric Steven Raymond* : « Comment devenir un hacker », reproduit sur de nombreux sites web, notamment sur : http://www.secuser.com/dossiers/devenir_hacker.htm, ou encore sur : http://www.forumdz.com/showthread.php?t=4593

La démarche du programmeur

Comme le sorcier, le programmeur compétent semble doté d'un pouvoir étrange qui lui permet de transformer une machine en une autre, une machine à calculer en une machine à écrire ou à dessiner, par exemple, un peu à la manière d'un sorcier qui transformerait un prince charmant en grenouille, à l'aide de quelques incantations mystérieuses entrées au clavier. Comme le sorcier, il est capable de guérir une application apparemment malade, ou de jeter des sorts à d'autres, via l'internet.

Mais comment cela est-il possible ?

Cela peut paraître paradoxal, mais comme nous l'avons déjà fait remarquer plus haut, le vrai maître est en fait celui qui ne croit à aucune magie, aucun don, aucune intervention surnaturelle. Seule la froide, l'implacable, l'inconfortable logique est de mise.

Le mode de pensée d'un programmeur combine des constructions intellectuelles complexes, similaires à celles qu'accomplissent les mathématiciens, les ingénieurs et les scientifiques. Comme le mathématicien, il utilise des langages formels pour décrire des raisonnements (ou algorithmes). Comme l'ingénieur, il conçoit des dispositifs, il assemble des composants pour réaliser des mécanismes et il évalue leurs performances. Comme le scientifique, il observe le comportement de systèmes complexes, il crée des modèles, il teste des prédictions.

L'activité essentielle d'un programmeur consiste à résoudre des problèmes.

Il s'agit-là d'une compétence de haut niveau, qui implique des capacités et des connaissances diverses : être capable de (re)formuler un problème de plusieurs manières différentes, être capable d'imaginer des solutions innovantes et efficaces, être capable d'exprimer ces solutions de manière claire et complète. Comme nous l'avons déjà évoqué plus haut, il s'agira souvent de mettre en lumière les implications concrètes d'une représentation mentale « magique », simpliste ou trop abstraite.

La programmation d'un ordinateur consiste en effet à « expliquer » en détail à une machine ce qu'elle doit faire, en sachant d'emblée qu'elle ne peut pas véritablement « comprendre » un langage humain, mais seulement effectuer un traitement automatique sur des séquences de caractères. Il s'agit la plupart du temps de convertir un souhait exprimé à l'origine en termes « magiques », en un vrai raisonnement parfaitement structuré et élucidé dans ses moindres détails, que l'on appelle un *algorithme*.

Considérons par exemple une suite de nombres fournis dans le désordre : 47, 19, 23, 15, 21, 36, 5, 12 … Comment devons-nous nous y prendre pour obtenir d'un ordinateur qu'il les remette dans l'ordre ?

Le souhait « magique » est de n'avoir à effectuer qu'un clic de souris sur un bouton, ou entrer une seule instruction au clavier, pour qu'automatiquement les nombres se mettent en place. Mais le travail du sorcier-programmeur est justement de créer cette « magie ». Pour y arriver, il devra décortiquer tout ce qu'implique pour nous une telle opération de tri (au fait, existe-t-il une méthode unique pour cela, ou bien y en a-t-il plusieurs ?), et en traduire toutes les étapes en une suite d'instructions simples, telles que par exemple : « comparer les deux premiers nombres, les échanger s'ils ne sont pas dans l'ordre souhaité, recommencer avec le deuxième et le troisième, etc., etc., ...».

Si les instructions ainsi mises en lumière sont suffisamment simples, il pourra alors les encoder dans la machine en respectant de manière très stricte un ensemble de conventions fixées à l'avance, que l'on appelle un **langage informatique**. Pour « comprendre » celui-ci, la machine sera pourvue d'un mécanisme qui décode ces instructions en associant à chaque « mot » du langage une action précise. Ainsi seulement, la magie pourra s'accomplir.

Langage machine, langage de programmation

À strictement parler, un ordinateur n'est rien d'autre qu'une machine effectuant des opérations simples sur des séquences de signaux électriques, lesquels sont conditionnés de manière à ne pouvoir prendre que deux états seulement (par exemple un potentiel électrique maximum ou minimum). Ces séquences de signaux obéissent à une logique du type « tout ou rien » et peuvent donc être considérés conventionnellement comme des suites de nombres ne prenant jamais que les deux valeurs 0 et 1. Un système numérique ainsi limité à deux chiffres est appelé système binaire.

Sachez dès à présent que dans son fonctionnement interne, un ordinateur est totalement incapable de traiter autre chose que des nombres binaires. Toute information d'un autre type doit être convertie, ou codée, *en format binaire*. Cela est vrai non seulement pour les données que l'on souhaite traiter (les textes, les images, les sons, les nombres, etc.), mais aussi pour les programmes, c'est-à-dire les séquences d'instructions que l'on va fournir à la machine pour lui dire ce qu'elle doit faire avec ces données.

Le seul « langage » que l'ordinateur puisse véritablement « comprendre » est donc très éloigné de ce que nous utilisons nous-mêmes. C'est une longue suite de 1 et de 0 (les « bits ») souvent traités par groupes de 8 (les « octets »), 16, 32, ou même 64. Ce « langage machine » est évidemment presque incompréhensible pour nous. Pour « parler » à un ordinateur, il nous faudra utiliser des systèmes de traduction automatiques, capables de convertir en nombres binaires des suites de caractères formant des mots-clés (anglais en général) qui seront plus significatifs pour nous.

Ces systèmes de traduction automatique seront établis sur la base de toute une série de conventions, dont il existera évidemment de nombreuses variantes.

Le système de traduction proprement dit s'appellera *interpréteur* ou bien *compilateur*, suivant la méthode utilisée pour effectuer la traduction. On appellera *langage de programmation* un ensemble de mots-clés (choisis arbitrairement) associé à un ensemble de règles très précises indiquant comment on peut assembler ces mots pour former des « phrases » que l'interpréteur ou le compilateur puisse traduire en langage machine (binaire).

Suivant son niveau d'abstraction, on pourra dire d'un langage qu'il est « de bas niveau » (ex : *assembleur*) ou « de haut niveau » (ex : *Pascal*, *Perl*, *Smalltalk*, *Scheme*, *Lisp*...). Un langage de bas niveau est constitué d'instructions très élémentaires, très « proches de la machine ». Un langage de haut niveau comporte des instructions plus abstraites, plus « puissantes » (et donc plus « magiques »). Cela signifie que chacune de ces instructions pourra être traduite par l'interpréteur ou le compilateur en un grand nombre d'instructions machine élémentaires.

Le langage que vous avez allez apprendre en premier est *Python*. Il s'agit d'un langage de haut niveau, dont la traduction en code binaire est complexe et prend donc toujours un certain

temps. Cela pourrait paraître un inconvénient. En fait, les avantages que présentent les langages de haut niveau sont énormes : il est *beaucoup plus facile* d'écrire un programme dans un langage de haut niveau ; l'écriture du programme prend donc beaucoup moins de temps ; la probabilité d'y faire des fautes est nettement plus faible ; la maintenance (c'est-à-dire l'apport de modifications ultérieures) et la recherche des erreurs (les « bugs ») sont grandement facilitées. De plus, un programme écrit dans un langage de haut niveau sera souvent *portable*, c'est-à-dire que l'on pourra le faire fonctionner sans guère de modifications sur des machines ou des systèmes d'exploitation différents. Un programme écrit dans un langage de bas niveau ne peut jamais fonctionner que sur un seul type de machine : pour qu'une autre l'accepte, il faut le réécrire entièrement.

Dans ce que nous venons d'expliquer sommairement, vous aurez sans doute repéré au passage de nombreuses « boîtes noires » : interpréteur, système d'exploitation, langage, instructions machine, code binaire, etc. L'apprentissage de la programmation va vous permettre d'en entrouvrir quelques-unes. Restez cependant conscients que vous n'arriverez pas à les décortiquer toutes. De nombreux *objets* informatiques créés par d'autres resteront probablement « magiques » pour vous pendant longtemps (à commencer par le langage de programmation lui-même, par exemple). Vous devrez donc faire confiance à leurs auteurs, quitte à être déçus parfois en constatant que cette confiance n'est pas toujours méritée. Restez donc vigilants, apprenez à vérifier, à vous documenter sans cesse. Dans vos propres productions, soyez rigoureux et évitez à tout prix la « magie noire » (les programmes pleins d'astuces tarabiscotées que vous êtes seul à comprendre) : un *hacker* digne de confiance n'a rien à cacher.

Édition du code source - Interprétation

Le programme tel que nous l'écrivons dans un langage de programmation quelconque est à strictement parler un simple texte. Pour rédiger ce texte, on peut faire appel à toutes sortes de logiciels plus ou moins perfectionnés, à la condition qu'ils ne produisent que du texte brut, c'est-à-dire sans mise en page particulière ni aucun attribut de style (pas de spécification de police, donc, pas de gros titres, pas de gras, ni de souligné, ni d'italique, etc.)[3].

Le texte ainsi produit est ce que nous appellerons désormais un « code source ».

Comme nous l'avons déjà évoqué plus haut, le code source doit être traduit en une suite d'instructions binaires directement compréhensibles par la machine : le « code objet ». Dans le cas de Python, cette traduction est prise en charge par un *interpréteur* assisté d'un *pré-compilateur*. Cette technique hybride (également utilisée par le langage *Java*) vise à exploiter au maximum les avantages de l'interprétation et de la compilation, tout en minimisant leurs inconvénients respectifs.

Veuillez consulter un ouvrage d'informatique générale si vous voulez en savoir davantage sur ces deux techniques. Sachez simplement à ce sujet que vous pourrez réaliser des programmes extrê-

[3]Ces logiciels sont appelés des *éditeurs de texte*. Même s'ils proposent divers automatismes, et sont souvent capables de mettre en évidence certains éléments du texte traité (coloration syntaxique, par ex.), ils ne produisent strictement que du texte non formaté. Ils sont donc assez différents des logiciels de *traitement de texte*, dont la fonction consiste justement à mettre en page et à ornementer un texte avec des attributs de toute sorte, de manière à le rendre aussi agréable à lire que possible.

mement performants avec Python, même s'il est indiscutable qu'un langage strictement compilé tel que le C peut toujours faire mieux en termes de rapidité d'exécution.

Mise au point d'un programme - Recherche des erreurs (debug)

La programmation est une démarche très complexe, et comme c'est le cas dans toute activité humaine, on y commet de nombreuses erreurs. Pour des raisons anecdotiques, les erreurs de programmation s'appellent des « *bugs* » (ou « bogues », en Français)[4], et l'ensemble des techniques que l'on met en œuvre pour les détecter et les corriger s'appelle « *debug* » (ou « débogage »).

En fait, il peut exister dans un programme trois types d'erreurs assez différentes, et il convient que vous appreniez à bien les distinguer.

Erreurs de syntaxe

Python ne peut exécuter un programme que si sa syntaxe est parfaitement correcte. Dans le cas contraire, le processus s'arrête et vous obtenez un message d'erreur. Le terme syntaxe se réfère aux règles que les auteurs du langage ont établies pour la structure du programme.

Tout langage comporte sa syntaxe. Dans la langue française, par exemple, une phrase doit toujours commencer par une majuscule et se terminer par un point. ainsi cette phrase comporte deux erreurs de syntaxe

Dans les textes ordinaires, la présence de quelques petites fautes de syntaxe par-ci par-là n'a généralement pas d'importance. Il peut même arriver (en poésie, par exemple), que des fautes de syntaxe soient commises volontairement. Cela n'empêche pas que l'on puisse comprendre le texte.

Dans un programme d'ordinateur, par contre, la moindre erreur de syntaxe produit invariablement un arrêt de fonctionnement (un « plantage ») ainsi que l'affichage d'un message d'erreur. Au cours des premières semaines de votre carrière de programmeur, vous passerez certainement pas mal de temps à rechercher vos erreurs de syntaxe. Avec de l'expérience, vous en commettrez beaucoup moins.

Gardez à l'esprit que les mots et les symboles utilisés n'ont aucune signification en eux-mêmes : ce ne sont que des suites de codes destinés à être convertis automatiquement en nombres binaires. Par conséquent, il vous faudra être très attentifs à respecter scrupuleusement la syntaxe du langage.

---

[4] *bug* est à l'origine un terme anglais servant à désigner de petits insectes gênants, tels les punaises. Les premiers ordinateurs fonctionnaient à l'aide de « lampes » radios qui nécessitaient des tensions électriques assez élevées. Il est arrivé à plusieurs reprises que des petits insectes s'introduisent dans cette circuiterie complexe et se fassent électrocuter, leurs cadavres calcinés provoquant alors des court-circuits et donc des pannes incompréhensibles.
Le mot français « *bogue* » a été choisi par homonymie approximative. Il désigne la coque épineuse de la châtaigne.

Finalement, souvenez-vous que **tous** les détails ont de l'importance. Il faudra en particulier faire très attention à la *casse* (c'est-à-dire l'emploi des majuscules et des minuscules) et à la *ponctuation*. Toute erreur à ce niveau (même minime en apparence, tel l'oubli d'une virgule, par exemple) peut modifier considérablement la signification du code, et donc le déroulement du programme.

Il est heureux que vous fassiez vos débuts en programmation avec un langage interprété tel que Python. La recherche des erreurs y est facile et rapide. Avec les langages compilés (tel le C++), il vous faudrait recompiler l'intégralité du programme après chaque modification, aussi minime soit-elle.

Erreurs sémantiques

Le second type d'erreur est l'erreur sémantique ou erreur de logique. S'il existe une erreur de ce type dans un de vos programmes, celui-ci s'exécute parfaitement, en ce sens que vous n'obtenez aucun message d'erreur, mais le résultat n'est pas celui que vous attendiez : vous obtenez autre chose.

En réalité, le programme fait exactement ce que vous lui avez dit de faire. Le problème est que ce que vous lui avez dit de faire ne correspond pas à ce que vous vouliez qu'il fasse. La séquence d'instructions de votre programme ne correspond pas à l'objectif poursuivi. La sémantique (la logique) est incorrecte.

Rechercher des fautes de logique peut être une tâche ardue. C'est là que se révélera votre aptitude à démonter toute forme résiduelle de « pensée magique » dans vos raisonnements. Il vous faudra analyser patiemment ce qui sort de la machine et tâcher de vous représenter une par une les opérations qu'elle a effectuées, à la suite de chaque instruction.

Erreurs à l'exécution

Le troisième type d'erreur est l'erreur en cours d'exécution (*Run-time error*), qui apparaît seulement lorsque votre programme fonctionne déjà, mais que des circonstances particulières se présentent (par exemple, votre programme essaie de lire un fichier qui n'existe plus). Ces erreurs sont également appelées des *exceptions*, parce qu'elles indiquent généralement que quelque chose d'exceptionnel (et de malencontreux) s'est produit. Vous rencontrerez davantage ce type d'erreurs lorsque vous programmerez des projets de plus en plus volumineux, et vous apprendrez plus loin dans ce cours qu'il existe des techniques particulières pour les gérer.

Recherche des erreurs et expérimentation

L'une des compétences les plus importantes à acquérir au cours de votre apprentissage est celle qui consiste à *déboguer* efficacement un programme. Il s'agit d'une activité intellectuelle parfois énervante mais toujours très riche, dans laquelle il faut faire montre de beaucoup de perspicacité.

Ce travail ressemble par bien des aspects à une enquête policière. Vous examinez un ensemble de faits, et vous devez émettre des hypothèses explicatives pour reconstituer les processus et les événements qui ont logiquement entraîné les résultats que vous constatez.

Cette activité s'apparente aussi au travail expérimental en sciences. Vous vous faites une première idée de ce qui ne va pas, vous modifiez votre programme et vous essayez à nouveau. Vous avez émis une hypothèse, qui vous permet de prédire ce que devra donner la modification. Si la prédiction se vérifie, alors vous avez progressé d'un pas sur la voie d'un programme qui fonctionne. Si la prédiction se révèle fausse, alors il vous faut émettre une nouvelle hypothèse. Comme l'a bien dit Sherlock Holmes : « Lorsque vous avez éliminé l'impossible, ce qui reste, même si c'est improbable, doit être la vérité » (A. Conan Doyle, *Le signe des quatre*).

Pour certaines personnes, « programmer » et « déboguer » signifient exactement la même chose. Ce qu'elles veulent dire par là est que l'activité de programmation consiste en fait à modifier, à corriger sans cesse un même programme, jusqu'à ce qu'il se comporte finalement comme vous le vouliez. L'idée est que la construction d'un programme commence toujours par une ébauche qui fait déjà quelque chose (et qui est donc déjà déboguée), à laquelle on ajoute couche par couche de petites modifications, en corrigeant au fur et à mesure les erreurs, afin d'avoir de toute façon à chaque étape du processus un programme qui fonctionne.

Par exemple, vous savez que Linux est un système d'exploitation (et donc un gros logiciel) qui comporte des milliers de lignes de code. Au départ, cependant, cela a commencé par un petit programme simple que Linus Torvalds avait développé pour tester les particularités du processeur *Intel 80386*. D'après Larry Greenfield (« *The Linux user's guide* », beta version 1) : « L'un des premiers projets de Linus était un programme destiné à convertir une chaîne de caractères AAAA en BBBB. C'est cela qui plus tard finit par devenir Linux ! ».

Ce qui précède ne signifie pas que nous voulions vous pousser à programmer par approximations successives, à partir d'une vague idée. Lorsque vous démarrerez un projet de programmation d'une certaine importance, il faudra au contraire vous efforcer d'établir le mieux possible un *cahier des charges* détaillé, lequel s'appuiera sur un plan solidement construit pour l'application envisagée.