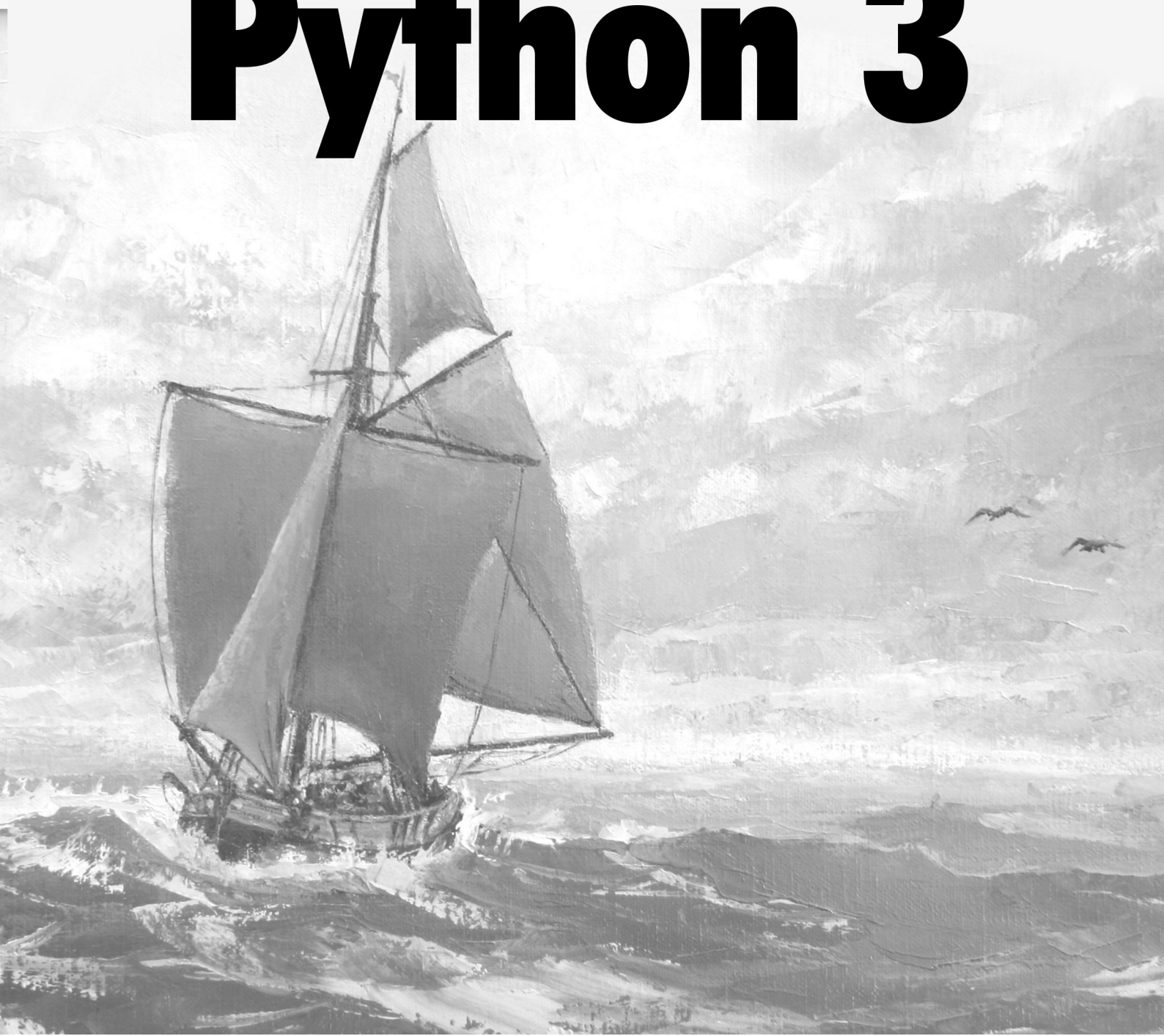
Gérard Swinnen

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 computer 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 d'après une gravure de J.J. Baujean. Elle met en scène un petit sloop de cabotage de la fin du 18e siècle. Ces bâtiments de 60 à 80 tonneaux possédaient une grande voile de fortune, utilisée par vent arrière comme on le voit ici, ainsi qu'un hunier pour les plus grands d'entre eux.

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 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'ils 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 concrètement aux yeux d'un programmeur débutant. Nous ob-

servons 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 il 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).*

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 :

---

[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 **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 maîtrise. 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 réexpliquer 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 toutefois 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 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. Il est disponible sur le site de l'AFPY, à l'adresse : http://www.afpy.org/Members/bcordeau/Python3v1-1.pdf/download

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, éditions Eyrolles, Paris, 2009, 586 p., ISBN 978-2-212-12483-5. 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, éditions 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, éditions 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, éditions 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://infos.pythomium.net/download/cours_python.zip

ainsi que sur la fiche de l'ouvrage :

http://www.editions-eyrolles.com

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* (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, et à Robert Cordeau, professeur à l'IUT d'Orsay, pour ses conseils et sa courageuse relecture. 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, Muriel Shan Sei Fan, Taï-Marc Le Thanh, Anne-Lise Banéath et Igor Barzilai 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 *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 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 états 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 vague 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 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 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, sachez que vous serez constamment confronté à diverses formes de cette « pensée magique », non seulement chez les autres (par exemple ceux qui vous demanderont de réaliser tel ou tel programme), mais aussi et surtout 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 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 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 accomplir de nouvelles tâches, 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 *hacker* 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 (Open Source). Selon 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 plutôt les appeler *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 qui fonctionnent, certes, mais qui sont obscurs et confus, indéchiffrables pour toute autre personne que leur auteur (et encore !). Cette forme de programmation absconse et ingérable est souvent aussi qualifiée de « magie noire » par les hackers.

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,

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

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, à aucun don, à 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 qu'à cliquer 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. ».

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 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 « bogues ») 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 conscient 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 déçu parfois en constatant que cette confiance n'est pas toujours méritée. Restez donc vigilant, 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 extrêmement 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 »).

[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 exemple), 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.

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.

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 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

[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.

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 en général que quelque chose d'exceptionnel (et de malencontreux) s'est produit. Vous rencontrerez 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 *In-*

tel 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.

Diverses méthodes existent pour effectuer cette tâche d'analyse, mais leur étude sort du cadre de ces notes. Nous vous présenterons cependant plus loin (voir chapitre 15) quelques idées de base.

2

Premiers pas

La programmation est donc l'art de commander à un ordinateur de faire exactement ce que vous voulez, et Python compte parmi les langages qu'il est capable de comprendre pour recevoir vos ordres. Nous allons essayer cela tout de suite avec des ordres très simples concernant des nombres, puisque les nombres constituent son matériau de prédilection. Nous allons lui fournir nos premières « instructions », et préciser au passage la définition de quelques termes essentiels du vocabulaire informatique, que vous rencontrerez constamment dans la suite de cet ouvrage.

Comme nous l'avons expliqué dans la préface (voir : Versions du langage, page XII), nous avons pris le parti d'utiliser dans ce cours la nouvelle version 3 de Python, laquelle a introduit quelques changements syntaxiques par rapport aux versions précédentes. Dans la mesure du possible, nous vous indiquerons ces différences dans le texte, afin que vous puissiez sans problème analyser ou utiliser d'anciens programmes écrits pour Python 1 ou 2.

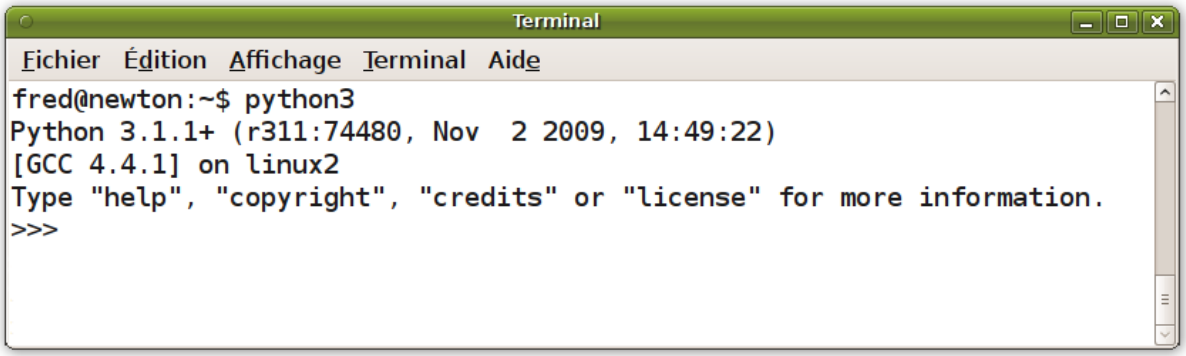
Calculer avec Python

Python présente la particularité de pouvoir être utilisé de plusieurs manières différentes. Vous allez d'abord l'utiliser *en mode interactif*, c'est-à-dire de manière à dialoguer avec lui directement depuis le clavier. Cela vous permettra de découvrir très vite un grand nombre de fonctionnalités du langage. Dans un second temps, vous apprendrez comment créer vos premiers programmes (scripts) et les sauvegarder sur disque.

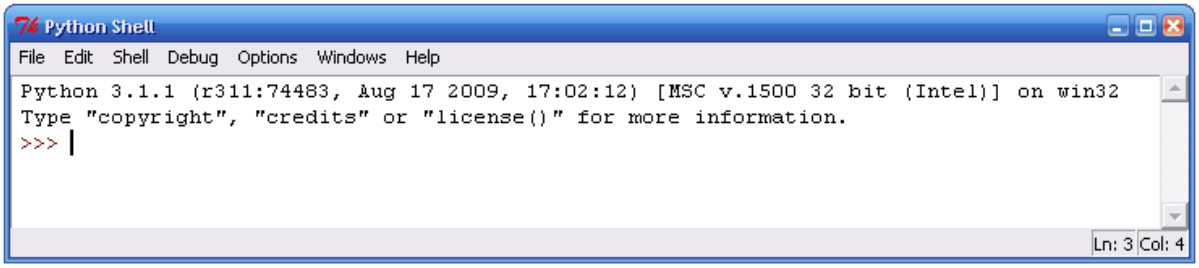
L'interpréteur peut être lancé directement depuis la ligne de commande (dans un « shell » *Linux*, ou bien dans une fenêtre *DOS* sous *Windows*) : il suffit d'y taper la commande **`python3`** (en supposant que le logiciel lui-même ait été correctement installé, et qu'il s'agisse d'une des dernières versions de Python), ou **`python`** (si la version de Python installée sur votre ordinateur est antérieure à la version 3.0).

Si vous utilisez une interface graphique telle que *Windows*, *Gnome*, *WindowMaker* ou *KDE*, vous préférerez vraisemblablement travailler dans une « fenêtre de terminal », ou encore dans un environ-

nement de travail spécialisé tel que *IDLE*. Voici par exemple ce qui apparaît dans une fenêtre de terminal *Gnome* (sous *Ubuntu Linux*)[5] :



Avec *IDLE* sous *Windows*, votre environnement de travail ressemblera à celui-ci :



Les trois caractères « supérieur à » constituent le signal d'invite, ou *prompt principal*, lequel vous indique que Python est prêt à exécuter une commande.

Par exemple, vous pouvez tout de suite utiliser l'interpréteur comme une simple calculatrice de bureau. Veuillez donc vous-même tester les commandes ci-dessous (Prenez l'habitude d'utiliser votre cahier d'exercices pour noter les résultats qui apparaissent à l'écran) :

```
>>> 5+3

>>> 2 - 9          # les espaces sont optionnels

>>> 7 + 3 * 4      # la hiérarchie des opérations mathématiques
                   # est-elle respectée ?

>>> (7+3)*4

>>> 20 / 3         # attention : ceci fonctionnerait différemment sous Python 2

>>> 20 // 3
```

[5] Sous *Windows,* vous aurez surtout le choix entre l'environnement *IDLE* développé par Guido Van Rossum, auquel nous donnons nous-même la préférence, et *PythonWin*, une interface de développement développée par Mark Hammond. D'autres environnements de travail plus sophistiqués existent aussi, tels l'excellent *Boa Constructor* par exemple (qui fonctionne de façon très similaire à *Delphi*), mais nous estimons qu'ils ne conviennent guère aux débutants. Pour tout renseignement complémentaire, veuillez consulter le site Web de Python.
Sous *Linux*, nous préférons personnellement travailler dans une simple fenêtre de terminal pour lancer l'interpréteur Python ou l'exécution des scripts, et faire appel à un éditeur de texte ordinaire tel que *Gedit*, *Kate*, ou un peu plus spécialisé comme *Geany* pour l'édition de ces derniers.

Comme vous pouvez le constater, les opérateurs arithmétiques pour l'addition, la soustraction, la multiplication et la division sont respectivement +, -, * et /. Les parenthèses ont la fonction attendue.

> *Sous Python 3, l'opérateur de division / effectue une division réelle. Si vous souhaitez obtenir une division entière (c'est-à-dire dont le résultat – tronqué – ne peut être qu'un entier), vous devez utiliser l'opérateur //. Veuillez noter que ceci est l'un des changements de syntaxe apportés à la version 3 de Python, par rapport aux versions précédentes. Si vous utilisez l'une de ces versions, sachez que l'opérateur / y effectue par défaut une division entière, si on lui fournit des arguments qui sont eux-mêmes des entiers, et une division réelle, si au moins l'un des arguments est un réel. Cet ancien comportement de Python a été heureusement abandonné, car il conduisait parfois à des bogues difficilement repérables.*

```
>>> 20.5 / 3

>>> 8,7 / 5          # Erreur !
```

Veuillez remarquer au passage une règle qui vaut dans tous les langages de programmation : les conventions mathématiques de base sont celles qui sont en vigueur dans les pays anglophones. Le séparateur décimal y est donc toujours un point, et non une virgule comme chez nous. Notez aussi que dans le monde de l'informatique, les nombres réels sont souvent désignés comme des nombres « à virgule flottante » (*floating point numbers*).

Données et variables

Nous aurons l'occasion de détailler plus loin les différents types de données numériques. Mais avant cela, nous pouvons dès à présent aborder un concept de grande importance.

L'essentiel du travail effectué par un programme d'ordinateur consiste à manipuler des *données*. Ces données peuvent être très diverses (tout ce qui est *numérisable*, en fait[6]), mais dans la mémoire de l'ordinateur elles se ramènent toujours en définitive à *une suite finie de nombres binaires*.

Pour pouvoir accéder aux données, le programme d'ordinateur (quel que soit le langage dans lequel il est écrit) fait abondamment usage d'un grand nombre de *variables* de différents types.

Une variable apparaît dans un langage de programmation sous un *nom de variable* à peu près quelconque (voir ci-après), mais pour l'ordinateur il s'agit d'une *référence* désignant une adresse mémoire, c'est-à-dire un emplacement précis dans la mémoire vive.

À cet emplacement est stockée une *valeur* bien déterminée. C'est la donnée proprement dite, qui est donc stockée sous la forme d'une suite de nombres binaires, mais qui n'est pas nécessairement un nombre aux yeux du langage de programmation utilisé. Cela peut être en fait à peu près n'importe quel « objet » susceptible d'être placé dans la mémoire d'un ordinateur, par exemple : un nombre entier, un nombre réel, un nombre complexe, un vecteur, une chaîne de caractères typographiques, un tableau, une fonction, etc.

Pour distinguer les uns des autres ces divers contenus possibles, le langage de programmation fait usage de différents types de variables (le type *entier*, le type *réel*, le type *chaîne de caractères*, le type *liste*, etc.). Nous allons expliquer tout cela dans les pages suivantes.

[6] *Que peut-on numériser au juste ?* Voilà une question très importante, qu'il vous faudra débattre dans votre cours d'informatique générale.

Noms de variables et mots réservés

Les noms de variables sont des noms que vous choisissez vous-même assez librement. Efforcez-vous cependant de bien les choisir : de préférence assez courts, mais aussi explicites que possible, de manière à exprimer clairement ce que la variable est censée contenir. Par exemple, des noms de variables tels que *altitude*, *altit* ou *alt* conviennent mieux que x pour exprimer une altitude.

Un bon programmeur doit veiller à ce que ses lignes d'instructions soient faciles à lire.

Sous Python, les noms de variables doivent en outre obéir à quelques règles simples :

- Un nom de variable est une séquence de lettres (a → z , A → Z) et de chiffres (0 → 9), qui doit toujours commencer par une lettre.
- Seules les lettres ordinaires sont autorisées. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que $, #, @, etc. sont interdits, à l'exception du caractère _ (souligné).
- La casse est significative (les caractères majuscules et minuscules sont distingués).
  Attention : Joseph, joseph, JOSEPH sont donc des variables différentes. Soyez attentifs !

Prenez l'habitude d'écrire l'essentiel des noms de variables en caractères minuscules (y compris la première lettre[7]). Il s'agit d'une simple convention, mais elle est largement respectée. N'utilisez les majuscules qu'à l'intérieur même du nom, pour en augmenter éventuellement la lisibilité, comme dans *tableDesMatieres*.

En plus de ces règles, il faut encore ajouter que vous ne pouvez pas utiliser comme nom de variables les 33 « mots réservés » ci-dessous (ils sont utilisés par le langage lui-même) :

and	as	assert	break	class	continue	def
del	elif	else	except	False	finally	for
from	global	if	import	in	is	lambda
None	nonlocal	not	or	pass	raise	return
True	try	while	with	yield		

Affectation (ou assignation)

Nous savons désormais comment choisir judicieusement un nom de variable. Voyons à présent comment nous pouvons *définir* une variable et lui *affecter* une valeur. Les termes « affecter une valeur » ou « assigner une valeur » à une variable sont équivalents. Ils désignent l'opération par laquelle on établit un lien entre le nom de la variable et sa valeur (son contenu).

En Python comme dans de nombreux autres langages, l'opération d'affectation est représentée par le signe *égale*[8] :

---

[7] Les noms commençant par une majuscule ne sont pas interdits, mais l'usage veut qu'on le réserve plutôt aux variables qui désignent des *classes* (le concept de classe sera abordé plus loin dans cet ouvrage). Il arrive aussi que l'on écrive entièrement en majuscules certaines variables que l'on souhaite traiter comme des pseudo-constantes (c'est-à-dire des variables que l'on évite de modifier au cours du programme).

[8] Il faut bien comprendre qu'il ne s'agit en aucune façon d'une égalité, et que l'on aurait très bien pu choisir un autre symbolisme, tel que **n ← 7** par exemple, comme on le fait souvent dans certains pseudo-langages servant à décrire des algorithmes, pour bien montrer qu'il s'agit de relier un contenu (la valeur 7) à un contenant (la variable n).

```
>>> n = 7                       # définir la variable n et lui donner la valeur 7
>>> msg = "Quoi de neuf ?"     # affecter la valeur "Quoi de neuf ?" à msg
>>> pi = 3.14159               # assigner sa valeur à la variable pi
```

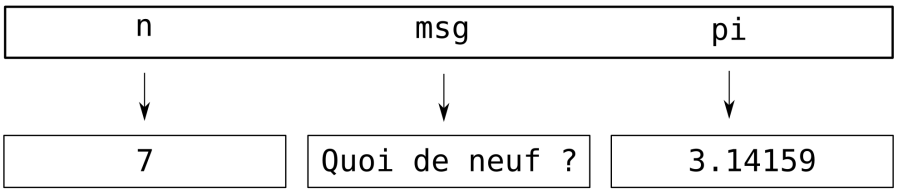
Les exemples ci-dessus illustrent des instructions d'affectation Python tout à fait classiques. Après qu'on les ait exécutées, il existe dans la mémoire de l'ordinateur, à des endroits différents :

- trois noms de variables, à savoir **n**, **msg** et **pi** ;
- trois séquences d'octets, où sont encodées le nombre entier **7**, la chaîne de caractères **Quoi de neuf ?** et le nombre réel **3,14159**.

Les trois instructions d'affectation ci-dessus ont eu pour effet chacune de réaliser plusieurs opérations dans la mémoire de l'ordinateur :

- créer et mémoriser un **nom de variable** ;
- lui attribuer un **type** bien déterminé (ce point sera explicité à la page suivante) ;
- créer et mémoriser une **valeur** particulière ;
- établir un **lien** (par un système interne de pointeurs) entre le nom de la variable et l'emplacement mémoire de la valeur correspondante.

On peut mieux se représenter tout cela par un diagramme d'état tel que celui-ci :



Les trois noms de variables sont des *références*, mémorisées dans une zone particulière de la mémoire que l'on appelle espace de noms, alors que les valeurs correspondantes sont situées ailleurs, dans des emplacements parfois fort éloignés les uns des autres. Nous aurons l'occasion de préciser ce concept plus loin dans ces pages.

Afficher la valeur d'une variable

À la suite de l'exercice ci-dessus, nous disposons donc des trois variables **n**, **msg** et **pi**.
Pour afficher leur valeur à l'écran, il existe deux possibilités. La première consiste à entrer au clavier le nom de la variable, puis <*Enter*>. Python répond en affichant la valeur correspondante :

```
>>> n
7
>>> msg
'Quoi de neuf ?'
>>> pi
3.14159
```

Il s'agit cependant là d'une fonctionnalité secondaire de l'interpréteur, qui est destinée à vous faciliter la vie lorsque vous faites de simples exercices à la ligne de commande. À l'intérieur d'un programme, vous utiliserez toujours la fonction **`print()`**[9] :

[9] Les fonctions seront définies en détail dans les chapitres 6 et 7 (voir page 49).

```
>>> print(msg)
Quoi de neuf ?
>>> print(n)
7
```

Remarquez la subtile différence dans les affichages obtenus avec chacune des deux méthodes. La fonction **print()** n'affiche strictement que la valeur de la variable, telle qu'elle a été encodée, alors que l'autre méthode (celle qui consiste à entrer seulement le nom de la variable) affiche aussi des apostrophes afin de vous rappeler que la variable traitée est du type « chaîne de caractères » (nous y reviendrons).

> *Dans les versions de Python antérieures à la version 3.0, le rôle de la fonction* ***print()*** *était assuré par une instruction* ***print*** *particulière, faisant d'ailleurs l'objet d'un mot réservé (voir page 14). Cette instruction s'utilisait sans parenthèses. Dans les exercices précédents, il fallait donc entrer* ***print n*** *ou* ***print msg****. Si vous essayez plus tard de faire fonctionner sous Python 3 des programmes écrits dans l'une ou l'autre version ancienne, sachez donc que vous devrez ajouter des parenthèses après chaque instruction* ***print*** *afin de convertir celle-ci en fonction (des utilitaires permettent de réaliser cela automatiquement).*
> *Dans ces mêmes versions anciennes, les chaînes de caractères étaient traitées différemment (nous en reparlerons en détail plus loin). Suivant la configuration de votre ordinateur, vous pouviez alors rencontrer quelques effets bizarres avec les chaînes contenant des caractères accentués, tels que :*
> ***>>> msg = "Mon prénom est Chimène"***
> ***>>> msg***
> ***'Mon pr\xe9nom est Chim\xe8ne'***
> *Ces bizarreries appartiennent désormais au passé, mais nous verrons plus loin qu'un programmeur digne de ce nom doit savoir de quelle manière sont encodés les caractères typographiques rencontrés dans différentes sources de données, car les normes définissant ces encodages ont changé au cours des années, et il faut connaître les techniques qui permettent de les convertir.*

Typage des variables

Sous Python, il n'est pas nécessaire d'écrire des lignes de programme spécifiques pour définir le type des variables avant de pouvoir les utiliser. Il vous suffit en effet d'assigner une valeur à un nom de variable pour que celle-ci soit *automatiquement créée avec le type qui correspond au mieux à la valeur fournie*. Dans l'exercice précédent, par exemple, les variables **n**, **msg** et **pi** ont été créées automatiquement chacune avec un type différent (« nombre entier » pour **n**, « chaîne de caractères » pour **msg**, « nombre à virgule flottante » (ou « *float* », en anglais) pour **pi**).

Ceci constitue une particularité intéressante de Python, qui le rattache à une famille particulière de langages où l'on trouve aussi par exemple *Lisp*, *Scheme*, et quelques autres. On dira à ce sujet que *le typage des variables sous Python est un typage dynamique*, par opposition au *typage statique* qui est de règle par exemple en C++ ou en Java. Dans ces langages, il faut toujours – par des instructions distinctes – d'abord déclarer (définir) le nom et le type des variables, et ensuite seulement leur assigner un contenu, lequel doit bien entendu être compatible avec le type déclaré.

Le typage statique est préférable dans le cas des langages compilés, parce qu'il permet d'optimiser l'opération de compilation (dont le résultat est un code binaire « figé »).

Le typage dynamique quant à lui permet d'écrire plus aisément des constructions logiques de niveau élevé (métaprogrammation, réflexivité), en particulier dans le contexte de la programmation orientée objet (polymorphisme). Il facilite également l'utilisation de structures de données très riches telles que les listes et les dictionnaires.

Affectations multiples

Sous Python, on peut assigner une valeur à plusieurs variables simultanément. Exemple :

```
>>> x = y = 7
>>> x
7
>>> y
7
```

On peut aussi effectuer des *affectations parallèles* à l'aide d'un seul opérateur :

```
>>> a, b = 4, 8.33
>>> a
4
>>> b
8.33
```

Dans cet exemple, les variables **a** et **b** prennent simultanément les nouvelles valeurs **4** et **8,33**.

> *Les francophones que nous sommes avons pour habitude d'utiliser la virgule comme séparateur décimal, alors que les langages de programmation utilisent toujours la convention en vigueur dans les pays de langue anglaise, c'est-à-dire le point décimal. La virgule, quant à elle, est très généralement utilisée pour séparer différents éléments (arguments, etc.) comme on le voit dans notre exemple, pour les variables elles-mêmes ainsi que pour les valeurs qu'on leur attribue.*

Exercices

2.1 Décrivez le plus clairement et le plus complètement possible ce qui se passe à chacune des trois lignes de l'exemple ci-dessous :

```
>>> largeur = 20
>>> hauteur = 5 * 9.3
>>> largeur * hauteur
930
```

2.2 Assignez les valeurs respectives 3, 5, 7 à trois variables a, b, c.
Effectuez l'opération a-b//c. Interprétez le résultat obtenu.

Opérateurs et expressions

On manipule les valeurs et les variables qui les référencent en les combinant avec des *opérateurs* pour former des *expressions*. Exemple :

```
a, b = 7.3, 12
y = 3*a + b/5
```

Dans cet exemple, nous commençons par affecter aux variables **a** et **b** les valeurs **7,3** et **12**.
Comme déjà expliqué précédemment, Python assigne automatiquement le type « réel » à la variable **a**, et le type « entier » à la variable **b**.

La seconde ligne de l'exemple consiste à affecter à une nouvelle variable **y** le résultat d'une *expression* qui combine les *opérateurs* ***** , **+** et **/** avec les *opérandes* **a**, **b**, **3** et **5**. Les opérateurs sont les symboles spéciaux utilisés pour représenter des opérations mathématiques simples, telles l'addition ou la multiplication. Les opérandes sont les valeurs combinées à l'aide des opérateurs.

Python évalue chaque expression qu'on lui soumet, aussi compliquée soit-elle, et le résultat de cette évaluation est toujours lui-même une valeur. À cette valeur, il attribue automatiquement un type, lequel dépend de ce qu'il y a dans l'expression. Dans l'exemple ci-dessus, **y** sera du type réel, parce que l'expression évaluée pour déterminer sa valeur contient elle-même au moins un réel.

Les opérateurs Python ne sont pas seulement les quatre opérateurs mathématiques de base. Nous avons déjà signalé l'existence de l'opérateur de division entière **//**. Il faut encore ajouter l'opérateur ****** pour l'exponentiation, ainsi qu'un certain nombre d'opérateurs logiques, des opérateurs agissant sur les chaînes de caractères, des opérateurs effectuant des tests d'identité ou d'appartenance, etc. Nous reparlerons de tout cela au fil des pages suivantes.

Signalons au passage la disponibilité de l'opérateur *modulo*, représenté par le caractère typographique **%**. Cet opérateur fournit *le reste de la division entière* d'un nombre par un autre. Essayez par exemple :

```
>>> 10 % 3      # (et prenez note de ce qui se passe !)
>>> 10 % 5
```

Cet opérateur vous sera très utile plus loin, notamment pour tester si un nombre **a** est divisible par un nombre **b**. Il suffira en effet de vérifier que **a % b** donne un résultat égal à zéro.

Exercice

2.3 Testez les lignes d'instructions suivantes. Décrivez ce qui se passe :

```
>>>  r , pi = 12, 3.14159
>>>  s = pi * r**2
>>>  print(s)
>>>  print(type(r), type(pi), type(s))
```

Quelle est, à votre avis, l'utilité de la *fonction* **type()** ?
(Note : les *fonctions* seront décrites en détail aux chapitres 6 et 7.)

Priorité des opérations

Lorsqu'il y a plus d'un opérateur dans une expression, l'ordre dans lequel les opérations doivent être effectuées dépend de *règles de priorité*. Sous Python, les règles de priorité sont les mêmes que celles qui vous ont été enseignées au cours de mathématique. Vous pouvez les mémoriser aisément à l'aide d'un « truc » mnémotechnique, l'acronyme *PEMDAS* :

- *P* pour *parenthèses*. Ce sont elles qui ont la plus haute priorité. Elles vous permettent donc de « forcer » l'évaluation d'une expression dans l'ordre que vous voulez.
  Ainsi **2*(3-1) = 4** , et **(1+1)**(5-2) = 8**.
- *E* pour *exposants*. Les exposants sont évalués ensuite, avant les autres opérations.
  Ainsi **2**1+1 = 3** (et non 4), et **3*1**10 = 3** (et non 59049 !).

- *M* et *D* pour *multiplication* et *division*, qui ont la même priorité. Elles sont évaluées avant *l'addition A* et *la soustraction S*, lesquelles sont donc effectuées en dernier lieu.
  Ainsi `2*3-1 = 5` (plutôt que 4), et `2/3-1 = -0.3333...` (plutôt que 1.0).
- Si deux opérateurs ont la même priorité, l'évaluation est effectuée de gauche à droite.
  Ainsi dans l'expression `59*100//60`, la multiplication est effectuée en premier, et la machine doit donc ensuite effectuer 5900//60, ce qui donne 98. Si la division était effectuée en premier, le résultat serait 59 (rappelez-vous ici que l'opérateur `//` effectue une division entière, et vérifiez en effectuant 59*(100//60)).

Composition

Jusqu'ici nous avons examiné les différents éléments d'un langage de programmation, à savoir : les *variables*, les *expressions* et les *instructions*, mais sans traiter de la manière dont nous pouvons les combiner les unes avec les autres.

Or l'une des grandes forces d'un langage de programmation de haut niveau est qu'il permet de construire des instructions complexes par assemblage de fragments divers. Ainsi par exemple, si vous savez comment additionner deux nombres et comment afficher une valeur, vous pouvez combiner ces deux instructions en une seule :

```
>>> print(17 + 3)
>>> 20
```

Cela n'a l'air de rien, mais cette fonctionnalité qui paraît si évidente va vous permettre de programmer des algorithmes complexes de façon claire et concise. Exemple :

```
>>> h, m, s = 15, 27, 34
>>> print("nombre de secondes écoulées depuis minuit = ", h*3600 + m*60 + s)
```

Attention, cependant : il y a une limite à ce que vous pouvez combiner ainsi :

Dans une expression, ce que vous placez à la gauche du signe égale doit toujours être une variable, et non une expression. Cela provient du fait que le signe égale n'a pas ici la même signification qu'en mathématique : comme nous l'avons déjà signalé, il s'agit d'un symbole d'affectation (nous plaçons un certain contenu dans une variable) et non un symbole d'égalité. Le symbole d'égalité (dans un test conditionnel, par exemple) sera évoqué un peu plus loin.

Ainsi par exemple, l'instruction `m + 1 = b` est tout à fait illégale.

Par contre, écrire `a = a + 1` est inacceptable en mathématique, alors que cette forme d'écriture est très fréquente en programmation. L'instruction `a = a + 1` signifie en l'occurrence « augmenter la valeur de la variable a d'une unité » (ou encore : « incrémenter `a` »).

Nous aurons l'occasion de revenir bientôt sur ce sujet. Mais auparavant, il nous faut encore aborder un autre concept de grande importance.

3

Contrôle du flux d'exécution

Dans notre premier chapitre, nous avons vu que l'activité essentielle d'un programmeur est la résolution de problèmes. Or, pour résoudre un problème informatique, il faut toujours effectuer une série d'actions dans un certain ordre. La description structurée de ces actions et de l'ordre dans lequel il convient de les effectuer s'appelle un algorithme.

Le « chemin » suivi par Python à travers un programme est appelé un flux d'exécution, et les constructions qui le modifient sont appelées des instructions de contrôle de flux.

Les structures de contrôle sont les groupes d'instructions qui déterminent l'ordre dans lequel les actions sont effectuées. En programmation moderne, il en existe seulement trois : la séquence[10] *et la sélection, que nous allons décrire dans ce chapitre, et la répétition que nous aborderons au chapitre suivant.*

Séquence d'instructions

Sauf mention explicite, les instructions d'un programme s'exécutent les unes après les autres, *dans l'ordre où elles ont été écrites à l'intérieur du script*.

Cette affirmation peut vous paraître banale et évidente à première vue. L'expérience montre cependant qu'un grand nombre d'erreurs sémantiques dans les programmes d'ordinateur sont la conséquence d'une mauvaise disposition des instructions. Plus vous progresserez dans l'art de la programmation, plus vous vous rendrez compte qu'il faut être extrêmement attentif à l'ordre dans lequel vous placez vos instructions les unes derrière les autres. Par exemple, dans la séquence d'instructions suivantes :

```
>>> a, b = 3, 7
>>> a = b
>>> b = a
>>> print(a, b)
```

Vous obtiendrez un résultat contraire si vous intervertissez les 2e et 3e lignes.

[10] Tel qu'il est utilisé ici, le terme de ***séquence*** désigne donc une série d'instructions qui se suivent. Nous préférerons dans la suite de cet ouvrage réserver ce terme à un concept Python précis, lequel englobe les ***chaînes de caractères***, les ***tuples*** et les ***listes*** (voir plus loin).

Python exécute normalement les instructions de la première à la dernière, sauf lorsqu'il rencontre une *instruction conditionnelle* comme l'instruction **if** décrite ci-après (nous en rencontrerons d'autres plus loin, notamment à propos des boucles de répétition). Une telle instruction va permettre au programme de suivre différents chemins suivant les circonstances.

Sélection ou exécution conditionnelle

Si nous voulons pouvoir écrire des applications véritablement utiles, il nous faut des techniques permettant d'aiguiller le déroulement du programme dans différentes directions, en fonction des circonstances rencontrées. Pour ce faire, nous devons disposer d'instructions capables de *tester une certaine condition* et de modifier le comportement du programme en conséquence.

La plus simple de ces instructions conditionnelles est l'instruction **if**. Pour expérimenter son fonctionnement, veuillez entrer dans votre éditeur Python les deux lignes suivantes :

```
>>> a = 150
>>> if (a > 100):
...
```

La première commande affecte la valeur 150 à la variable **a**. Jusqu'ici rien de nouveau.
Lorsque vous finissez d'entrer la seconde ligne, par contre, vous constatez que Python réagit d'une nouvelle manière. En effet, et à moins que vous n'ayez oublié le caractère « : » à la fin de la ligne, vous constatez que le *prompt principal* (**>>>**) est maintenant remplacé par un *prompt secondaire* constitué de trois points[11].

Si votre éditeur ne le fait pas automatiquement, vous devez à présent effectuer une tabulation (ou entrer 4 espaces) avant d'entrer la ligne suivante, de manière à ce que celle-ci soit *indentée* (c'est-à-dire en retrait) par rapport à la précédente. Votre écran devrait se présenter maintenant comme suit :

```
>>> a = 150
>>> if (a > 100):
...     print("a dépasse la centaine")
...
```

Frappez encore une fois <*Enter*>. Le programme s'exécute, et vous obtenez :

```
a dépasse la centaine
```

Recommencez le même exercice, mais avec **a = 20** en guise de première ligne : cette fois Python n'affiche plus rien.

L'expression que vous avez placée entre parenthèses après **if** est ce que nous appellerons désormais une *condition*. L'instruction **if** permet de tester la validité de cette condition. Si la condition est vraie, alors l'instruction que nous avons *indentée* après le **:** est exécutée. Si la condition est fausse, rien ne se passe. Notez que les parenthèses utilisées ici avec l'instruction **if** sont optionnelles : nous les avons utilisées pour améliorer la lisibilité. Dans d'autres langages, il se peut qu'elles soient obligatoires.

Recommencez encore, en ajoutant deux lignes comme indiqué ci-dessous. Veillez bien à ce que la quatrième ligne débute tout à fait à gauche (pas d'indentation), mais que la cinquième soit à nouveau indentée (de préférence avec un retrait identique à celui de la troisième) :

[11] Dans certaines versions de l'éditeur Python pour ***Windows***, le prompt secondaire n'apparaît pas.

```
>>> a = 20
>>> if (a > 100):
...     print("a dépasse la centaine")
... else:
...     print("a ne dépasse pas cent")
...
```

Frappez <*Enter*> encore une fois. Le programme s'exécute, et affiche cette fois :

```
a ne dépasse pas cent
```

Comme vous l'aurez certainement déjà compris, l'instruction **else** (« sinon », en anglais) permet de programmer une exécution alternative, dans laquelle le programme doit choisir entre deux possibilités. On peut faire mieux encore en utilisant aussi l'instruction **elif** (contraction de « else if ») :

```
>>> a = 0
>>> if a > 0 :
...     print("a est positif")
... elif a < 0 :
...     print("a est négatif")
... else:
...     print("a est nul")
...
```

Opérateurs de comparaison

La condition évaluée après l'instruction if peut contenir les *opérateurs de comparaison* suivants :

```
x == y       # x est égal à y
x != y       # x est différent de y
x > y        # x est plus grand que y
x < y        # x est plus petit que y
x >= y       # x est plus grand que, ou égal à y
x <= y       # x est plus petit que, ou égal à y
```

Exemple

```
>>> a = 7
>>> if (a % 2 == 0):
...     print("a est pair")
...     print("parce que le reste de sa division par 2 est nul")
... else:
...     print("a est impair")
...
```

Notez bien que l'opérateur de comparaison pour l'égalité de deux valeurs est constitué de deux signes « égale » et non d'un seul[12]. Le signe « égale » utilisé seul est un opérateur d'affectation, et non un opérateur de comparaison. Vous retrouverez le même symbolisme en *C++* et en *Java.*

Instructions composées – blocs d'instructions

La construction que vous avez utilisée avec l'instruction **if** est votre premier exemple d'**instruction composée**. Vous en rencontrerez bientôt d'autres. Sous Python, les instructions composées ont tou-

[12] Rappel : l'opérateur `%` est l'opérateur *modulo* : il calcule le reste d'une division entière. Ainsi par exemple, `a % 2` fournit le reste de la division de `a` par 2.

jours la même structure : une ligne d'en-tête terminée par un double point, suivie d'une ou de plusieurs instructions indentées sous cette ligne d'en-tête. Exemple :

```
Ligne d'en-tête:
    première instruction du bloc
    ... ...
    ... ...
    dernière instruction du bloc
```

S'il y a plusieurs instructions indentées sous la ligne d'en-tête, *elles doivent l'être exactement au même niveau* (comptez un décalage de 4 caractères, par exemple). Ces instructions indentées constituent ce que nous appellerons désormais un *bloc d'instructions*. Un bloc d'instructions est une suite d'instructions formant un ensemble logique, qui n'est exécuté que dans certaines conditions définies dans la ligne d'en-tête. Dans l'exemple du paragraphe précédent, les deux lignes d'instructions indentées sous la ligne contenant l'instruction **if** constituent un même bloc logique : ces deux lignes ne sont exécutées – toutes les deux – que si la condition testée avec l'instruction **if** se révèle vraie, c'est-à-dire si le reste de la division de **a** par 2 est nul.

Instructions imbriquées

Il est parfaitement possible d'imbriquer les unes dans les autres plusieurs instructions composées, de manière à réaliser des structures de décision complexes. Exemple :

```
if embranchement == "vertébrés":                    # 1
    if classe == "mammifères":                      # 2
        if ordre == "carnivores":                   # 3
            if famille == "félins":                 # 4
                print("c'est peut-être un chat")    # 5
        print("c'est en tous cas un mammifère")     # 6
    elif classe == "oiseaux":                       # 7
        print("c'est peut-être un canari")          # 8
print("la classification des animaux est complexe") # 9
```

Analysez cet exemple. Ce fragment de programme n'imprime la phrase « c'est peut-être un chat » que dans le cas où les quatre premières conditions testées sont vraies.

Pour que la phrase « c'est en tous cas un mammifère » soit affichée, il faut et il suffit que les deux premières conditions soient vraies. L'instruction d'affichage de cette phrase (ligne 4) se trouve en effet au même niveau d'indentation que l'instruction : **if ordre == "carnivores"**: (ligne 3). Les deux font donc partie d'un même bloc, lequel est entièrement exécuté si les conditions testées aux lignes 1 et 2 sont vraies.

Pour que la phrase « c'est peut-être un canari » soit affichée, il faut que la variable **embranchement** contienne « vertébrés », et que la variable **classe** contienne « oiseaux ».

Quant à la phrase de la ligne 9, elle est affichée dans tous les cas, parce qu'elle fait partie du même bloc d'instructions que la ligne 1.

Quelques règles de syntaxe Python

Tout ce qui précède nous amène à faire le point sur quelques règles de syntaxe :

Les limites des instructions et des blocs sont définies par la mise en page

Dans de nombreux langages de programmation, il faut terminer chaque ligne d'instructions par un caractère spécial (souvent le point-virgule). Sous Python, c'est le caractère de fin de ligne[13] qui joue ce rôle. (Nous verrons plus loin comment outrepasser cette règle pour étendre une instruction complexe sur plusieurs lignes.) On peut également terminer une ligne d'instructions par un commentaire. Un commentaire Python commence toujours par le caractère spécial **#**. Tout ce qui est inclus entre ce caractère et le saut à la ligne suivant est complètement ignoré par le compilateur.

Dans la plupart des autres langages, un bloc d'instructions doit être délimité par des symboles spécifiques (parfois même par des instructions, telles que **begin** et **end**). En *C++* et en *Java*, par exemple, un bloc d'instructions doit être délimité par des accolades. Cela permet d'écrire les blocs d'instructions les uns à la suite des autres, sans se préoccuper ni d'indentation ni de sauts à la ligne, mais cela peut conduire à l'écriture de programmes confus, difficiles à relire pour les pauvres humains que nous sommes. On conseille donc à tous les programmeurs qui utilisent ces langages de se servir *aussi* des sauts à la ligne et de l'indentation pour bien délimiter visuellement les blocs.

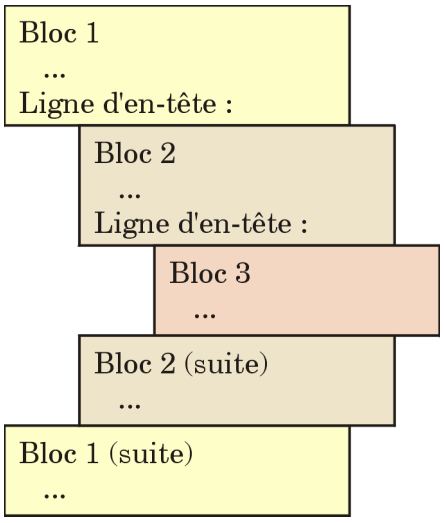
Avec Python, vous *devez* utiliser les sauts à la ligne et l'indentation, mais en contrepartie vous n'avez pas à vous préoccuper d'autres symboles délimiteurs de blocs. En définitive, Python vous force donc à écrire du code lisible, et à prendre de bonnes habitudes que vous conserverez lorsque vous utiliserez d'autres langages.

Instruction composée : en-tête, double point, bloc d'instructions indenté

Nous aurons de nombreuses occasions d'approfondir le concept de « bloc d'instructions » et de faire des exercices à ce sujet dès le chapitre suivant.

Le schéma ci-contre en résume le principe.

- Les blocs d'instructions sont toujours associés à une ligne d'en-tête contenant une instruction bien spécifique (if, elif, else, while, def, etc.) *se terminant par un double point*.
- Les blocs sont *délimités par l'indentation* : toutes les lignes d'un même bloc doivent être indentées exactement de la même manière (c'est-à-dire décalées vers la droite d'un même nombre d'espaces). Le nombre d'espaces à utiliser pour l'indentation est quelconque, mais la plupart des programmeurs utilisent des multiples de 4.
- Notez que le code du bloc le plus externe (bloc 1) ne peut pas lui-même être écarté de la marge de gauche (il n'est imbriqué dans rien).



[13] Ce caractère n'apparaît ni à l'écran, ni sur les listings imprimés. Il est cependant bien présent, à un point tel qu'il fait même problème dans certains cas, parce qu'il n'est pas encodé de la même manière par tous les systèmes d'exploitation. Nous en reparlerons plus loin, à l'occasion de notre étude des fichiers texte (page 114).

> ***Important***
> *Vous pouvez aussi indenter à l'aide de tabulations, mais alors vous devrez faire très attention à ne pas utiliser tantôt des espaces, tantôt des tabulations pour indenter les lignes d'un même bloc. En effet, même si le résultat paraît identique à l'écran, espaces et tabulations sont des codes binaires distincts : Python considérera donc que ces lignes indentées différemment font partie de blocs différents. Il peut en résulter des erreurs difficiles à déboguer.*
> *En conséquence, la plupart des programmeurs préfèrent se passer des tabulations. Si vous utilisez un éditeur de texte « intelligent », vous avez tout intérêt à activer son option « Remplacer les tabulations par des espaces ».*

Les espaces et les commentaires sont normalement ignorés

À part ceux qui servent à l'indentation, en début de ligne, les espaces placés à l'intérieur des instructions et des expressions sont presque toujours ignorés (sauf s'ils font partie d'une chaîne de caractères). Il en va de même pour les commentaires : ceux-ci commencent toujours par un caractère dièse (#) et s'étendent jusqu'à la fin de la ligne courante.

4

Instructions répétitives

L'une des tâches que les machines font le mieux est la répétition sans erreur de tâches identiques. Il existe bien des méthodes pour programmer ces tâches répétitives. Nous allons commencer par l'une des plus fondamentales : la boucle de répétition construite autour de l'instruction while.

Réaffectation

Nous ne l'avions pas encore signalé explicitement : il est permis de ré-affecter une nouvelle valeur à une même variable, autant de fois qu'on le souhaite.

L'effet d'une ré-affectation est de remplacer l'ancienne valeur d'une variable par une nouvelle.

```
>>> altitude = 320
>>> print(altitude)
320
>>> altitude = 375
>>> print(altitude)
375
```

Ceci nous amène à attirer une nouvelle fois votre attention sur le fait que le symbole égale utilisé sous Python pour réaliser une affectation ne doit en aucun cas être confondu avec un symbole d'égalité tel qu'il est compris en mathématique. Il est tentant d'interpréter l'instruction **altitude = 320** comme une affirmation d'égalité, mais ce n'en est pas une !

- Premièrement, l'égalité est *commutative*, alors que l'affectation ne l'est pas. Ainsi, en mathématique, les écritures **a = 7** et **7 = a** sont équivalentes, alors qu'une instruction de programmation telle que **375 = altitude** serait illégale.
- Deuxièmement, l'égalité est *permanente*, alors que l'affectation peut être remplacée comme nous venons de le voir. Lorsqu'en mathématique, nous affirmons une égalité telle que a = b au début d'un raisonnement, alors a continue à être égal à b durant tout le développement qui suit.
  En programmation, une première instruction d'affectation peut rendre égales les valeurs de deux variables, et une instruction ultérieure en changer ensuite l'une ou l'autre. Exemple :

```
>>> a = 5
>>> b = a         # a et b contiennent des valeurs égales
>>> b = 2         # a et b sont maintenant différentes
```

Rappelons ici que Python permet d'affecter leurs valeurs à plusieurs variables simultanément :

```
>>> a, b, c, d = 3, 4, 5, 7
```

Cette fonctionnalité de Python est bien plus intéressante encore qu'elle n'en a l'air à première vue. Supposons par exemple que nous voulions maintenant échanger les valeurs des variables **a** et **c** (actuellement, **a** contient la valeur **3**, et **c** la valeur **5** ; nous voudrions que ce soit l'inverse). Comment faire ?

Exercice

4.1 Écrivez les lignes d'instructions nécessaires pour obtenir ce résultat.

À la suite de l'exercice proposé ci-dessus, vous aurez certainement trouvé une méthode, et un professeur vous demanderait certainement de la commenter en classe. Comme il s'agit d'une opération courante, les langages de programmation proposent souvent des raccourcis pour l'effectuer (par exemple des instructions spécialisées, telle l'instruction SWAP du langage *Basic*). Sous Python, *l'affectation parallèle* permet de programmer l'échange d'une manière particulièrement élégante :

```
>>> a, b = b, a
```

(On pourrait bien entendu échanger d'autres variables en même temps, dans la même instruction.)

Répétitions en boucle – L'instruction while

En programmation, on appelle *boucle* un système d'instructions qui permet de répéter un certain nombre de fois (voire indéfiniment) toute une série d'opérations. Python propose deux instructions particulières pour construire des boucles : l'instruction **for … in …** , très puissante, que nous étudierons au chapitre 10, et l'instruction **while** que nous allons découvrir tout de suite.

Veuillez donc entrer les commandes ci-dessous :

```
>>> a = 0
>>> while (a < 7):        # (n'oubliez pas le double point !)
...     a = a + 1         # (n'oubliez pas l'indentation !)
...     print(a)
```

Frappez encore une fois <*Enter*>.

Que se passe-t-il ?

Avant de lire les commentaires de la page suivante, prenez le temps d'ouvrir votre cahier et d'y noter cette série de commandes. Décrivez aussi le résultat obtenu, et essayez de l'expliquer de la manière la plus détaillée possible.

Commentaires

Le mot **while** signifie « tant que » en anglais. Cette instruction utilisée à la seconde ligne indique à Python qu'il lui faut *répéter continuellement le bloc d'instructions qui suit, tant que* le contenu de la variable **a** reste inférieur à 7.

Comme l'instruction **if** abordée au chapitre précédent, l'instruction **while** amorce une *instruction composée*. Le double point à la fin de la ligne introduit le bloc d'instructions à répéter, lequel doit obligatoirement se trouver en retrait. Comme vous l'avez appris au chapitre précédent, toutes les ins-

tructions d'un même bloc doivent être indentées exactement au même niveau (c'est-à-dire décalées à droite d'un même nombre d'espaces).

Nous avons ainsi construit notre première *boucle de programmation*, laquelle répète un certain nombre de fois le bloc d'instructions indentées. Voici comment cela fonctionne :

- Avec l'instruction **while**, Python commence par évaluer la validité de la *condition* fournie entre parenthèses (celles-ci sont optionnelles, nous ne les avons utilisées que pour clarifier notre explication).
- Si la condition se révèle fausse, alors tout le bloc qui suit est ignoré et l'exécution du programme se termine[14].
- Si la codition est vraie, alors Python exécute tout le bloc d'instructions constituant *le corps de la boucle*, c'est-à-dire :
  - l'instruction **a = a + 1** qui incrémente d'une unité le contenu de la variable **a** (ce qui signifie que l'on affecte à la variable **a** une nouvelle valeur, qui est égale à la valeur précédente augmentée d'une unité).
  - l'appel de la fonction **print()** pour afficher la valeur courante de la variable **a**.
- lorsque ces deux instructions ont été exécutées, nous avons assisté à une première *itération*, et le programme boucle, c'est-à-dire que l'exécution reprend à la ligne contenant l'instruction **while**. La condition qui s'y trouve est à nouveau évaluée, et ainsi de suite.
  Dans notre exemple, si la condition **a < 7** est encore vraie, le corps de la boucle est exécuté une nouvelle fois et le bouclage se poursuit.

Remarques

- La variable évaluée dans la condition doit exister au préalable (il faut qu'on lui ait déjà affecté au moins une valeur).
- Si la condition est fausse au départ, le corps de la boucle n'est jamais exécuté.
- Si la condition reste toujours vraie, alors le corps de la boucle est répété indéfiniment (tout au moins tant que Python lui-même continue à fonctionner). Il faut donc veiller à ce que le corps de la boucle contienne au moins une instruction qui change la valeur d'une variable intervenant dans la condition évaluée par while, de manière à ce que cette condition puisse devenir fausse et la boucle se terminer.
  Exemple de boucle sans fin (à éviter !) :

```
>>> n = 3
>>> while n < 5:
...     print("hello !")
```

Élaboration de tables

Recommencez à présent le premier exercice, mais avec la petite modification ci-dessous :

[14] … du moins dans cet exemple. Nous verrons un peu plus loin qu'en fait l'exécution continue avec la première instruction qui suit le bloc indenté, et qui fait partie du même bloc que l'instruction **while** elle-même.

```
>>> a = 0
>>> while a < 12:
...     a = a +1
...     print(a , a**2 , a**3)
```

Vous devriez obtenir la liste des carrés et des cubes des nombres de 1 à 12.

Notez au passage que la fonction **print()** permet d'afficher plusieurs expressions l'une à la suite de l'autre sur la même ligne : il suffit de les séparer par des virgules. Python insère automatiquement un espace entre les éléments affichés.

Construction d'une suite mathématique

Le petit programme ci-dessous permet d'afficher les dix premiers termes d'une suite appelée « *Suite de Fibonacci* ». Il s'agit d'une suite de nombres dont chaque terme est égal à la somme des deux termes qui le précèdent. Analysez ce programme (qui utilise judicieusement l'affectation parallèle) et décrivez le mieux possible le rôle de chacune des instructions.

```
>>> a, b, c = 1, 1, 1
>>> while c < 11 :
...     print(b, end =" ")
...     a, b, c = b, a+b, c+1
```

Lorsque vous lancez l'exécution de ce programme, vous obtenez :

```
1 2 3 5 8 13 21 34 55 89
```

Les termes de la suite de Fibonacci sont affichés sur la même ligne. Vous obtenez ce résultat grâce au second argument `end =" "` fourni à la fonction **print()**. Par défaut, la fonction **print()** ajoute en effet un caractère de saut à la ligne à toute valeur qu'on lui demande d'afficher. L'argument `end =" "` signifie que vous souhaitez remplacer le saut à la ligne par un simple espace. Si vous supprimez cet argument, les nombres seront affichés les uns en-dessous des autres.

Dans vos programmes futurs, vous serez très souvent amenés à mettre au point des boucles de répétition comme celle que nous analysons ici. Il s'agit d'une question essentielle, que vous devez apprendre à maîtriser parfaitement. Soyez sûr que vous y arriverez progressivement, à force d'exercices.

Lorsque vous examinez un problème de cette nature, vous devez considérer les lignes d'instruction, bien entendu, mais surtout décortiquer *les états successifs des différentes variables* impliquées dans la boucle. Cela n'est pas toujours facile, loin de là. Pour vous aider à y voir plus clair, prenez la peine de dessiner sur papier une table d'états similaire à celle que nous reproduisons ci-dessous pour notre programme « suite de Fibonacci » :

Variables	a	b	c
Valeurs initiales	1	1	1
Valeurs prises successivement, au cours des itérations	1 2 3 5 …	2 3 5 8 …	2 3 4 5 …
Expression de remplacement	b	a+b	c+1

Dans une telle table, on effectue en quelque sorte « à la main » le travail de l'ordinateur, en indiquant ligne par ligne les valeurs que prendront chacune des variables au fur et à mesure des itérations successives. On commence par inscrire en haut du tableau les noms des variables concernées. Sur la ligne suivante, les valeurs initiales de ces variables (valeurs qu'elles possèdent avant le démarrage de la boucle). Enfin, tout en bas du tableau, les expressions utilisées dans la boucle pour modifier l'état de chaque variable à chaque itération.

On remplit alors quelques lignes correspondant aux premières itérations. Pour établir les valeurs d'une ligne, il suffit d'appliquer à celles de la ligne précédente, l'expression de remplacement qui se trouve en bas de chaque colonne. On vérifie ainsi que l'on obtient bien la suite recherchée. Si ce n'est pas le cas, il faut essayer d'autres expressions de remplacement.

Exercices

4.2 Écrivez un programme qui affiche les 20 premiers termes de la table de multiplication par 7.

4.3 Écrivez un programme qui affiche une table de conversion de sommes d'argent exprimées en euros, en dollars canadiens. La progression des sommes de la table sera « géométrique », comme dans l'exemple ci-dessous :

```
1 euro(s) = 1.65 dollar(s)
2 euro(s) = 3.30 dollar(s)
4 euro(s) = 6.60 dollar(s)
8 euro(s) = 13.20 dollar(s)
```

etc. (S'arrêter à 16384 euros.)

4.4 Écrivez un programme qui affiche une suite de 12 nombres dont chaque terme soit égal au triple du terme précédent.

Premiers scripts, ou comment conserver nos programmes

Jusqu'à présent, vous avez toujours utilisé Python *en mode interactif* (c'est-à-dire que vous avez à chaque fois entré les commandes directement dans l'interpréteur, sans les sauvegarder au préalable dans un fichier). Cela vous a permis d'apprendre très rapidement les bases du langage, par expérimentation directe. Cette façon de faire présente toutefois un gros inconvénient : toutes les séquences d'instructions que vous avez écrites disparaissent irrémédiablement dès que vous fermez l'interpréteur. Avant de poursuivre plus avant votre étude, il est donc temps que vous appreniez à sauvegarder vos programmes dans des fichiers, sur disque dur ou clef USB, de manière à pouvoir les retravailler par étapes successives, les transférer sur d'autres machines, etc.

Pour ce faire, vous allez désormais rédiger vos séquences d'instructions dans un *éditeur de texte* quelconque (par exemple *Kate*, *Gedit*, *Geany*... sous *Linux*, *Wordpad, Geany, Komodo editor*... sous *Windows*, ou encore l'éditeur incorporé dans l'interface de développement *IDLE* qui fait partie de la distribution de Python pour Windows). Ainsi vous écrirez *un script*, que vous pourrez ensuite sauvegarder, modifier, copier, etc. comme n'importe quel autre texte traité par ordinateur[15].

Par la suite, lorsque vous voudrez tester l'exécution de votre programme, il vous suffira de lancer l'interpréteur Python en lui fournissant (comme argument) le nom du fichier qui contient le script. Par exemple, si vous avez placé un script dans un fichier nommé « MonScript », il suffira d'entrer la commande suivante dans une fenêtre de terminal pour que ce script s'exécute :

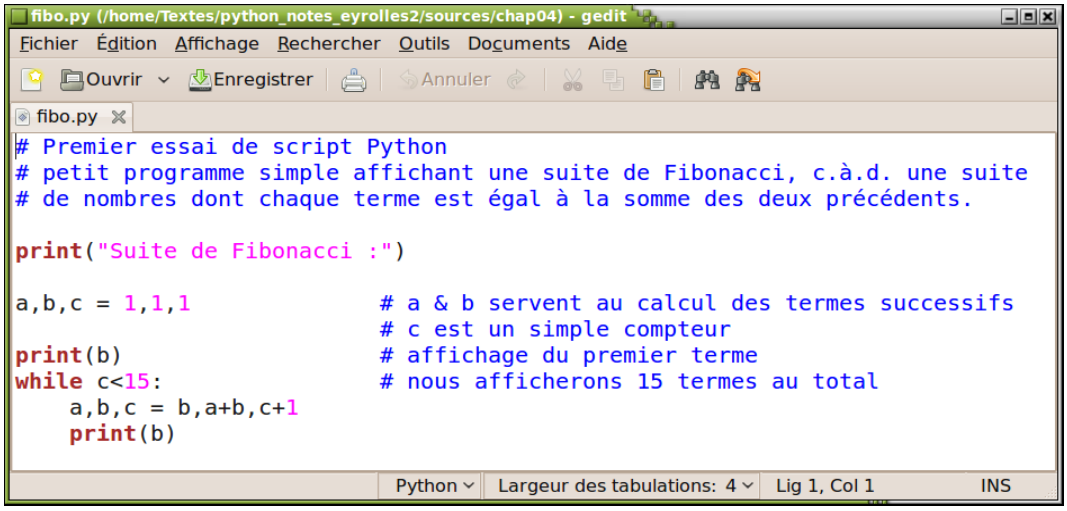
```
python3 MonScript [16]
```

Pour faire mieux encore, veillez à choisir pour votre fichier un nom qui se termine par l'extension **.py**

Si vous respectez cette convention, vous pourrez aussi lancer l'exécution du script, simplement en cliquant sur son nom ou sur l'icône correspondante dans le gestionnaire de fichiers (c'est-à-dire l'explorateur, sous *Windows*, ou bien *Nautilus*, *Konqueror*... sous *Linux*).

Ces gestionnaires graphiques « savent » en effet qu'ils doivent lancer l'interpréteur Python chaque fois que leur utilisateur essaye d'ouvrir un fichier dont le nom se termine par **.py** (cela suppose bien entendu qu'ils aient été correctement configurés). La même convention permet en outre aux éditeurs « intelligents » de reconnaître automatiquement les scripts Python, et d'adapter leur coloration syntaxique en conséquence.

La figure ci-après illustre l'utilisation de l'éditeur *Gedit* sous *Linux (Ubuntu)* pour écrire un script :



[15] Il serait parfaitement possible d'utiliser un système de traitement de texte, à la condition d'effectuer la sauvegarde sous un format « texte pur » (sans balises de mise en page). Il est cependant préférable d'utiliser un véritable éditeur « intelligent » tel que *Gedit*, *Geany*, ou *IDLE*, muni d'une fonction de coloration syntaxique pour Python, qui vous aide à éviter les fautes de syntaxe. Avec *IDLE*, suivez le menu : File → New window (ou tapez <Ctrl-N>) pour ouvrir une nouvelle fenêtre dans laquelle vous écrirez votre script. Pour l'exécuter, il vous suffira (après sauvegarde), de suivre le menu : Edit → Run script (ou de taper <Ctrl-F5>).

[16] Si l'interpréteur Python 3 a été installé sur votre machine comme interpréteur Python par défaut, vous devriez pouvoir aussi entrer tout simplement : **python MonScript** . Mais attention : si plusieurs versions de Python sont présentes, il se peut que cette commande active plutôt une version antérieure (Python 2.x).

Un script Python contiendra des séquences d'instructions identiques à celles que vous avez expérimentées jusqu'à présent. Puisque ces séquences sont destinées à être conservées et relues plus tard par vous-même ou par d'autres, *il vous est très fortement recommandé d'expliciter vos scripts le mieux possible, en y incorporant de nombreux commentaires*. La principale difficulté de la programmation consiste en effet à mettre au point des algorithmes corrects. Afin que ces algorithmes puissent être vérifiés, corrigés, modifiés, etc. dans de bonnes conditions, il est essentiel que leur auteur les décrive le plus complètement et le plus clairement possible. Et le meilleur emplacement pour cette description est le corps même du script (ainsi elle ne peut pas s'égarer).

> *Un bon programmeur veille toujours à insérer un grand nombre de commentaires dans ses scripts. En procédant ainsi, non seulement il facilite la compréhension de ses algorithmes pour d'autres lecteurs éventuels, mais encore il se force lui-même à avoir les idées plus claires.*

On peut insérer des commentaires quelconques à peu près n'importe où dans un script. Il suffit de les faire précéder d'un caractère #. Lorsqu'il rencontre ce caractère, l'interpréteur Python ignore tout ce qui suit, jusqu'à la fin de la ligne courante.

Comprenez bien qu'il est important d'inclure des commentaires *au fur et à mesure de l'avancement de votre travail de programmation*. N'attendez pas que votre script soit terminé pour les ajouter « après coup ». Vous vous rendrez progressivement compte qu'un programmeur passe *énormément de temps* à relire son propre code (pour le modifier, y rechercher des erreurs, etc.). Cette relecture sera grandement facilitée si le code comporte de nombreuses explications et remarques.

Ouvrez donc un éditeur de texte, et rédigez le script ci-dessous :

```
# Premier essai de script Python
# petit programme simple affichant une suite de Fibonacci, c.à.d. une suite
# de nombres dont chaque terme est égal à la somme des deux précédents.

a, b, c = 1, 1, 1           # a & b servent au calcul des termes successifs
                            # c est un simple compteur
print(b)                    # affichage du premier terme
while c<15:                 # nous afficherons 15 termes au total
    a, b, c = b, a+b, c+1
    print(b)
```

Afin de vous montrer tout de suite le bon exemple, nous commençons ce script par trois lignes de commentaires, qui contiennent une courte description de la fonctionnalité du programme. Prenez l'habitude de faire de même dans vos propres scripts.

Certaines lignes de code sont également documentées. Si vous procédez comme nous l'avons fait, c'est-à-dire en insérant des commentaires à la droite des instructions correspondantes, veillez à les écarter suffisamment de celles-ci, afin de ne pas gêner leur lisibilité.

Lorsque vous avez bien vérifié votre texte, sauvegardez-le et exécutez-le.

> *Bien que ce ne soit pas indispensable, nous vous recommandons une fois encore de choisir pour vos scripts des noms de fichiers se terminant par l'extension* ***.py****. Cela aide beaucoup à les identifier comme tels dans un répertoire. Les gestionnaires graphiques de fichiers (explorateur Windows, Nautilus, Konqueror) se servent d'ailleurs de cette extension pour leur associer une icône spécifique. Évitez cependant de choisir des noms*

> *qui risqueraient d'être déjà attribués à des modules python existants : des noms tels que* **`math.py`** *ou* **`tkinter.py`***, par exemple, sont à proscrire !*

Si vous travaillez en mode texte sous *Linux*, ou dans une fenêtre *MS-DOS*, vous pouvez exécuter votre script à l'aide de la commande **`python3`** suivie du nom du script. Si vous travaillez en mode graphique sous *Linux*, vous pouvez ouvrir une fenêtre de terminal et faire la même chose :

```
python3 monScript.py
```

Dans un gestionnaire graphique de fichiers, vous pouvez en principe lancer l'exécution de votre script en (double) cliquant sur l'icône correspondante. Ceci ne pourra cependant fonctionner que si c'est bien Python 3 qui a été désigné comme interpréteur par défaut pour les fichiers comportant l'extension **`.py`** (des problèmes peuvent en effet apparaître si plusieurs versions de Python sont installées sur votre machine – voyez votre professeur ou votre gourou local pour détailler ces questions).

Si vous travaillez avec *IDLE*, vous pouvez lancer l'exécution du script en cours d'édition, directement à l'aide de la combinaison de touches <Ctrl-F5>. Dans d'autres environnements de travail spécifiquement dédiés à Python, tels que *Geany*, vous trouverez également des icônes et/ou des raccourcis clavier pour lancer l'exécution (il s'agit souvent de la touche <F5>). Consultez votre professeur ou votre gourou local concernant les autres possibilités de lancement éventuelles sur différents systèmes d'exploitation.

Problèmes éventuels liés aux caractères accentués

Si vous avez rédigé votre script avec un éditeur récent (tels ceux que nous avons déjà indiqués), le script décrit ci-dessus devrait s'exécuter sans problème avec la version actuelle de Python 3. Si votre logiciel est ancien ou mal configuré, il se peut que vous obteniez un message d'erreur similaire à celui-ci :

```
File "fibo2.py", line 2
SyntaxError: Non-UTF-8 code starting with '\xe0' in file fibo2.py on line 2, but no
encoding declared; see http://python.org/dev/peps/pep-0263/ for details
```

Ce message vous indique que le script contient des caractères typographiques encodés suivant une norme ancienne (vraisemblablement la norme ISO-8859-1 ou Latin-1).

Nous détaillerons les différentes normes d'encodage plus loin dans ce livre. Pour l'instant, il vous suffit de savoir que vous devez dans ce cas :

- Soit reconfigurer votre éditeur de textes pour qu'il encode les caractères en Utf-8 (ou vous procurer un autre éditeur fonctionnant suivant cette norme). C'est la meilleure solution, car ainsi vous serez certain à l'avenir de travailler en accord avec les conventions de standardisation actuelles, qui finiront tôt ou tard par remplacer partout les anciennes.
- Soit inclure le pseudo-commentaire suivant au début de tous vos scripts (obligatoirement à la 1e ou à la 2e ligne) :
  ```
  # -*- coding:Latin-1 -*-
  ```
  Le pseudo-commentaire ci-dessus indique à Python que vous utilisez dans votre script le jeu de caractères accentués *ASCII étendu* correspondant aux principales langues de l'Europe occidentale

(français, allemand, portugais, etc.), encodé sur un seul octet suivant la norme *ISO-8859-1*, laquelle est souvent désignée aussi par l'étiquette *Latin-1* .

Python peut traiter correctement les caractères encodés suivant toute une série de normes. Il faut donc lui signaler celle que vous utilisez à l'aide d'un pseudo-commentaire en début de script. Sans cette indication, Python considère que vos scripts ont été encodés en Utf-8[17], suivant la nouvelle norme *Unicode*, qui a été mise au point pour standardiser la représentation numérique de tous les caractères spécifiques des différentes langues mondiales, ainsi que les symboles mathématiques, scientifiques, etc. Il existe plusieurs représentations ou *encodages* de cette norme, nous approfondirons cette question plus loin[18]. Pour le moment, il vous suffit de savoir que l'encodage le plus répandu sur les ordinateurs récents est *Utf-8*. Dans ce système, les caractères standard (*ASCII*) sont encore encodés sur un seul octet, ce qui assure une certaine compatibilité avec l'ancienne norme d'encodage *Latin-1*, mais les autres caractères (parmi lesquels nos caractères accentués) peuvent être encodés sur 2, 3, ou même parfois 4 octets.

Nous apprendrons comment gérer et convertir ces différents encodages, lorsque nous étudierons plus en détail le traitement des fichiers texte (au chapitre 9).

Exercices

4.5 Écrivez un programme qui calcule le volume d'un parallélépipède rectangle dont sont fournis au départ la largeur, la hauteur et la profondeur.

4.6 Écrivez un programme qui convertit un nombre entier de secondes fourni au départ en un nombre d'années, de mois, de jours, de minutes et de secondes (utilisez l'opérateur modulo : **%**).

4.7 Écrivez un programme qui affiche les 20 premiers termes de la table de multiplication par 7, en signalant au passage (à l'aide d'une astérisque) ceux qui sont des multiples de 3.
Exemple : 7 14 21 * 28 35 42 * 49 ...

4.8 Écrivez un programme qui calcule les 50 premiers termes de la table de multiplication par 13, mais n'affiche que ceux qui sont des multiples de 7.

4.9 Écrivez un programme qui affiche la suite de symboles suivante :

```
*
**
***
****
*****
******
*******
```

[17] Dans les versions de Python antérieures à la version 3.0, l'encodage par défaut était ASCII. Il fallait donc toujours préciser en début de script les autres encodages (y compris Utf-8).

[18] Voir page 127

5

Principaux types de données

Dans le chapitre 2, nous avons déjà manipulé des données de différents types : des nombres entiers ou réels, et des chaînes de caractères. Il est temps à présent d'examiner d'un peu plus près ces types de données, et également de vous en faire découvrir d'autres.

Les données numériques

Dans les exercices réalisés jusqu'à présent, nous avons déjà utilisé des données de deux types : les nombres *entiers* ordinaires et les nombres *réels* (aussi appelés nombres *à virgule flottante*). Tâchons de mettre en évidence les caractéristiques (et les limites) de ces concepts.

Le type integer

Supposons que nous voulions modifier légèrement notre précédent exercice sur la suite de Fibonacci, de manière à obtenir l'affichage d'un plus grand nombre de termes. A priori, il suffit de modifier la condition de bouclage, dans la deuxième ligne. Avec `while c <50:`, nous devrions obtenir quarante-neuf termes. Modifions donc légèrement l'exercice, de manière à afficher aussi le type de la variable principale :

```
>>> a, b, c = 1, 1, 1
>>> while c <50:
        print(c, ":", b, type(b))
        a, b, c = b, a+b, c+1
...
...
...  (affichage des 43 premiers termes)
...
44 : 1134903170 <class 'int'>
45 : 1836311903 <class 'int'>
46 : 2971215073 <class 'int'>
47 : 4807526976 <class 'int'>
48 : 7778742049 <class 'int'>
49 : 12586269025 <class 'int'>
```

Que pouvons-nous constater ? Il semble que Python soit capable de traiter des nombres entiers de taille illimitée. La fonction **type()** nous permet de vérifier à chaque itération que le type de la variable **b** reste bien en permanence de ce type.

L'exercice que nous venons de réaliser pourrait cependant intriguer ceux d'entre vous qui s'interrogent sur la représentation interne des nombres dans un ordinateur. Vous savez probablement en effet que le cœur de celui-ci est constitué par un circuit intégré électronique (une puce de silicium) à très haut degré d'intégration, qui peut effectuer plus d'un milliard d'opérations en une seule seconde, mais seulement sur des nombres binaires de taille limitée : 32 bits actuellement[19]. Or, la gamme de valeurs décimales qu'il est possible d'encoder sous forme de nombres binaires de 32 bits s'étend de -2147483648 à +2147483647.

Les opérations effectuées sur des entiers compris entre ces deux limites sont donc toujours très rapides, parce que le processeur est capable de les traiter directement. En revanche, lorsqu'il est question de traiter des nombres entiers plus grands, ou encore des nombres réels (nombres « à virgule flottante »), les logiciels que sont les interpréteurs et compilateurs doivent effectuer un gros travail de codage/décodage, afin de ne présenter en définitive au processeur que des opérations binaires sur des nombres entiers, de 32 bits au maximum.

Vous n'avez pas à vous préoccuper de ces considérations techniques. Lorsque vous lui demandez de traiter des entiers quelconques, Python les transmet au processeur sous la forme de nombres binaires de 32 bits chaque fois que c'est possible, afin d'optimiser la vitesse de calcul et d'économiser l'espace mémoire. Lorsque les valeurs à traiter sont des nombres entiers se situant au-delà des limites indiquées plus haut, leur encodage dans la mémoire de l'ordinateur devient plus complexe, et leur traitement par le processeur nécessite alors plusieurs opérations successives. Tout cela se fait automatiquement, sans que vous n'ayez à vous en soucier[20].

Vous pouvez donc effectuer avec Python des calculs impliquant des valeurs entières comportant un nombre de chiffres significatifs quelconque. Ce nombre n'est limité en effet que par la taille de la mémoire disponible sur l'ordinateur utilisé. Il va de soi cependant que les calculs impliquant de très grands nombres devront être décomposés par l'interpréteur en calculs multiples sur des nombres plus simples, ce qui pourra nécessiter un temps de traitement considérable dans certains cas.

Exemple :

```
>>> a, b, c = 3, 2, 1
>>> while c < 15:
        print(c, ": ", b)
        a, b, c = b, a*b, c+1

1 :  2
2 :  6
3 :  12
4 :  72
5 :  864
6 :  62208
7 :  53747712
8 :  3343537668096
9 :  179707499645975396352
10 :  600858794305667322270155425185792
11 :  107978831564966913814384922944738457859243070439030784
12 :  648800305446607527907368373691049776950010342842280428918276494561862 34
```

[19] La plupart des ordinateurs de bureau actuels contiennent un microprocesseur à registres de 32 bits (même s'il s'agit d'un modèle dual core. Les processeurs 64 bits seront cependant bientôt monnaie courante.

[20] Les précédentes versions de Python disposaient de deux types d'entiers : **`integer`** et **`long integer`**, mais la conversion entre ces deux types est devenue automatique dès la version 2.2.

```
582611607420928
13 :  700566989011183200292376413995762169216245450579726979173836923132717 54
883621235064434673400268965204696103008832506249008437424702378475 52
14 :  454528076456265799856362940482493512051682398707229461514016556556583 98
642227616335815123825782460196980206141536747116094173550514227947953005917 00
969504226930790382476340558291752968319462245039335017547760330040127583682 56
>>>
```

Dans l'exemple ci-dessus, la valeur des nombres affichés augmente très rapidement, car chacun d'eux est égal au produit des deux termes précédents. Bien évidemment, vous pouvez continuer cette suite mathématique si vous voulez. La progression continue avec des nombres gigantesques, mais la vitesse de calcul diminue au fur et à mesure.

> *Les entiers de valeur comprise entre les deux limites indiquées plus haut occupent chacun 32 bits dans la mémoire de l'ordinateur. Les très grands entiers occupent une place variable, en fonction de leur taille.*

Le type float

Vous avez déjà rencontré précédemment cet autre type de donnée numérique : le type « nombre réel », ou « nombre à virgule flottante », désigné en anglais par l'expression *floating point number*, et que pour cette raison on appellera type **float** sous Python.

Ce type autorise les calculs sur de très grands ou très petits nombres (données scientifiques, par exemple), avec un degré de précision constant.

Pour qu'une donnée numérique soit considérée par Python comme étant du type **float**, il suffit qu'elle contienne dans sa formulation un élément tel qu'un point décimal ou un exposant de 10.

Par exemple, les données :

```
       3.14      10.      .001      1e100      3.14e-10
```

sont automatiquement interprétées par Python comme étant du type ***float***.

Essayons donc ce type de données dans un nouveau petit programme (inspiré du précédent) :

```
>>> a, b, c = 1., 2., 1            # => a et b seront du type 'float'
>>> while c <18:
...     a, b, c = b, b*a, c+1
...     print(b)

2.0
4.0
8.0
32.0
256.0
8192.0
2097152.0
17179869184.0
3.6028797019e+16
6.18970019643e+26
2.23007451985e+43
1.38034926936e+70
3.07828173409e+113
4.24910394253e+183
1.30799390526e+297
```

```
Inf
Inf
```

Comme vous l'aurez certainement bien compris, nous affichons cette fois encore une série dont les termes augmentent extrêmement vite, chacun d'eux étant égal au produit des deux précédents. Au neuvième terme, Python passe automatiquement à la notation scientifique (« e+n » signifie en fait : « fois dix à l'exposant n »). Après le quinzième terme, nous assistons à nouveau à un dépassement de capacité (sans message d'erreur) : les nombres vraiment trop grands sont tout simplement notés « inf » (pour « infini »).

Le type *float* utilisé dans notre exemple permet de manipuler des nombres (positifs ou négatifs) compris entre 10^{-308} et 10^{308} avec une précision de 12 chiffres significatifs. Ces nombres sont encodés d'une manière particulière sur 8 octets (64 bits) dans la mémoire de la machine : une partie du code correspond aux 12 chiffres significatifs, et une autre à l'ordre de grandeur (exposant de 10).

Exercices

5.1 Écrivez un programme qui convertisse en radians un angle fourni au départ en degrés, minutes, secondes.

5.2 Écrivez un programme qui convertisse en degrés, minutes, secondes un angle fourni au départ en radians.

5.3 Écrivez un programme qui convertisse en degrés Celsius une température exprimée au départ en degrés Fahrenheit, ou l'inverse.
La formule de conversion est : $T_F = T_C \times 1{,}8 + 32$.

5.4 Écrivez un programme qui calcule les intérêts accumulés chaque année pendant 20 ans, par capitalisation d'une somme de 100 euros placée en banque au taux fixe de 4,3 %

5.5 Une légende de l'Inde ancienne raconte que le jeu d'échecs a été inventé par un vieux sage, que son roi voulut remercier en lui affirmant qu'il lui accorderait n'importe quel cadeau en récompense. Le vieux sage demanda qu'on lui fournisse simplement un peu de riz pour ses vieux jours, et plus précisément un nombre de grains de riz suffisant pour que l'on puisse en déposer 1 seul sur la première case du jeu qu'il venait d'inventer, deux sur la suivante, quatre sur la troisième, et ainsi de suite jusqu'à la 64e case.
Écrivez un programme Python qui affiche le nombre de grains à déposer sur chacune des 64 cases du jeu. Calculez ce nombre de deux manières :
- le nombre exact de grains (nombre entier) ;
- le nombre de grains en notation scientifique (nombre réel).

Les données alphanumériques

Jusqu'à présent nous n'avons manipulé que des nombres. Mais un programme d'ordinateur peut également traiter des caractères alphabétiques, des mots, des phrases, ou des suites de symboles quelconques. Dans la plupart des langages de programmation, il existe pour cet usage des structures de données particulières que l'on appelle « chaînes de caractères ».

> *Nous apprendrons au chapitre 10 qu'il ne faut pas confondre les notions de « chaîne de caractères » et « séquence d'octets » comme le faisaient abusivement les langages de programmation anciens (dont les premières versions de Python). Pour l'instant, réjouissons-nous que Python traite désormais de manière parfaitement cohérente toutes les chaînes de caractères, ceux-ci pouvant faire partie d'alphabets quelconques*[21].

Le type string

Une donnée de type *string* peut se définir en première approximation comme une suite quelconque de caractères. Dans un script python, on peut délimiter une telle suite de caractères, soit par des apostrophes (simple quotes), soit par des guillemets (double quotes). Exemples :

```
>>> phrase1 = 'les oeufs durs.'
>>> phrase2 = '"Oui", répondit-il,'
>>> phrase3 = "j'aime bien"
>>> print(phrase2, phrase3, phrase1)
"Oui", répondit-il, j'aime bien les oeufs durs.
```

Les 3 variables **phrase1**, **phrase2**, **phrase3** sont donc des variables de type *string*.

Remarquez l'utilisation des guillemets pour délimiter une chaîne dans laquelle il y a des apostrophes, ou l'utilisation des apostrophes pour délimiter une chaîne qui contient des guillemets. Remarquez aussi encore une fois que la fonction **print()** insère un espace entre les éléments affichés.

Le caractère spécial « \ » (*antislash*) permet quelques subtilités complémentaires :

- En premier lieu, il permet d'écrire sur plusieurs lignes une commande qui serait trop longue pour tenir sur une seule (cela vaut pour n'importe quel type de commande).
- À l'intérieur d'une chaîne de caractères, l'*antislash* permet d'insérer un certain nombre de codes spéciaux (sauts à la ligne, apostrophes, guillemets, etc.). Exemples :

```
>>> txt3 = '"N\'est-ce pas ?" répondit-elle.'
>>> print(txt3)
"N'est-ce pas ?" répondit-elle.
>>> Salut = "Ceci est une chaîne plutôt longue\n contenant plusieurs lignes \
... de texte (Ceci fonctionne\n de la même façon en C/C++.\n\
...     Notez que les blancs en début\n de ligne sont significatifs.\n"
>>> print(Salut)
Ceci est une chaîne plutôt longue
 contenant plusieurs lignes de texte (Ceci fonctionne
 de la même façon en C/C++.
    Notez que les blancs en début
 de ligne sont significatifs.
```

[21] Ceci constitue donc l'une des grandes nouveautés de la version 3 de Python par rapport aux versions précédentes. Dans celles-ci, une donnée de type *string* était en réalité une *séquence d'octets* et non une *séquence de caractères*. Cela ne posait guère de problèmes pour traiter des textes contenant seulement les caractères principaux des langues d'Europe occidentale, car il était possible d'encoder chacun de ces caractères sur un seul octet (en suivant par exemple la norme Latin-1). Cela entraînait cependant de grosses difficultés si l'on voulait rassembler dans un même texte des caractères tirés d'alphabets différents, ou simplement utiliser des alphabets comportant plus de 256 caractères, des symboles mathématiques particuliers, etc. Vous trouverez davantage d'informations à ce sujet au chapitre 10.

Remarques

- La séquence `\n` dans une chaîne provoque un saut à la ligne.
- La séquence `\'` permet d'insérer une apostrophe dans une chaîne délimitée par des apostrophes. De la même manière, la séquence `\"` permet d'insérer des guillemets dans une chaîne délimitée elle-même par des guillemets.
- Rappelons encore ici que la casse est significative dans les noms de variables (il faut respecter scrupuleusement le choix initial de majuscules ou minuscules).

Triple quotes

Pour insérer plus aisément des caractères spéciaux ou « exotiques » dans une chaîne, sans faire usage de l'*antislash*, ou pour faire accepter l'*antislash* lui-même dans la chaîne, on peut encore délimiter la chaîne à l'aide de *triples guillemets* ou de *triples apostrophes* :

```
>>> a1 = """
...   Exemple de texte préformaté, c'est-à-dire
...      dont les indentations et les
...         caractères spéciaux \ ' " sont
... conservés sans
...       autre forme de procès."""
>>> print(a1)

  Exemple de texte préformaté, c'est-à-dire
     dont les indentations et les
        caractères spéciaux \ ' " sont
conservés sans
      autre forme de procès.
>>>
```

Accès aux caractères individuels d'une chaîne

Les chaînes de caractères constituent un cas particulier d'un type de données plus général que l'on appelle des *données composites*. Une donnée composite est une entité qui rassemble dans une seule structure un ensemble d'entités plus simples : dans le cas d'une chaîne de caractères, par exemple, ces entités plus simples sont évidemment les caractères eux-mêmes. En fonction des circonstances, nous souhaiterons traiter la chaîne de caractères, tantôt comme un seul objet, tantôt comme une collection de caractères distincts. Un langage de programmation tel que Python doit donc être pourvu de mécanismes qui permettent d'accéder séparément à chacun des caractères d'une chaîne. Comme vous allez le voir, cela n'est pas bien compliqué.

Python considère qu'une chaîne de caractères est un objet de la catégorie des *séquences*, lesquelles sont des *collections ordonnées d'éléments*. Cela signifie simplement que les caractères d'une chaîne sont toujours disposés dans un certain ordre. Par conséquent, chaque caractère de la chaîne peut être désigné par sa place dans la séquence, à l'aide d'un *index*.

Pour accéder à un caractère bien déterminé, on utilise le nom de la variable qui contient la chaîne et on lui accole, entre deux crochets, l'index numérique qui correspond à la position du caractère dans la chaîne.

Attention cependant : comme vous aurez l'occasion de le vérifier par ailleurs, les données informatiques sont presque toujours numérotées *à partir de zéro* (et non à partir de un). C'est le cas pour les caractères d'une chaîne.

Exemple :

```
>>> ch = "Christine"
>>> print(ch[0], ch[3], ch[5])
C i t
```

Vous pouvez recommencer l'exercice de l'exemple ci-dessus en utilisant cette fois un ou deux caractères non-ASCII, tels que lettres accentuées, cédilles, etc. Contrairement à ce qui pouvait se passer dans certains cas avec les versions de Python antérieures à la version 3.0, vous obtenez sans surprise les résultats attendus :

```
>>> ch ="Noël en Décembre"
>>> print(ch[1],ch[2],ch[3],ch[4],ch[8],ch[9],ch[10],ch[11],ch[12])
o ë l   D é c e m
```

Ne vous préoccupez pas pour l'instant de savoir de quelle manière Python mémorise et traite les caractères typographiques dans la mémoire de l'ordinateur. Sachez cependant que la technique utilisée exploite la norme internationale Unicode. qui permet de distinguer de façon univoque n'importe quel caractère de n'importe quel alphabet. Vous pourrez donc mélanger dans une même chaîne des caractères latins, grecs, cyrilliques, arabes... (y compris les caractères accentués), ainsi que des symboles mathématiques, des pictogrammes, etc.

> *Nous verrons au chapitre 10 (voir page 121) comment faire apparaître d'autres caractères que ceux qui sont directement accessibles au clavier.*

Opérations élémentaires sur les chaînes

Python intègre de nombreuses *fonctions* qui permettent d'effectuer divers traitements sur les chaînes de caractères (conversions majuscules/minuscules, découpage en chaînes plus petites, recherche de mots, etc.). Une fois de plus, cependant, nous devons vous demander de patienter : ces questions ne seront développées qu'à partir du chapitre 10 (voir page 121).

Pour l'instant, nous pouvons nous contenter de savoir qu'il est possible d'accéder individuellement à chacun des caractères d'une chaîne, comme cela a été expliqué dans la section précédente. Sachons en outre que l'on peut aussi :

- assembler plusieurs petites chaînes pour en construire de plus grandes. Cette opération s'appelle *concaténation* et on la réalise sous Python à l'aide de l'opérateur `+` (cet opérateur réalise donc l'opération d'addition lorsqu'on l'applique à des nombres, et l'opération de concaténation lorsqu'on l'applique à des chaînes de caractères). Exemple :

```
a = 'Petit poisson'
b = ' deviendra grand'
c = a + b
print(c)
petit poisson deviendra grand
```

- déterminer la longueur (c'est-à-dire le nombre de caractères) d'une chaîne, en faisant appel à la fonction intégrée **len()** :

```
>>> ch ='Georges'
>>> print(len(ch))
7
```

Cela marche tout aussi bien si la chaîne contient des caractères accentués :

```
>>> ch ='René'
>>> print(len(ch))
4
```

- Convertir en nombre véritable une chaîne de caractères qui représente un nombre. Exemple :

```
>>> ch = '8647'
>>> print(ch + 45)
```

→ *** erreur *** : on ne peut pas additionner une chaîne et un nombre

```
>>> n = int(ch)
>>> print(n + 65)
8712                    # OK : on peut additionner 2 nombres
```

Dans cet exemple, la fonction intégrée **int()** convertit la chaîne en nombre entier. Il serait également possible de convertir une chaîne de caractères en nombre réel, à l'aide de la fonction intégrée **float()**.

Exercices

5.6 Écrivez un script qui détermine si une chaîne contient ou non le caractère « e ».

5.7 Écrivez un script qui compte le nombre d'occurrences du caractère « e » dans une chaîne.

5.8 Écrivez un script qui recopie une chaîne (dans une nouvelle variable), en insérant des astérisques entre les caractères.
Ainsi par exemple, « **gaston** » devra devenir « **g*a*s*t*o*n** »

5.9 Écrivez un script qui recopie une chaîne (dans une nouvelle variable) en l'inversant.
Ainsi par exemple, « **zorglub** » deviendra « **bulgroz** ».

5.10 En partant de l'exercice précédent, écrivez un script qui détermine si une chaîne de caractères donnée est un *palindrome* (c'est-à-dire une chaîne qui peut se lire indifféremment dans les deux sens), comme par exemple « radar » ou « s.o.s ».

Les listes (première approche)

Les chaînes que nous avons abordées à la rubrique précédente constituaient un premier exemple de *données composites*. On appelle ainsi les structures de données qui sont utilisées pour regrouper de manière structurée des ensembles de valeurs. Vous apprendrez progressivement à utiliser plusieurs autres types de données composites, parmi lesquelles les *listes*, les *tuples* et les *dictionnaires*[22]. Nous n'allons cependant aborder ici que le premier de ces trois types, et ce de façon assez sommaire. Il s'agit-là en effet d'un sujet fort vaste, sur lequel nous devrons revenir à plusieurs reprises.

Sous Python, on peut définir une liste comme *une collection d'éléments séparés par des virgules, l'ensemble étant enfermé dans des crochets*. Exemple :

[22] Vous pourrez même créer vos propres types de données composites, lorsque vous aurez assimilé le concept de ***classe*** (voir page 163).

```
>>> jour = ['lundi', 'mardi', 'mercredi', 1800, 20.357, 'jeudi', 'vendredi']
>>> print(jour)
['lundi', 'mardi', 'mercredi', 1800, 20.357, 'jeudi', 'vendredi']
```

Dans cet exemple, la valeur de la variable **jour** est une liste.

Comme on peut le constater dans le même exemple, les éléments individuels qui constituent une liste peuvent être de types variés. Dans cet exemple, en effet, les trois premiers éléments sont des chaînes de caractères, le quatrième élément est un entier, le cinquième un réel, etc. (nous verrons plus loin qu'un élément d'une liste peut lui-même être une liste !). À cet égard, le concept de liste est donc assez différent du concept de « tableau » (*array*) ou de « variable indicée » que l'on rencontre dans d'autres langages de programmation.

Remarquons aussi que, comme les chaînes de caractères, les listes sont des *séquences*, c'est-à-dire des *collections ordonnées d'objets*. Les divers éléments qui constituent une liste sont en effet toujours disposés dans le même ordre, et l'on peut donc accéder à chacun d'entre eux individuellement si l'on connaît son *index* dans la liste. Comme c'était déjà le cas pour les caractères dans une chaîne, il faut cependant retenir que la numérotation de ces index commence *à partir de zéro*, et non à partir de un.

Exemples :

```
>>> jour = ['lundi', 'mardi', 'mercredi', 1800, 20.357, 'jeudi', 'vendredi']
>>> print(jour[2])
mercredi
>>> print(jour[4])
20.357
```

À la différence de ce qui se passe pour les chaînes, qui constituent un type de données *non-modifiables* (nous aurons plus loin diverses occasions de revenir là-dessus), il est possible de changer les éléments individuels d'une liste :

```
>>> print(jour)
['lundi', 'mardi', 'mercredi', 1800, 20.357, 'jeudi', 'vendredi']
>>> jour[3] = jour[3] +47
>>> print(jour)
['lundi', 'mardi', 'mercredi', 1847, 20.357, 'jeudi', 'vendredi']
```

On peut donc remplacer certains éléments d'une liste par d'autres, comme ci-dessous :

```
>>> jour[3] = 'Juillet'
>>> print(jour)
['lundi', 'mardi', 'mercredi', 'Juillet', 20.357, 'jeudi', 'vendredi']
```

La *fonction intégrée* **len()**, que nous avons déjà rencontrée à propos des chaînes, s'applique aussi aux listes. Elle renvoie le nombre d'éléments présents dans la liste :

```
>>> print(len(jour))
7
```

Une autre *fonction intégrée* permet de supprimer d'une liste un élément quelconque (à partir de son index). Il s'agit de la fonction **del()** [23] :

```
>>> del(jour[4])
>>> print(jour)
['lundi', 'mardi', 'mercredi', 'juillet', 'jeudi', 'vendredi']
```

Il est également tout à fait possible d'ajouter un élément à une liste, mais pour ce faire, il faut considérer que la liste est un *objet*, dont on va utiliser l'une des *méthodes*. Les concepts informatiques *d'objet* et de *méthode* ne seront expliqués qu'un peu plus loin dans ces notes, mais nous pouvons dès à présent montrer « comment ça marche » dans le cas particulier d'une liste :

```
>>> jour.append('samedi')
>>> print(jour)
['lundi', 'mardi', 'mercredi', 'juillet', 'jeudi', 'vendredi', 'samedi']
>>>
```

Dans la première ligne de l'exemple ci-dessus, nous avons appliqué la *méthode* **append()** à *l'objet* **jour**, avec *l'argument* 'samedi'. Si l'on se rappelle que le mot « append » signifie « ajouter » en anglais, on peut comprendre que la méthode **append()** est une sorte de *fonction* qui est en quelque manière attachée ou intégrée aux objets du type « liste ». L'argument que l'on utilise avec cette fonction est bien entendu l'élément que l'on veut ajouter à la fin de la liste.

Nous verrons plus loin qu'il existe ainsi toute une série de ces *méthodes* (c'est-à-dire des fonctions intégrées, ou plutôt « encapsulées » dans les objets de type « liste »). Notons simplement au passage que l'on applique une méthode à un objet *en reliant les deux à l'aide d'un point*. (D'abord le nom de la variable qui référence l'objet, puis le point, puis le nom de la méthode, cette dernière toujours accompagnée d'une paire de parenthèses.)

Comme les chaînes de caractères, les listes seront approfondies plus loin dans ces notes (voir page 141). Nous en savons cependant assez pour commencer à les utiliser dans nos programmes. Veuillez par exemple analyser le petit script ci-dessous et commenter son fonctionnement :

```
jour = ['dimanche','lundi','mardi','mercredi','jeudi','vendredi','samedi']
a, b = 0, 0
while a<25:
     a = a + 1
     b = a % 7
     print(a, jour[b])
```

La 5e ligne de cet exemple fait usage de l'opérateur « *modulo* » déjà rencontré précédemment et qui peut rendre de grands services en programmation. On le représente par % dans de nombreux langages (dont Python). Quelle est l'opération effectuée par cet opérateur ?

[23] Il existe en fait tout un ensemble de techniques qui permettent de découper une liste en tranches, d'y insérer des groupes d'éléments, d'en enlever d'autres, etc., en utilisant une syntaxe particulière où n'interviennent que les index.
Cet ensemble de techniques (qui peuvent aussi s'appliquer aux chaînes de caractères) porte le nom générique de *slicing* (tranchage). On le met en œuvre en plaçant plusieurs indices au lieu d'un seul entre les crochets que l'on accole au nom de la variable. Ainsi jour[1:3] désigne le sous-ensemble ['mardi', 'mercredi'].
Ces techniques un peu particulières sont décrites plus loin (voir pages 121 et suivantes).

Exercices

5.11 Soient les listes suivantes :

```
t1 = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
t2 = ['Janvier', 'Février', 'Mars', 'Avril', 'Mai', 'Juin',
      'Juillet', 'Août', 'Septembre', 'Octobre', 'Novembre', 'Décembre']
```

Écrivez un petit programme qui crée une nouvelle liste **t3**. Celle-ci devra contenir tous les éléments des deux listes en les alternant, de telle manière que chaque nom de mois soit suivi du nombre de jours correspondant :

```
['Janvier',31,'Février',28,'Mars',31, etc...].
```

5.12 Écrivez un programme qui affiche « proprement » tous les éléments d'une liste. Si on l'appliquait par exemple à la liste **t2** de l'exercice ci-dessus, on devrait obtenir :

```
Janvier Février Mars Avril Mai Juin Juillet Août Septembre Octobre Novembre
Décembre
```

5.13 Écrivez un programme qui recherche le plus grand élément présent dans une liste donnée. Par exemple, si on l'appliquait à la liste `[32, 5, 12, 8, 3, 75, 2, 15]`, ce programme devrait afficher :

```
le plus grand élément de cette liste a la valeur 75.
```

5.14 Écrivez un programme qui analyse un par un tous les éléments d'une liste de nombres (par exemple celle de l'exercice précédent) pour générer deux nouvelles listes. L'une contiendra seulement les nombres *pairs* de la liste initiale, et l'autre les nombres *impairs*. Par exemple, si la liste initiale est celle de l'exercice précédent, le programme devra construire une liste **pairs** qui contiendra `[32, 12, 8, 2]`, et une liste **impairs** qui contiendra `[5, 3, 75, 15]`. Astuce : pensez à utiliser l'opérateur **modulo** (%) déjà cité précédemment.

5.15 Écrivez un programme qui analyse un par un tous les éléments d'une liste de mots (par exemple : `['Jean', 'Maximilien', 'Brigitte', 'Sonia', 'Jean-Pierre', 'Sandra']`) pour générer deux nouvelles listes. L'une contiendra les mots comportant moins de 6 caractères, l'autre les mots comportant 6 caractères ou davantage.

6

Fonctions prédéfinies

L'un des concepts les plus importants en programmation est celui de fonction[24]*. Les fonctions permettent en effet de décomposer un programme complexe en une série de sous-programmes plus simples, lesquels peuvent à leur tour être décomposés en fragments plus petits, et ainsi de suite. D'autre part, les fonctions sont réutilisables : si nous disposons d'une fonction capable de calculer une racine carrée, par exemple, nous pouvons l'utiliser un peu partout dans nos programmes sans avoir à la ré-écrire à chaque fois.*

La fonction print()

Nous avons bien évidemment déjà rencontré cette fonction. Précisons simplement ici qu'elle permet d'afficher n'importe quel nombre de valeurs fournies en arguments (c'est-à-dire entre les parenthèses). Par défaut, ces valeurs seront séparées les unes des autres par un espace, et le tout se terminera par un saut à la ligne.

Vous pouvez remplacer le séparateur par défaut (l'espace) par un autre caractère quelconque (ou même par aucun caractère), grâce à l'argument **sep**. Exemple :

```
>>> print("Bonjour", "à", "tous", sep ="*")
Bonjour*à*tous
 >>> print("Bonjour", "à", "tous", sep ="")
Bonjouràtous
```

De même, vous pouvez remplacer le saut à la ligne par l'argument **end** :

```
>>> n =0
>>> while n<6:
...     print("zut", end ="")
...     n = n+1
...
zutzutzutzutzut
```

[24] Sous Python, le terme « fonction » est utilisé indifféremment pour désigner à la fois de véritables fonctions mais également des **procédures**. Nous indiquerons plus loin la distinction entre ces deux concepts proches.

Interaction avec l'utilisateur : la fonction input()

La plupart des scripts élaborés nécessitent à un moment ou l'autre une intervention de l'utilisateur (entrée d'un paramètre, clic de souris sur un bouton, etc.). Dans un script en mode texte (comme ceux que nous avons créés jusqu'à présent), la méthode la plus simple consiste à employer la fonction intégrée **input()**. Cette fonction provoque une interruption dans le programme courant. L'utilisateur est invité à entrer des caractères au clavier et à terminer avec *<Enter>*. Lorsque cette touche est enfoncée, l'exécution du programme se poursuit, et la fonction fournit en retour une chaîne de caractères correspondant à ce que l'utilisateur a saisi. Cette chaîne peut alors être assignée à une variable quelconque, convertie, etc.

On peut invoquer la fonction **input()** en laissant les parenthèses vides. On peut aussi y placer en argument un message explicatif destiné à l'utilisateur. Exemple :

```
prenom = input("Entrez votre prénom : ")
print("Bonjour,", prenom)
```

ou encore :

```
print("Veuillez entrer un nombre positif quelconque : ", end=" ")
ch = input()
nn = int(ch)              # conversion de la chaîne en un nombre entier
print("Le carré de", nn, "vaut", nn**2)
```

Soulignons que la fonction **input()** renvoie toujours une chaîne de caractères[25]. Si vous souhaitez que l'utilisateur entre une valeur numérique, vous devrez donc convertir la valeur entrée (qui sera donc de toute façon de type **string**) en une valeur numérique du type qui vous convient, par l'intermédiaire des fonctions intégrées **int()** (si vous attendez un entier) ou **float()** (si vous attendez un réel). Exemple :

```
>>> a = input("Entrez une donnée numérique : ")
Entrez une donnée numérique : 52.37
>>> type(a)
<class 'str'>
>>> b = float(a)        # conversion de la chaîne en un nombre réel
>>> type(b)
<class 'float'>
```

Importer un module de fonctions

Vous avez déjà rencontré d'autres *fonctions intégrées* au langage lui-même, comme la fonction **len()**, par exemple, qui permet de connaître la longueur d'une chaîne de caractères. Il va de soi cependant qu'il n'est pas possible d'intégrer toutes les fonctions imaginables dans le corps standard de Python, car il en existe virtuellement une infinité : vous apprendrez d'ailleurs très bientôt comment en créer vous-même de nouvelles. Les fonctions intégrées au langage sont relativement peu nombreuses : ce sont seulement celles qui sont susceptibles d'être utilisées très fréquemment. Les autres sont regroupées dans des fichiers séparés que l'on appelle des *modules*.

[25] Dans les versions de Python antérieures à la version 3.0, la valeur renvoyée par **input()** était de type variable, suivant ce que l'utilisateur avait saisi. Le comportement actuel est en fait celui de l'ancienne fonction **raw_input()**, que lui préféraient la plupart des programmeurs.

> *Les modules sont des fichiers qui regroupent des ensembles de fonctions*[26].

Vous verrez plus loin combien il est commode de découper un programme important en plusieurs fichiers de taille modeste pour en faciliter la maintenance. Une application Python typique sera alors constituée d'un programme principal, accompagné de un ou plusieurs modules contenant chacun les définitions d'un certain nombre de fonctions accessoires.

Il existe un grand nombre de modules pré-programmés qui sont fournis d'office avec Python. Vous pouvez en trouver d'autres chez divers fournisseurs. Souvent on essaie de regrouper dans un même module des ensembles de fonctions apparentées, que l'on appelle des *bibliothèques*.

Le module *math*, par exemple, contient les définitions de nombreuses fonctions mathématiques telles que *sinus*, *cosinus*, *tangente*, *racine carrée*, etc. Pour pouvoir utiliser ces fonctions, il vous suffit d'incorporer la ligne suivante au début de votre script :

```
from math import *
```

Cette ligne indique à Python qu'il lui faut inclure dans le programme courant *toutes* les fonctions (c'est là la signification du symbole « joker » *) du module *math*, lequel contient une bibliothèque de fonctions mathématiques pré-programmées.

Dans le corps du script lui-même, vous écrirez par exemple :
`racine = sqrt(nombre)` pour assigner à la variable **`racine`** la racine carrée de **`nombre`**,
`sinusx = sin(angle)` pour assigner à la variable **`sinusx`** le sinus de **`angle`** (en radians !), etc.

Exemple :

```
# Démo : utilisation des fonctions du module <math>

from math import *

nombre = 121
angle = pi/6        # soit 30°
                    # (la bibliothèque math inclut aussi la définition de pi)
print("racine carrée de", nombre, "=", sqrt(nombre))
print("sinus de", angle, "radians", "=", sin(angle))
```

L'exécution de ce script provoque l'affichage suivant :

```
racine carrée de 121 = 11.0
sinus de 0.523598775598 radians = 0.5
```

Ce court exemple illustre déjà fort bien quelques caractéristiques importantes des fonctions :

- une fonction apparaît sous la forme d'*un nom quelconque associé à des parenthèses*
  exemple : **sqrt()**
- dans les parenthèses, on *transmet* à la fonction un ou plusieurs *arguments*
  exemple : **sqrt(121)**
- la fonction fournit une *valeur de retour* (on dira aussi qu'elle « retourne », ou mieux, qu'elle « renvoie » une valeur)
  exemple : **11.0**

[26] En toute rigueur, un module peut contenir aussi des définitions de variables ainsi que des classes. Nous pouvons toutefois laisser ces précisions de côté, provisoirement.

Nous allons développer tout ceci dans les pages suivantes. Veuillez noter au passage que les fonctions mathématiques utilisées ici ne représentent qu'un tout premier exemple. Un simple coup d'œil dans la documentation des bibliothèques Python vous permettra de constater que de très nombreuses fonctions sont d'ores et déjà disponibles pour réaliser une multitude de tâches, y compris des algorithmes mathématiques très complexes (Python est couramment utilisé dans les universités pour la résolution de problèmes scientifiques de haut niveau). Il est donc hors de question de fournir ici une liste détaillée. Une telle liste est aisément accessible dans le système d'aide de Python :

Documentation HTML → Python documentation → Modules index → math

Au chapitre suivant, nous apprendrons comment créer nous-mêmes de nouvelles fonctions.

Exercices

> *Dans tous ces exercices, utilisez la fonction* ***input()*** *pour l'entrée des données.*

6.1 Écrivez un programme qui convertisse en mètres par seconde et en km/h une vitesse fournie par l'utilisateur en miles/heure. (*Rappel : 1 mile = 1609 mètres*)

6.2 Écrivez un programme qui calcule le périmètre et l'aire d'un triangle quelconque dont l'utilisateur fournit les 3 côtés.
(Rappel : l'aire d'un triangle quelconque se calcule à l'aide de la formule :

$$S=\sqrt{d\cdot(d-a)\cdot(d-b)\cdot(d-c)}$$

dans laquelle d désigne la longueur du demi-périmètre, et a, b, c celles des trois côtés.)

6.3 Écrivez un programme qui calcule la période d'un pendule simple de longueur donnée.

La formule qui permet de calculer la période d'un pendule simple est $T=2\pi\sqrt{\frac{l}{g}}$,

l représentant la longueur du pendule et ***g*** la valeur de l'accélération de la pesanteur au lieu d'expérience.

6.4 Écrivez un programme qui permette d'encoder des valeurs dans une liste. Ce programme devrait fonctionner en boucle, l'utilisateur étant invité à entrer sans cesse de nouvelles valeurs, jusqu'à ce qu'il décide de terminer en frappant <*Enter*> en guise d'entrée. Le programme se terminerait alors par l'affichage de la liste. Exemple de fonctionnement :

```
Veuillez entrer une valeur : 25
Veuillez entrer une valeur : 18
Veuillez entrer une valeur : 6284
Veuillez entrer une valeur :
[25, 18, 6284]
```

Un peu de détente avec le module turtle

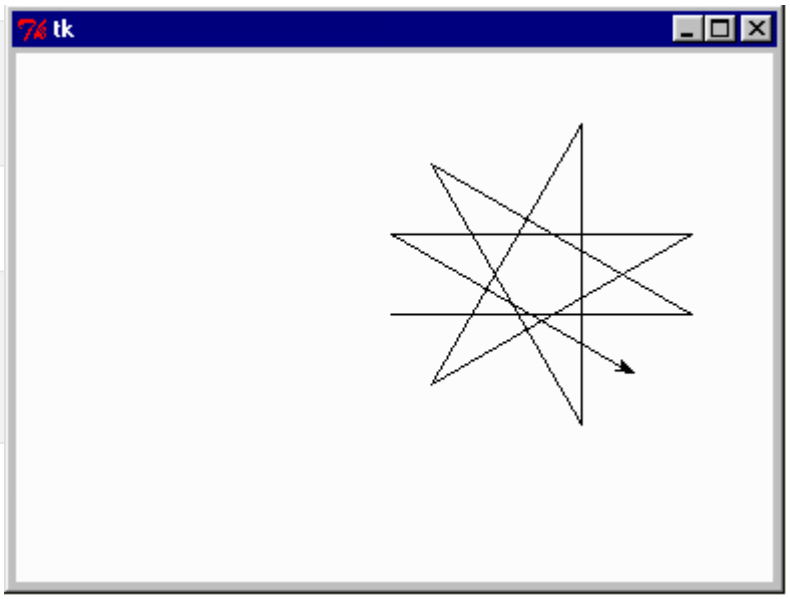
Comme nous venons de le voir, l'une des grandes qualités de Python est qu'il est extrêmement facile de lui ajouter de nombreuses fonctionnalités par importation de divers *modules*.

Pour illustrer cela, et nous amuser un peu avec d'autres objets que des nombres, nous allons explorer un module Python qui permet de réaliser des « graphiques tortue », c'est-à-dire des dessins

géométriques correspondant à la piste laissée derrière elle par une petite « tortue » virtuelle, dont nous contrôlons les déplacements sur l'écran de l'ordinateur à l'aide d'instructions simples.

Activer cette tortue est un vrai jeu d'enfant. Plutôt que de vous donner de longues explications, nous vous invitons à essayer tout de suite :

```
>>> from turtle import *
>>> forward(120)
>>> left(90)
>>> color('red')
>>> forward(80)
```



L'exercice est évidemment plus riche si l'on utilise des boucles :

```
>>> reset()
>>> a = 0
>>> while a <12:
        a = a +1
        forward(150)
        left(150)
```

Attention cependant : avant de lancer un tel script, assurez-vous toujours qu'il ne comporte pas de boucle sans fin (voir page 29), car si c'est le cas vous risquez de ne plus pouvoir reprendre le contrôle des opérations (en particulier sous *Windows*).

Amusez-vous à écrire des scripts qui réalisent des dessins suivant un modèle imposé à l'avance. Les principales fonctions mises à votre disposition dans le module *turtle* sont les suivantes :

reset()	On efface tout et on recommence
goto(x, y)	Aller à l'endroit de coordonnées x, y
forward(distance)	Avancer d'une distance donnée
backward(distance)	Reculer
up()	Relever le crayon (pour pouvoir avancer sans dessiner)
down()	Abaisser le crayon (pour recommencer à dessiner)
color(couleur)	*couleur* peut être une chaîne prédéfinie ('red', 'blue', etc.)
left(angle)	Tourner à gauche d'un angle donné (exprimé en degrés)
right(angle)	Tourner à droite
width(épaisseur)	Choisir l'épaisseur du tracé
fill(1)	Remplir un contour fermé à l'aide de la couleur sélectionnée
write(texte)	*texte* doit être une chaîne de caractères

Véracité/fausseté d'une expression

Lorsqu'un programme contient des instructions telles que **while** ou **if**, l'ordinateur qui exécute ce programme doit évaluer la véracité d'une condition, c'est-à-dire déterminer si une expression est *vraie* ou *fausse*. Par exemple, une boucle initiée par `while c<20:` s'exécutera aussi longtemps que la condition `c<20` restera *vraie*.

Mais comment un ordinateur peut-il déterminer si quelque chose est *vrai* ou *faux* ?

En fait – et vous le savez déjà – un ordinateur ne manipule strictement que des nombres. Tout ce qu'un ordinateur doit traiter doit d'abord toujours être converti en valeur numérique. Cela s'applique aussi à la notion de vrai/faux. En Python, tout comme en *C*, en *Basic* et en de nombreux autres langages de programmation, on considère que toute valeur numérique autre que zéro est « vraie ». Seule la valeur zéro est « fausse ». Exemple :

```
ch = input('Entrez un nombre entier quelconque')
n =int(ch)
if n:
    print("vrai")
else:
    print("faux")
```

Le petit script ci-dessus n'affiche **faux** que si vous entrez la valeur 0. Pour toute autre valeur numérique, vous obtiendrez **vrai**.

Ce qui précède signifie donc qu'une expression à évaluer, telle par exemple la condition **a > 5**, est d'abord convertie par l'ordinateur en une valeur numérique (**1** si l'expression est vraie, et **zéro** si l'expression est fausse). Ce n'est pas tout à fait évident, parce que l'interpréteur Python est doté d'un dispositif qui traduit ces deux valeurs en **True** ou **False** lorsqu'on les lui demande explicitement. Exemple :

```
>>> a, b = 3, 8
>>> c = (a < b)
>>> d = (a > b)
>>> c
True
>>> d
False
```

L'expression **a < b** est évaluée, et son résultat (**vrai**) est mémorisé dans la variable **c**. De même pour le résultat de l'expression inverse, dans la variable **d**[27].

À l'aide d'une petite astuce, nous pouvons tout de même vérifier que ces valeurs `True` et `False` sont en réalité les deux nombres 1 et 0 « déguisés » :

```
>>> accord = ["non", "oui"]
>>> accord[d]
non
>>> accord[c]
oui
```

En utilisant les valeurs des variables **c** et **d** comme indices pour extraire les éléments de la liste **accord**, nous confirmons bien que False =0 et True =1.

Le petit script ci-après est très similaire au précédent. Il nous permet de tester le caractère vrai ou faux d'une chaîne de caractères :

```
ch = input("Entrez une chaîne de caractères quelconque")
if ch:
    print("vrai")
```

[27] Ces variables sont d'un type entier un peu particulier : le type « booléen ». Les variables de ce type ne peuvent prendre que les deux valeurs **True** et **False** (en réalité, 1 et 0).

```
else:
    print("faux")
```

Vous obtiendrez **faux** pour toute chaîne *vide*, et **vrai** pour toute chaîne contenant au moins un caractère. Vous pourriez de la même manière tester la « véracité » d'une liste, et constater qu'une liste vide est fausse, alors qu'une liste ayant un contenu quelconque est vraie[28].

L'instruction **if ch:**, à la troisième ligne de cet exemple, est donc équivalente à une instruction du type **if ch !="":**, du moins de notre point de vue d'êtres humains. Pour l'ordinateur, cependant, ce n'est pas tout à fait pareil. Pour lui, l'instruction **if ch:** consiste à vérifier directement si la valeur de la variable **ch** est une chaîne vide ou non, comme nous venons de le voir. Cependant, la seconde formulation **if ch != "":** lui impose de commencer par comparer le contenu de **ch** *à la valeur d'une autre donnée que nous lui fournissons par notre programme* (une chaîne vide), puis à évaluer *ensuite* si le résultat de cette comparaison est lui-même **vrai** ou **faux** (ou en d'autres termes, à vérifier si ce résultat est lui-même **True** ou **False**). Cela lui demande donc deux opérations successives, là où la première formulation ne lui en demande qu'une seule. La première formulation est donc plus performante.

Pour les mêmes raisons, dans un script tel celui-ci :

```
ch =input("Veuillez entrer un nombre : ")
n =int(ch)
if n % 2:
     print("Il s'agit d'un nombre impair.")
else:
     print("Il s'agit d'un nombre pair.")
```

il est plus efficace de programmer la troisième ligne comme nous l'avons fait ci-dessus, plutôt que d'écrire explicitement **if n % 2 != 0**, car cette formulation imposerait à l'ordinateur d'effectuer deux comparaisons successives au lieu d'une seule.

Ce raisonnement « proche de la machine » vous paraîtra probablement un peu subtil au début, mais croyez bien que cette forme d'écriture vous deviendra très vite tout à fait familière.

Révision

Dans ce qui suit, nous n'allons pas apprendre de nouveaux concepts mais simplement utiliser tout ce que nous connaissons déjà, pour réaliser de vrais petits programmes.

Contrôle du flux – utilisation d'une liste simple

Commençons par un petit retour sur les branchements conditionnels (il s'agit peut-être là du groupe d'instructions le plus important, dans n'importe quel langage !) :

```
# Utilisation d'une liste et de branchements conditionnels

print("Ce script recherche le plus grand de trois nombres")
print("Veuillez entrer trois nombres séparés par des virgules : ")
ch =input()
# Note : l'association des fonctions eval() et list() permet de convertir
```

[28] Les autres structures de données se comportent d'une manière similaire. Les *tuples* et les *dictionnaires* que vous étudierez au chapitre 10 sont également considérés comme **faux** lorsqu'ils sont vides, et **vrais** lorsqu'ils possèdent un contenu.

```
# en liste toute chaîne de valeurs séparées par des virgules :[29]
nn = list(eval(ch))
max, index = nn[0], 'premier'
if nn[1] > max:                      # ne pas omettre le double point !
    max = nn[1]
    index = 'second'
if nn[2] > max:
    max = nn[2]
    index = 'troisième'
print("Le plus grand de ces nombres est", max)
print("Ce nombre est le", index, "de votre liste.")
```

Dans cet exercice, vous retrouvez à nouveau le concept de « bloc d'instructions », déjà abondamment commenté aux chapitres 3 et 4, et que vous devez absolument assimiler. Pour rappel, les blocs d'instructions sont délimités par *l'indentation*. Après la première instruction **if**, par exemple, il y a deux lignes indentées définissant un bloc d'instructions. Ces instructions ne seront exécutées que si la condition `nn[1] > max` est vraie.

La ligne suivante, par contre (celle qui contient la deuxième instruction **if**) n'est pas indentée. Cette ligne se situe donc au même niveau que celles qui définissent le corps principal du programme. L'instruction contenue dans cette ligne est donc toujours exécutée, alors que les deux suivantes (qui constituent encore un autre bloc) ne sont exécutées que si la condition `nn[2] > max` est vraie.

En suivant la même logique, on voit que les instructions des deux dernières lignes font partie du bloc principal et sont donc toujours exécutées.

Boucle while – instructions imbriquées

Continuons dans cette voie en imbriquant d'autres structures :

```
 1# # Instructions composées <while> - <if> - <elif> - <else>
 2#
 3# print('Choisissez un nombre de 1 à 3 (ou 0 pour terminer) ', end=' ')
 4# ch = input()
 5# a = int(ch)              # conversion de la chaîne entrée en entier
 6# while a:                # équivalent à : < while a != 0: >
 7#     if a == 1:
 8#         print("Vous avez choisi un :")
 9#         print("le premier, l'unique, l'unité ..."
10#     elif a == 2:
11#         print("Vous préférez le deux :")
12#         print("la paire, le couple, le duo ...")
13#     elif a == 3:
14#         print("Vous optez pour le plus grand des trois :")
15#         print("le trio, la trinité, le triplet ...")
16#     else :
17#         print("Un nombre entre UN et TROIS, s.v.p.")
18#     print('Choisissez un nombre de 1 à 3 (ou 0 pour terminer) ', end =' ')
19#     a = int(input())
20# print("Vous avez entré zéro :")
21# print("L'exercice est donc terminé.")
```

[29] En fait, la fonction `eval()` évalue le contenu de la chaîne fournie en argument comme étant une expression Python dont elle doit renvoyer le résultat. Par exemple : `eval("7 + 5")` renvoie l'entier `12`. Si on lui fournit une chaîne de valeurs séparées par des virgules, cela correspond pour elle à un *tuple*. Les tuples sont des séquences apparentées aux listes. Ils seront abordés au chapitre 10 (cf. page 152).

Nous retrouvons ici une boucle **while**, associée à un groupe d'instructions **if**, **elif** et **else**. Notez bien cette fois encore comment la structure logique du programme est créée à l'aide des indentations (... et n'oubliez pas le caractère « : » à la fin de chaque ligne d'en-tête !).

À la ligne 6, l'instruction **while** est utilisée comme expliqué à la page 55 : pour la comprendre, il vous suffit de vous rappeler que toute valeur numérique autre que zéro est considérée comme `vraie` par l'interpréteur Python. Vous pouvez remplacer cette forme d'écriture par `while a != 0` : si vous préférez (rappelons à ce sujet que l'opérateur de comparaison `!=` signifie « est différent de »), mais c'est moins efficace.

Cette « boucle while » relance le questionnement après chaque réponse de l'utilisateur (du moins jusqu'à ce que celui-ci décide de quitter en entrant une valeur nulle).

Dans le corps de la boucle, nous trouvons le groupe d'instructions **if**, **elif** et **else** (de la ligne 7 à la ligne 17), qui aiguille le flux du programme vers les différentes réponses, ensuite une instruction **print()** et une instruction **input()** (lignes 18 et 19) qui seront exécutées dans tous les cas de figure : notez bien leur niveau d'indentation, qui est le même que celui du bloc **if**, **elif** et **else**. Après ces instructions, le programme boucle et l'exécution reprend à l'instruction **while** (ligne 6).

À la ligne 19, nous utilisons la composition pour écrire un code plus compact, qui est équivalent aux lignes 4 et 5 rassemblées en une seule.

Les deux dernières instructions **print()** (lignes 20 et 21) ne sont exécutées qu'à la sortie de la boucle.

Exercices

6.5 Que fait le programme ci-dessous, dans les quatre cas où l'on aurait défini au préalable que la variable **a** vaut 1, 2, 3 ou 15 ?

```
if a !=2:
    print('perdu')
elif a ==3:
    print('un instant, s.v.p.')
else :
    print('gagné')
```

6.6 Que font ces programmes ?

```
a)   a = 5
     b = 2
     if (a==5) & (b<2):
         print('"&" signifie "et"; on peut aussi utiliser\
                le mot "and"')
b)   a, b = 2, 4
     if (a==4) or (b!=4):
         print('gagné')
     elif (a==4) or (b==4):
         print('presque gagné')
c)   a = 1
     if not a:
         print('gagné')
     elif a:
         print('perdu')
```

6.7 Reprendre le programme c) avec a = 0 au lieu de a = 1.
Que se passe-t-il ? Conclure !

6.8 Écrire un programme qui, étant données deux bornes entières a et b, additionne les nombres multiples de 3 et de 5 compris entre ces bornes. Prendre par exemple a = 0, b = 32 ; le résultat devrait être alors 0 + 15 + 30 = 45.
Modifier légèrement ce programme pour qu'il additionne les nombres multiples de 3 ou de 5 compris entre les bornes a et b. Avec les bornes 0 et 32, le résultat devrait donc être : 0 + 3 + 5 + 6 + 9 + 10 + 12 + 15 + 18 + 20 + 21 + 24 + 25 + 27 + 30 = 225.

6.9 Déterminer si une année (dont le millésime est introduit par l'utilisateur) est bissextile ou non. Une année A est bissextile si A est divisible par 4. Elle ne l'est cependant pas si A est un multiple de 100, à moins que A ne soit multiple de 400.

6.10 Demander à l'utilisateur son nom et son sexe (M ou F). En fonction de ces données, afficher « Cher Monsieur » ou « Chère Mademoiselle » suivi du nom de la personne.

6.11 Demander à l'utilisateur d'entrer trois longueurs a, b, c. À l'aide de ces trois longueurs, déterminer s'il est possible de construire un triangle. Déterminer ensuite si ce triangle est rectangle, isocèle, équilatéral ou quelconque. Attention : un triangle rectangle peut être isocèle.

6.12 Demander à l'utilisateur qu'il entre un nombre. Afficher ensuite : soit la racine carrée de ce nombre, soit un message indiquant que la racine carrée de ce nombre ne peut être calculée.

6.13 Convertir une note scolaire N quelconque, entrée par l'utilisateur sous forme de points (par exemple 27 sur 85), en une note standardisée suivant le code ci-après :

Note	Appréciation
N >= 80 %	A
80 % > N >= 60 %	B
60 % > N >= 50 %	C
50 % > N >= 40 %	D
N < 40 %	E

6.14 Soit la liste suivante :

```
['Jean-Michel', 'Marc', 'Vanessa', 'Anne', 'Maximilien',
 'Alexandre-Benoît', 'Louise']
```

Écrivez un script qui affiche chacun de ces noms avec le nombre de caractères correspondant.

6.15 Écrire une boucle de programme qui demande à l'utilisateur d'entrer des notes d'élèves. La boucle se terminera seulement si l'utilisateur entre une valeur négative. Avec les notes ainsi entrées, construire progressivement une liste. Après chaque entrée d'une nouvelle note (et donc à chaque itération de la boucle), afficher le nombre de notes entrées, la note la plus élevée, la note la plus basse, la moyenne de toutes les notes.

6.16 Écrivez un script qui affiche la valeur de la force de gravitation s'exerçant entre deux masses de 10 000 kg , pour des distances qui augmentent suivant une progression géométrique de raison 2, à partir de 5 cm (0,05 mètre).

La force de gravitation est régie par la formule $F = 6{,}67 \,.\, 10^{-11} \cdot \frac{m \cdot m'}{d^2}$

Exemple d'affichage :

```
d = .05 m :  la force vaut  2.668 N
d = .1 m  :  la force vaut  0.667 N
d = .2 m  :  la force vaut  0.167 N
d = .4 m  :  la force vaut  0.0417 N
```

etc.

7

Fonctions originales

La programmation est l'art d'apprendre à un ordinateur comment accomplir des tâches qu'il n'était pas capable de réaliser auparavant. L'une des méthodes les plus intéressantes pour y arriver consiste à ajouter de nouvelles instructions au langage de programmation que vous utilisez, sous la forme de fonctions originales.

Définir une fonction

Les scripts que vous avez écrits jusqu'à présent étaient à chaque fois très courts, car leur objectif était seulement de vous faire assimiler les premiers éléments du langage. Lorsque vous commencerez à développer de véritables projets, vous serez confrontés à des problèmes souvent fort complexes, et les lignes de programme vont commencer à s'accumuler...

L'approche efficace d'un problème complexe consiste souvent à le décomposer en plusieurs sous-problèmes plus simples qui seront étudiés séparément (ces sous-problèmes peuvent éventuellement être eux-mêmes décomposés à leur tour, et ainsi de suite). Or il est important que cette décomposition soit représentée fidèlement dans les algorithmes[30] pour que ceux-ci restent clairs.

D'autre part, il arrivera souvent qu'une même séquence d'instructions doive être utilisée à plusieurs reprises dans un programme, et on souhaitera bien évidemment ne pas avoir à la reproduire systématiquement.

Les *fonctions*[31] et les *classes d'objets* sont différentes structures de sous-programmes qui ont été imaginées par les concepteurs des langages de haut niveau afin de résoudre les difficultés évoquées ci-dessus. Nous allons commencer par décrire ici la *définition de fonctions* sous Python. Les *objets* et les *classes* seront examinés plus loin.

Nous avons déjà rencontré diverses fonctions pré-programmées. Voyons à présent comment en définir nous-mêmes de nouvelles.

La syntaxe Python pour la définition d'une fonction est la suivante :

[30] On appelle algorithme la séquence détaillée de toutes les opérations à effectuer pour résoudre un problème.

[31] Il existe aussi dans d'autres langages des **routines** (parfois appelés sous-programmes) et des **procédures**. Il n'existe pas de **routines** en Python. Quant au terme de **fonction**, il désigne à la fois les fonctions au sens strict (qui fournissent une valeur en retour), et les procédures (qui n'en fournissent pas).

```
def nomDeLaFonction(liste de paramètres):
    ...
    bloc d'instructions
    ...
```

- Vous pouvez choisir n'importe quel nom pour la fonction que vous créez, à l'exception des mots réservés du langage[32], et à la condition de n'utiliser aucun caractère spécial ou accentué (le caractère souligné « _ » est permis). Comme c'est le cas pour les noms de variables, il vous est conseillé d'utiliser surtout des lettres minuscules, notamment au début du nom (les noms commençant par une majuscule seront réservés aux *classes* que nous étudierons plus loin).
- Comme les instructions **if** et **while** que vous connaissez déjà, l'instruction **def** est une *instruction composée*. La ligne contenant cette instruction se termine obligatoirement par un double point, lequel introduit un bloc d'instructions que vous ne devez pas oublier *d'indenter*.
- La *liste de paramètres* spécifie quelles informations il faudra fournir en guise *d'arguments* lorsque l'on voudra utiliser cette fonction (les parenthèses peuvent parfaitement rester vides si la fonction ne nécessite pas d'arguments).
- Une fonction s'utilise pratiquement comme une instruction quelconque. Dans le corps d'un programme, un *appel de fonction* est constitué du nom de la fonction suivi de parenthèses.
  Si c'est nécessaire, on place dans ces parenthèses le ou les arguments que l'on souhaite transmettre à la fonction. Il faudra en principe fournir un argument pour chacun des paramètres spécifiés dans la définition de la fonction, encore qu'il soit possible de définir pour ces paramètres des valeurs par défaut (voir plus loin).

Fonction simple sans paramètres

Pour notre première approche concrète des fonctions, nous allons travailler à nouveau en mode interactif. Le mode interactif de Python est en effet idéal pour effectuer des petits tests comme ceux qui suivent. C'est une facilité que n'offrent pas tous les langages de programmation !

```
>>> def table7():
...     n = 1
...     while n <11 :
...         print(n * 7, end =' ')
...         n = n +1
...
```

En entrant ces quelques lignes, nous avons défini une fonction très simple qui calcule et affiche les 10 premiers termes de la table de multiplication par 7. Notez bien les parenthèses[33], le double point, et l'indentation du bloc d'instructions qui suit la ligne d'en-tête (c'est ce bloc d'instructions qui constitue le corps de la fonction proprement dite).

Pour utiliser la fonction que nous venons de définir, il suffit de l'appeler par son nom. Ainsi :

```
>>> table7()
```

provoque l'affichage de :

[32] La liste complète des mots réservés Python se trouve page 14.

[33] Un nom de fonction doit toujours être accompagné de parenthèses, même si la fonction n'utilise aucun paramètre. Il en résulte une convention d'écriture qui stipule que dans un texte quelconque traitant de programmation d'ordinateur, un nom de fonction soit toujours accompagné d'une paire de parenthèses vides. Nous respecterons cette convention dans la suite de ce texte.

```
7 14 21 28 35 42 49 56 63 70
```

Nous pouvons maintenant réutiliser cette fonction à plusieurs reprises, autant de fois que nous le souhaitons. Nous pouvons également l'incorporer dans la définition d'une autre fonction, comme dans l'exemple ci-dessous :

```
>>> def table7triple():
...     print('La table par 7 en triple exemplaire :')
...     table7()
...     table7()
...     table7()
...
```

Utilisons cette nouvelle fonction, en entrant la commande :

```
>>> table7triple()
```

l'affichage résultant devrait être :

```
La table par 7 en triple exemplaire :
7 14 21 28 35 42 49 56 63 70
7 14 21 28 35 42 49 56 63 70
7 14 21 28 35 42 49 56 63 70
```

Une première fonction peut donc appeler une deuxième fonction, qui elle-même en appelle une troisième, etc. Au stade où nous sommes, vous ne voyez peut-être pas encore très bien l'utilité de tout cela, mais vous pouvez déjà noter deux propriétés intéressantes :

- Créer une nouvelle fonction vous offre l'opportunité de donner un nom à tout un ensemble d'instructions. De cette manière, vous pouvez simplifier le corps principal d'un programme, en dissimulant un algorithme secondaire complexe sous une commande unique, à laquelle vous pouvez donner un nom très explicite, en français si vous voulez.
- Créer une nouvelle fonction peut servir à raccourcir un programme, par élimination des portions de code qui se répètent. Par exemple, si vous devez afficher la table par 7 plusieurs fois dans un même programme, vous n'avez pas à réécrire chaque fois l'algorithme qui accomplit ce travail.

Une fonction est donc en quelque sorte une nouvelle instruction personnalisée, que vous ajoutez vous-même librement à votre langage de programmation.

Fonction avec paramètre

Dans nos derniers exemples, nous avons défini et utilisé une fonction qui affiche les termes de la table de multiplication par 7. Supposons à présent que nous voulions faire de même avec la table par 9. Nous pouvons bien entendu réécrire entièrement une nouvelle fonction pour cela. Mais si nous nous intéressons plus tard à la table par 13, il nous faudra encore recommencer. Ne serait-il donc pas plus intéressant de définir une fonction qui soit capable d'afficher n'importe quelle table, à la demande ?

Lorsque nous appellerons cette fonction, nous devrons bien évidemment pouvoir lui indiquer quelle table nous souhaitons afficher. Cette information que nous voulons transmettre à la fonction au moment même où nous l'appelons s'appelle un ***argument***. Nous avons déjà rencontré à plusieurs reprises des fonctions intégrées qui utilisent des arguments. La fonction **sin(a)**, par exemple, calcule le sinus de l'angle **a**. La fonction **sin()** utilise donc la valeur numérique de **a** comme argument pour effectuer son travail.

Dans la définition d'une telle fonction, il faut prévoir une variable particulière pour recevoir l'argument transmis. Cette variable particulière s'appelle un ***paramètre***. On lui choisit un nom en respectant les mêmes règles de syntaxe que d'habitude (pas de lettres accentuées, etc.), et on place ce nom entre les parenthèses qui accompagnent la définition de la fonction.

Voici ce que cela donne dans le cas qui nous intéresse :

```
>>> def table(base):
...     n = 1
...     while n <11 :
...         print(n * base, end =' ')
...         n = n +1
```

La fonction **table()** telle que définie ci-dessus utilise le paramètre **base** pour calculer les dix premiers termes de la table de multiplication correspondante.

Pour tester cette nouvelle fonction, il nous suffit de l'appeler avec un argument. Exemples :

```
>>> table(13)
13 26 39 52 65 78 91 104 117 130

>>> table(9)
9 18 27 36 45 54 63 72 81 90
```

Dans ces exemples, la valeur que nous indiquons entre parenthèses lors de l'appel de la fonction (et qui est donc un argument) est automatiquement affectée au paramètre **base**. Dans le corps de la fonction, **base** joue le même rôle que n'importe quelle autre variable. Lorsque nous entrons la commande **table(9)**, nous signifions à la machine que nous voulons exécuter la fonction **table()** en affectant la valeur **9** à la variable **base**.

Utilisation d'une variable comme argument

Dans les 2 exemples qui précèdent, l'argument que nous avons utilisé en appelant la fonction **table()** était à chaque fois une constante (la valeur 13, puis la valeur 9). Cela n'est nullement obligatoire. L'argument que nous utilisons dans l'appel d'une fonction peut être une variable lui aussi, comme dans l'exemple ci-dessous. Analysez bien cet exemple, essayez-le concrètement, et décrivez le mieux possible dans votre cahier d'exercices ce que vous obtenez, en expliquant avec vos propres mots ce qui se passe. Cet exemple devrait vous donner un premier aperçu de l'utilité des fonctions pour accomplir simplement des tâches complexes :

```
>>> a = 1
>>> while a <20:
...     table(a)
...     a = a +1
...
```

Remarque importante

Dans l'exemple ci-dessus, l'argument que nous passons à la fonction **table()** est le contenu de la variable **a**. À l'intérieur de la fonction, cet argument est affecté au paramètre **base**, qui est une tout autre variable. Notez donc bien dès à présent que :

> *Le nom d'une variable que nous passons comme argument n'a rien à voir avec le nom du paramètre correspondant dans la fonction.*

Ces noms peuvent être identiques si vous le voulez, mais vous devez bien comprendre qu'ils ne désignent pas la même chose (en dépit du fait qu'ils puissent éventuellement contenir une valeur identique).

Exercice

7.1 Importez le module ***turtle*** pour pouvoir effectuer des dessins simples.

Vous allez dessiner une série de triangles équilatéraux de différentes couleurs.
Pour ce faire, définissez d'abord une fonction **triangle()** capable de dessiner un triangle d'une couleur bien déterminée (ce qui signifie donc que la définition de votre fonction doit comporter un paramètre pour recevoir le nom de cette couleur).

Utilisez ensuite cette fonction pour reproduire ce même triangle en différents endroits, en changeant de couleur à chaque fois.

Fonction avec plusieurs paramètres

La fonction **table()** est certainement intéressante, mais elle n'affiche toujours que les dix premiers termes de la table de multiplication, alors que nous pourrions souhaiter qu'elle en affiche d'autres. Qu'à cela ne tienne. Nous allons l'améliorer en lui ajoutant des paramètres supplémentaires, dans une nouvelle version que nous appellerons cette fois **tableMulti()** :

```
>>> def tableMulti(base, debut, fin):
...     print('Fragment de la table de multiplication par', base, ':')
...     n = debut
...     while n <= fin :
...         print(n, 'x', base, '=', n * base)
...         n = n +1
```

Cette nouvelle fonction utilise donc trois paramètres : la base de la table comme dans l'exemple précédent, l'indice du premier terme à afficher, l'indice du dernier terme à afficher.

Essayons cette fonction en entrant par exemple :

```
>>> tableMulti(8, 13, 17)
```

ce qui devrait provoquer l'affichage de :

```
Fragment de la table de multiplication par 8 :
13 x 8 = 104
14 x 8 = 112
15 x 8 = 120
16 x 8 = 128
17 x 8 = 136
```

Notes

- Pour définir une fonction avec plusieurs paramètres, il suffit d'inclure ceux-ci entre les parenthèses qui suivent le nom de la fonction, en les séparant à l'aide de virgules.

- Lors de l'appel de la fonction, les arguments utilisés doivent être fournis *dans le même ordre* que celui des paramètres correspondants (en les séparant eux aussi à l'aide de virgules). Le premier argument sera affecté au premier paramètre, le second argument sera affecté au second paramètre, et ainsi de suite.
- À titre d'exercice, essayez la séquence d'instructions suivantes et décrivez dans votre cahier d'exercices le résultat obtenu :

```
>>> t, d, f = 11, 5, 10
>>> while t<21:
...     tableMulti(t,d,f)
...     t, d, f = t +1, d +3, f +5
...
```

Variables locales, variables globales

Lorsque nous définissons des variables à l'intérieur du corps d'une fonction, ces variables ne sont accessibles qu'à la fonction elle-même. On dit que ces variables sont des *variables locales* à la fonction. C'est par exemple le cas des variables **base**, **debut**, **fin** et **n** dans l'exercice précédent.

Chaque fois que la fonction **tableMulti()** est appelée, Python réserve pour elle (dans la mémoire de l'ordinateur) un nouvel *espace de noms*[34]. Les contenus des variables **base**, **debut**, **fin** et **n** sont stockés dans cet espace de noms qui est *inaccessible depuis l'extérieur de la fonction*. Ainsi par exemple, si nous essayons d'afficher le contenu de la variable **base** juste après avoir effectué l'exercice ci-dessus, nous obtenons un message d'erreur :

```
>>> print(base)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'base' is not defined
```

La machine nous signale clairement que le symbole **base** lui est inconnu, alors qu'il était correctement imprimé par la fonction **tableMulti()** elle-même. L'espace de noms qui contient le symbole **base** est strictement réservé au fonctionnement interne de **tableMulti()**, et il est automatiquement détruit dès que la fonction a terminé son travail.

Les variables définies à l'extérieur d'une fonction sont des *variables globales*. Leur contenu est « visible » de l'intérieur d'une fonction, mais la fonction *ne peut pas le modifier*. Exemple :

```
>>> def mask():
...     p = 20
...     print(p, q)
...
>>> p, q = 15, 38
>>> mask()
20 38
>>> print(p, q)
15 38
```

Analysons attentivement cet exemple :

[34] Ce concept d'*espace de noms* sera approfondi progressivement. Vous apprendrez également plus loin que les fonctions sont en fait des *objets* dont on crée à chaque fois une nouvelle *instance* lorsqu'on les appelle.

Nous commençons par définir une fonction très simple (qui n'utilise d'ailleurs aucun paramètre). À l'intérieur de cette fonction, une variable **p** est définie, avec **20** comme valeur initiale. Cette variable **p** qui est définie à l'intérieur d'une fonction sera donc une *variable locale*.

Une fois la définition de la fonction terminée, nous revenons au niveau principal pour y définir les deux variables **p** et **q** auxquelles nous attribuons les contenus **15** et **38**. Ces deux variables définies au niveau principal seront donc des *variables globales*.

Ainsi le même nom de variable **p** a été utilisé ici à deux reprises, *pour définir deux variables différentes* : l'une est globale et l'autre est locale. On peut constater dans la suite de l'exercice que ces deux variables sont bel et bien des variables distinctes, indépendantes, obéissant à une règle de priorité qui veut qu'à l'intérieur d'une fonction (où elles pourraient entrer en compétition), ce sont les variables définies localement qui ont la priorité.

On constate en effet que lorsque la fonction **mask()** est lancée, la variable globale **q** y est accessible, puisqu'elle est imprimée correctement. Pour **p**, par contre, c'est la valeur attribuée localement qui est affichée.

On pourrait croire d'abord que la fonction **mask()** a simplement modifié le contenu de la variable globale **p** (puisqu'elle est accessible). Les lignes suivantes démontrent qu'il n'en est rien : en dehors de la fonction **mask()**, la variable globale **p** conserve sa valeur initiale.

Tout ceci peut vous paraître compliqué au premier abord. Vous comprendrez cependant très vite combien il est utile que des variables soient ainsi définies comme étant locales, c'est-à-dire en quelque sorte confinées à l'intérieur d'une fonction. Cela signifie en effet que vous pourrez toujours utiliser une infinité de fonctions sans vous préoccuper le moins du monde des noms de variables qui y sont utilisées : ces variables ne pourront en effet jamais interférer avec celles que vous aurez vous-même définies par ailleurs.

Cet état de choses peut toutefois être modifié si vous le souhaitez. Il peut se faire par exemple que vous ayez à définir une fonction qui soit capable de modifier une variable globale. Pour atteindre ce résultat, il vous suffira d'utiliser l'instruction **global**. Cette instruction permet d'indiquer – à l'intérieur de la définition d'une fonction – quelles sont les variables à traiter globalement.

Dans l'exemple ci-dessous, la variable **a** utilisée à l'intérieur de la fonction **monter()** est non seulement accessible, mais également modifiable, parce qu'elle est signalée explicitement comme étant une variable qu'il faut traiter globalement. Par comparaison, essayez le même exercice en supprimant l'instruction **global** : la variable **a** n'est plus incrémentée à chaque appel de la fonction.

```
>>> def monter():
...     global a
...     a = a+1
...     print(a)
...
>>> a = 15
>>> monter()
16
>>> monter()
17
>>>
```

Vraies fonctions et procédures

Pour les puristes, les fonctions que nous avons décrites jusqu'à présent ne sont pas tout à fait des fonctions au sens strict, mais plus exactement des *procédures*[35]. Une « vraie » fonction (au sens strict) doit en effet *renvoyer une valeur* lorsqu'elle se termine. Une « vraie » fonction peut s'utiliser à la droite du signe égale dans des expressions telles que `y = sin(a)`. On comprend aisément que dans cette expression, la fonction **sin()** renvoie une valeur (le sinus de l'argument **a**) qui est directement affectée à la variable **y**.

Commençons par un exemple extrêmement simple :

```
>>> def cube(w):
...     return w*w*w
...
```

L'instruction **return** définit ce que doit être la valeur renvoyée par la fonction. En l'occurrence, il s'agit du cube de l'argument qui a été transmis lors de l'appel de la fonction. Exemple :

```
>>> b = cube(9)
>>> print(b)
729
```

À titre d'exemple un peu plus élaboré, nous allons maintenant modifier quelque peu la fonction **table()** sur laquelle nous avons déjà pas mal travaillé, afin qu'elle renvoie elle aussi une valeur. Cette valeur sera en l'occurrence une liste (la liste des dix premiers termes de la table de multiplication choisie). Voilà donc une occasion de reparler des listes. Dans la foulée, nous en profiterons pour apprendre encore un nouveau concept :

```
>>> def table(base):
...     resultat = []             # resultat est d'abord une liste vide
...     n = 1
...     while n < 11:
...         b = n * base
...         resultat.append(b)    # ajout d'un terme à la liste
...         n = n +1              # (voir explications ci-dessous)
...     return resultat
...
```

Pour tester cette fonction, nous pouvons entrer par exemple :

```
>>> ta9 = table(9)
```

Ainsi nous affectons à la variable **ta9** les dix premiers termes de la table de multiplication par 9, sous la forme d'une liste :

```
>>> print(ta9)
[9, 18, 27, 36, 45, 54, 63, 72, 81, 90]
>>> print(ta9[0])
9
>>> print(ta9[3])
36
>>> print(ta9[2:5])
[27, 36, 45]
>>>
```

[35] Dans certains langages de programmation, les fonctions et les procédures sont définies à l'aide d'instructions différentes. Python utilise la même instruction `def` pour définir les unes et les autres.

(Rappel : le premier élément d'une liste correspond à l'indice 0).

Notes

- Comme nous l'avons vu dans l'exemple précédent, l'instruction **return** définit ce que doit être la valeur « renvoyée » par la fonction. En l'occurrence, il s'agit ici du contenu de la variable **resultat**, c'est-à-dire la liste des nombres générés par la fonction[36].
- L'instruction **resultat.append(b)** est notre second exemple de l'utilisation d'un concept important sur lequel nous reviendrons encore abondamment par la suite : dans cette instruction, nous appliquons la *méthode* **append()** à l'*objet* **resultat**.

  Nous préciserons petit à petit ce qu'il faut entendre par *objet* en programmation. Pour l'instant, admettons simplement que ce terme très général s'applique notamment aux *listes* de Python. Une *méthode* n'est en fait rien d'autre qu'une fonction (que vous pouvez d'ailleurs reconnaître comme telle à la présence des parenthèses), mais *une fonction qui est associée à un objet*. Elle fait partie de la définition de cet objet, ou plus précisément de la *classe* particulière à laquelle cet objet appartient (nous étudierons ce concept de classe plus tard).

  Mettre en œuvre une méthode associée à un objet consiste en quelque sorte à « faire fonctionner » cet objet d'une manière particulière. Par exemple, on met en œuvre la méthode **methode4()** d'un objet **objet3**, à l'aide d'une instruction du type : **objet3.methode4()** , c'est-à-dire le nom de l'objet, puis le nom de la méthode, reliés l'un à l'autre par un point. Ce point joue un rôle essentiel : on peut le considérer comme un véritable *opérateur*.

  Dans notre exemple, nous appliquons donc la méthode **append()** à l'objet **resultat**, qui est une liste. Sous Python, les *listes* constituent donc une *classe* particulière d'objets, auxquels on peut effectivement appliquer toute une série de *méthodes*. En l'occurrence, la méthode **append()** des objets « listes » sert à leur ajouter un élément par la fin. L'élément à ajouter est transmis entre les parenthèses, comme tout argument qui se respecte.
- Nous aurions obtenu un résultat similaire si nous avions utilisé à la place de cette instruction une expression telle que « `resultat = resultat + [b]` » (l'opérateur de concaténation fonctionne en effet aussi avec les listes). Cette façon de procéder est cependant moins rationnelle et beaucoup moins efficace, car elle consiste à redéfinir à chaque itération de la boucle une nouvelle liste **resultat**, dans laquelle la totalité de la liste précédente est à chaque fois recopiée avec ajout d'un élément supplémentaire.
  Lorsque l'on utilise la méthode **append()**, par contre, l'ordinateur procède bel et bien à une modification de la liste existante (sans la recopier dans une nouvelle variable). Cette technique est donc préférable, car elle mobilise moins lourdement les ressources de l'ordinateur, et elle est plus rapide (surtout lorsqu'il s'agit de traiter des listes volumineuses).
- Il n'est pas du tout indispensable que la valeur renvoyée par une fonction soit affectée à une variable (comme nous l'avons fait jusqu'ici dans nos exemples par souci de clarté). Ainsi, nous aurions pu tester les fonction **cube()** et **table()** en entrant les commandes :

```
>>> print(cube(9))
>>> print(table(9))
>>> print(table(9)[3])
```

[36] `return` peut également être utilisé sans aucun argument, à l'intérieur d'une fonction, pour provoquer sa fermeture immédiate. La valeur retournée dans ce cas est l'objet `None` (objet particulier, correspondant à « rien »).

ou encore plus simplement encore :

```
>>> cube(9)...
```

Utilisation des fonctions dans un script

Pour cette première approche des fonctions, nous n'avons utilisé jusqu'ici que le mode interactif de l'interpréteur Python.

Il est bien évident que les fonctions peuvent aussi s'utiliser dans des scripts. Veuillez donc essayer vous-même le petit programme ci-dessous, lequel calcule le volume d'une sphère à l'aide de la formule que vous connaissez certainement : $V = \frac{4}{3} \pi R^3$

```
def cube(n):
   return n**3

def volumeSphere(r):
   return 4 * 3.1416 * cube(r) / 3

r = input('Entrez la valeur du rayon : ')
print('Le volume de cette sphère vaut', volumeSphere(float(r)))
```

Notes

À bien y regarder, ce programme comporte trois parties : les deux fonctions **cube()** et **volumeSphere()**, et ensuite le corps principal du programme.

Dans le corps principal du programme, on appelle la fonction **volumeSphere()**, en lui transmettant la valeur entrée par l'utilisateur pour le rayon, préalablement convertie en un nombre réel à l'aide de la fonction intégrée **float()**.

À l'intérieur de la fonction **volumeSphere()**, il y a un appel de la fonction **cube()**.

Notez bien que les trois parties du programme ont été disposées dans un certain ordre : *d'abord la définition des fonctions, et ensuite le corps principal du programme.* Cette disposition est nécessaire, parce que l'interpréteur exécute les lignes d'instructions du programme l'une après l'autre, dans l'ordre où elles apparaissent dans le code source.

> *Dans un script, la définition des fonctions doit précéder leur utilisation.*

Pour vous en convaincre, intervertissez cet ordre (en plaçant par exemple le corps principal du programme au début), et prenez note du type de message d'erreur qui est affiché lorsque vous essayez d'exécuter le script ainsi modifié.

En fait, le corps principal d'un programme Python constitue lui-même une entité un peu particulière, qui est toujours reconnue dans le fonctionnement interne de l'interpréteur sous le nom réservé **__main__** (le mot « main » signifie « principal », en anglais. Il est encadré par des caractères « souligné » en double, pour éviter toute confusion avec d'autres symboles). L'exécution d'un script commence toujours avec la première instruction de cette entité **__main__**, où qu'elle puisse se trouver dans le listing. Les instructions qui suivent sont alors exécutées l'une après l'autre, dans l'ordre, jus-

qu'au premier appel de fonction. Un appel de fonction est comme un détour dans le flux de l'exécution : au lieu de passer à l'instruction suivante, l'interpréteur exécute la fonction appelée, puis revient au programme appelant pour continuer le travail interrompu. Pour que ce mécanisme puisse fonctionner, il faut que l'interpréteur ait pu lire la définition de la fonction avant l'entité **__main__**, et celle-ci sera donc placée en général à la fin du script.

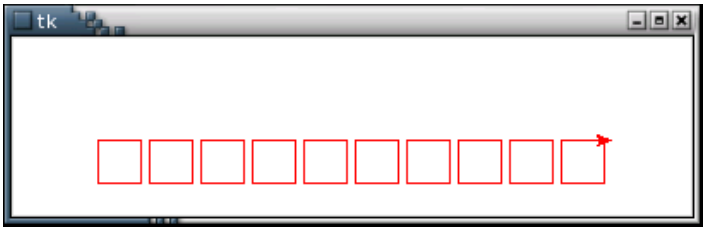
Dans notre exemple, l'entité **__main__** appelle une première fonction qui elle-même en appelle une deuxième. Cette situation est très fréquente en programmation. Si vous voulez comprendre correctement ce qui se passe dans un programme, vous devez donc apprendre à lire un script, non pas de la première à la dernière ligne, mais plutôt en suivant un cheminement analogue à ce qui se passe lors de l'exécution de ce script. Cela signifie donc concrètement que vous devrez souvent analyser un script en commençant par ses dernières lignes !

Modules de fonctions

Afin que vous puissiez mieux comprendre encore la distinction entre la définition d'une fonction et son utilisation au sein d'un programme, nous vous suggérons de placer fréquemment vos définitions de fonctions dans un module Python, et le programme qui les utilise dans un autre.

Exemple :

On souhaite réaliser la série de dessins ci-contre, à l'aide du module ***turtle*** :



Écrivez les lignes de code suivantes, et sauvegardez-les dans un fichier auquel vous donnerez le nom ***dessins_tortue.py*** :

```
from turtle import *

def carre(taille, couleur):
    "fonction qui dessine un carré de taille et de couleur déterminées"
    color(couleur)
    c =0
    while c <4:
        forward(taille)
        right(90)
        c = c +1
```

Vous pouvez remarquer que la définition de la fonction **carre()** commence par une chaîne de caractères. Cette chaîne ne joue aucun rôle fonctionnel dans le script : elle est traitée par Python *comme un simple commentaire*, mais qui est mémorisé à part dans un système de documentation interne automatique, lequel pourra ensuite être exploité par certains utilitaires et éditeurs « intelligents ».

Si vous programmez dans l'environnement IDLE, par exemple, vous verrez apparaître cette chaîne documentaire dans une « bulle d'aide », chaque fois que vous ferez appel aux fonctions ainsi documentées.

En fait, Python place cette chaîne dans une variable spéciale dont le nom est **__doc__** (le mot « doc » entouré de deux paires de caractères « souligné »), et qui est associée à l'objet fonction comme étant l'un de ses attributs (vous en apprendrez davantage au sujet de ces attributs lorsque nous aborderons

les classes d'objets, page 166). Ainsi, vous pouvez vous-même retrouver la chaîne de documentation d'une fonction quelconque en affichant le contenu de cette variable. Exemple :

```
>>> def essai():
...     "Cette fonction est bien documentée mais ne fait presque rien."
...     print("rien à signaler")
...
>>> essai()
rien à signaler
>>> print(essai.__doc__)
Cette fonction est bien documentée mais ne fait presque rien.
```

Prenez donc la peine d'incorporer une telle chaîne explicative dans toutes vos définitions de fonctions futures : il s'agit là d'une pratique hautement recommandable.

Le fichier que vous aurez créé ainsi est dorénavant un véritable *module de fonctions* Python, au même titre que les modules *turtle* ou *math* que vous connaissez déjà. Vous pouvez donc l'utiliser dans n'importe quel autre script, comme celui-ci, par exemple, qui effectuera le travail demandé :

```
from dessins_tortue import *

up()                    # relever le crayon
goto(-150, 50)          # reculer en haut à gauche

# dessiner dix carrés rouges, alignés :
i = 0
while i < 10:
    down()              # abaisser le crayon
    carre(25, 'red')    # tracer un carré
    up()                # relever le crayon
    forward(30)         # avancer + loin
    i = i +1
a = input()             # attendre
```

> ***Attention***
> *Vous pouvez à priori nommer vos modules de fonctions comme bon vous semble. Sachez cependant qu'il vous sera impossible d'importer un module si son nom est l'un des 33 mots réservés Python signalés à la page 14, car le nom du module importé deviendrait une variable dans votre script, et les mots réservés ne peuvent pas être utilisés comme noms de variables. Rappelons aussi qu'il vous faut éviter de donner à vos modules – et à tous vos scripts en général – le même nom que celui d'un module Python préexistant, sinon vous devez vous attendre à des conflits. Par exemple, si vous donnez le nom* ***turtle.py*** *à un exercice dans lequel vous avez placé une instruction d'importation du module* ***turtle****, c'est l'exercice lui-même que vous allez importer !*

Résumé : structure d'un programme Python type

```
# -*- coding:Utf8 -*-

####################################
# Programme Python type            #
# auteur : G.Swinnen, Liège, 2009  #
# licence : GPL                    #
####################################


####################################
# Importation de fonctions externes :

from math import sqrt


#################################
# Définition locale de fonctions :

def occurrences(car, ch):
    "Cette fonction renvoie le \
     nombre de caractères <car> \
     contenus dans la chaîne <ch>"

    nc = 0

    i = 0

    while i < len(ch):

        if ch[i] == car:
            nc = nc + 1

        i = i + 1

    return nc


#################################
# Corps principal du programme :

print("Veuillez entrer un nombre :")
nbr = eval(input())

print("Veuillez entrer une phrase :")
phr = input()
print("Entrez le caractère à compter :")
cch = input()

no = occurrences(cch, phr)
rc = sqrt(nbr**3)

print("La racine carrée du cube", end=' ')
print("du nombre fourni vaut", end=' ')
print(rc)

print("La phrase contient", end=' ')
print(no, "caractères", cch)
```

Un programme Python contient en général les blocs suivants, dans l'ordre :
- Quelques instructions d'initialisation (importation de fonctions et/ou de classes, définition éventuelle de variables globales).
- Les définitions locales de fonctions et/ou de classes.
- Le corps principal du programme.

Le programme peut utiliser un nombre quelconque de fonctions, lesquelles sont définies localement ou importées depuis des modules externes.
Vous pouvez vous-même définir de tels modules.

La définition d'une fonction comporte souvent une liste de PARAMÈTRES.
Ce sont toujours des VARIABLES, qui recevront leur valeur lorsque la fonction sera appelée.

Une boucle de répétition de type 'while' doit toujours inclure au moins quatre éléments :
- l'initialisation d'une variable 'compteur' ;
- l'instruction while proprement dite, dans laquelle on exprime la condition de répétition des instructions qui suivent ;
- le bloc d'instructions à répéter ;
- une instruction d'incrémentation du compteur.

La fonction "renvoie" toujours une valeur bien déterminée au programme appelant.
Si l'instruction 'return' n'est pas utilisée, ou si elle est utilisée sans argument, la fonction renvoie un objet vide : 'None'.

Le programme qui fait appel à une fonction lui transmet d'habitude une série d'ARGUMENTS, lesquels peuvent être des valeurs, des variables, ou même des expressions.

Exercices

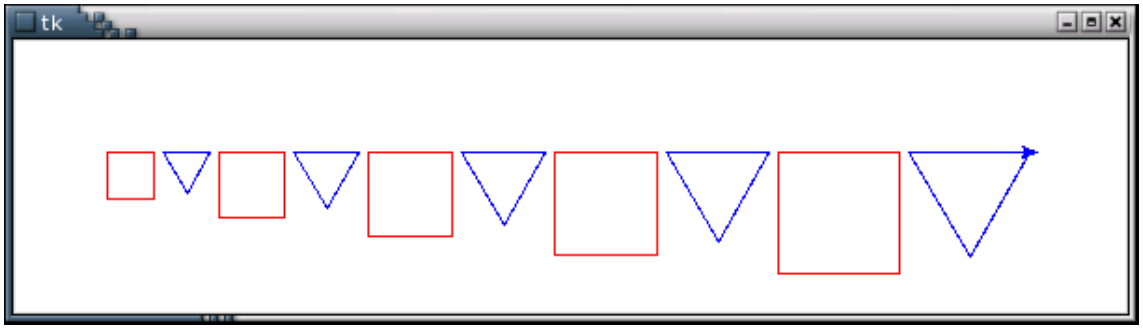
7.2 Définissez une fonction **ligneCar(n, ca)** qui renvoie une chaîne de **n** caractères **ca**.

7.3 Définissez une fonction **surfCercle(R)**. Cette fonction doit renvoyer la surface (l'aire) d'un cercle dont on lui a fourni le rayon **R** en argument. Par exemple, l'exécution de l'instruction : `print(surfCercle(2.5))` doit donner le résultat : `19.63495...`

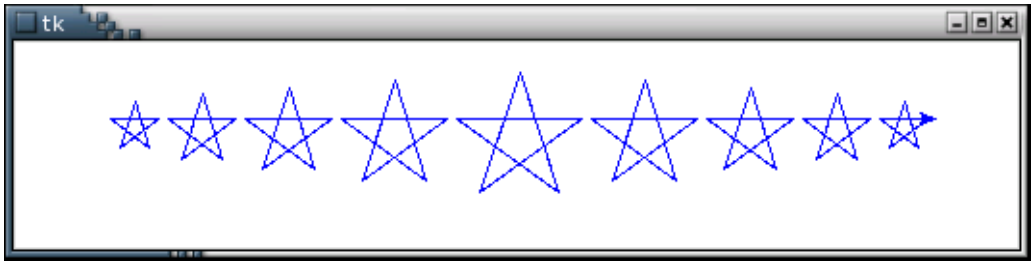
7.4 Définissez une fonction **volBoite(x1,x2,x3)** qui renvoie le volume d'une boîte parallélépipédique dont on fournit les trois dimensions **x1, x2, x3** en arguments.
Par exemple, l'exécution de l'instruction :
`print(volBoite(5.2, 7.7, 3.3))` doit donner le résultat : `132.132.`

7.5 Définissez une fonction **maximum(n1,n2,n3)** qui renvoie le plus grand de 3 nombres **n1, n2, n3** fournis en arguments. Par exemple, l'exécution de l'instruction :
`print(maximum(2,5,4))` doit donner le résultat : `5.`

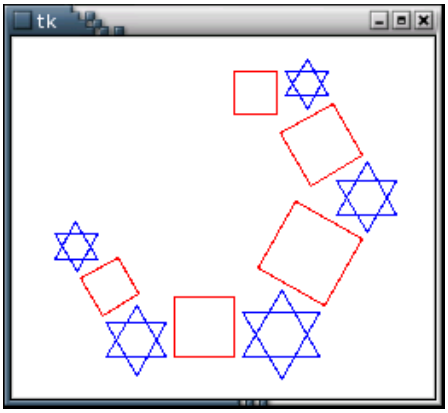
7.6 Complétez le module de fonctions graphiques **dessins_tortue.py** décrit à la page 71.
Commencez par ajouter un paramètre **angle** à la fonction **carre()**, de manière à ce que les carrés puissent être tracés dans différentes orientations.
Définissez ensuite une fonction **triangle(taille, couleur, angle)** capable de dessiner un triangle équilatéral d'une taille, d'une couleur et d'une orientation bien déterminées.
Testez votre module à l'aide d'un programme qui fera appel à ces fonctions à plusieurs reprises, avec des arguments variés pour dessiner une série de carrés et de triangles :

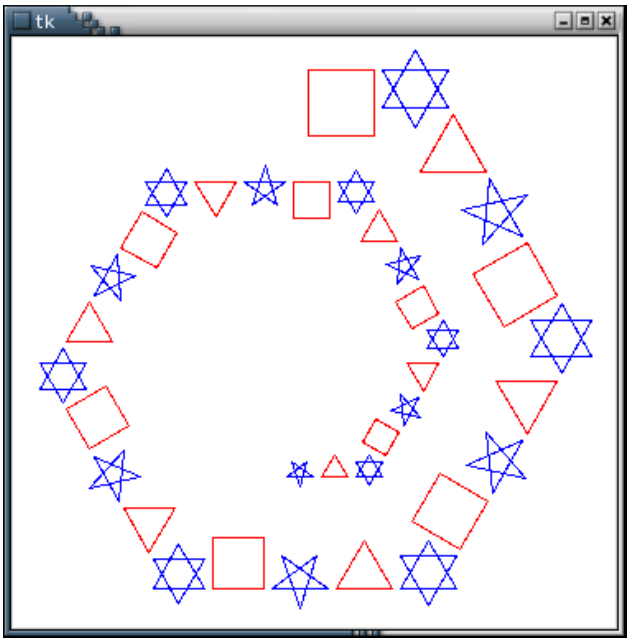


7.7 Ajoutez au module de l'exercice précédent une fonction **etoile5()** spécialisée dans le dessin d'étoiles à 5 branches. Dans votre programme principal, insérez une boucle qui dessine une rangée horizontale de de 9 petites étoiles de tailles variées :



7.8 Ajoutez au module de l'exercice précédent une fonction **etoile6()** capable de dessiner une étoile à 6 branches, elle-même constituée de deux triangles équilatéraux imbriqués. Cette nouvelle fonction devra faire appel à la fonction **triangle()** définie précédemment.
Votre programme principal dessinera également une série de ces étoiles :





7.9 Définissez une fonction **compteCar(ca,ch)** qui renvoie le nombre de fois que l'on rencontre le caractère **ca** dans la chaîne de caractères **ch**. Par exemple, l'exécution de l'instruction :
`print(compteCar('e', 'Cette phrase est un exemple'))` doit donner le résultat : `7`

7.10 Définissez une fonction **indexMax(liste)** qui renvoie l'index de l'élément ayant la valeur la plus élevée dans la liste transmise en argument. Exemple d'utilisation :

```
serie = [5, 8, 2, 1, 9, 3, 6, 7]
print(indexMax(serie))
4
```

7.11 Définissez une fonction **nomMois(n)** qui renvoie le nom du énième mois de l'année.
Par exemple, l'exécution de l'instruction :
`print(nomMois(4))` doit donner le résultat : `Avril.`

7.12 Définissez une fonction **inverse(ch)** qui permette d'inverser les l'ordre des caractères d'une chaîne quelconque. La chaîne inversée sera renvoyée au programme appelant.

7.13 Définissez une fonction **compteMots(ph)** qui renvoie le nombre de mots contenus dans la phrase **ph**. On considère comme mots les ensembles de caractères inclus entre des espaces.

Typage des paramètres

Vous avez appris que le *typage* des variables sous Python est un *typage dynamique*, ce qui signifie que le type d'une variable est défini au moment où on lui affecte une valeur. Ce mécanisme fonctionne aussi pour les paramètres d'une fonction. Le type d'un paramètre devient automatiquement le même que celui de l'argument qui a été transmis à la fonction. Exemple :

```
>>> def afficher3fois(arg):
...     print(arg, arg, arg)
...

>>> afficher3fois(5)
5 5 5

>>> afficher3fois('zut')
zut zut zut

>>> afficher3fois([5, 7])
[5, 7] [5, 7] [5, 7]

>>> afficher3fois(6**2)
36 36 36
```

Dans cet exemple, vous pouvez constater que la même fonction **afficher3fois()** accepte dans tous les cas l'argument qu'on lui transmet, que cet argument soit un nombre, une chaîne de caractères, une liste, *ou même une expression*. Dans ce dernier cas, Python commence par évaluer l'expression, et c'est le résultat de cette évaluation qui est transmis comme argument à la fonction.

Valeurs par défaut pour les paramètres

Dans la définition d'une fonction, il est possible (et souvent souhaitable) de définir un argument par défaut pour chacun des paramètres. On obtient ainsi une fonction *qui peut être appelée avec une partie seulement des arguments attendus*. Exemples :

```
>>> def politesse(nom, vedette ='Monsieur'):
...    print("Veuillez agréer ,", vedette, nom, ", mes salutations cordiales.")
...

>>> politesse('Dupont')
Veuillez agréer , Monsieur Dupont , mes salutations cordiales.

>>> politesse('Durand', 'Mademoiselle')
Veuillez agréer , Mademoiselle Durand , mes salutations cordiales.
```

Lorsque l'on appelle cette fonction en ne lui fournissant que le premier argument, le second reçoit tout de même une valeur par défaut. Si l'on fournit les deux arguments, la valeur par défaut pour le deuxième est tout simplement ignorée.

Vous pouvez définir une valeur par défaut pour tous les paramètres, ou une partie d'entre eux seulement. Dans ce cas, cependant, *les paramètres sans valeur par défaut doivent précéder les autres* dans la liste. Par exemple, la définition ci-dessous est incorrecte :

```
>>> def politesse(vedette ='Monsieur', nom):
```

Autre exemple :

```
>>> def question(annonce, essais =4, please ='Oui ou non, s.v.p.!'):
...     while essais >0:
...         reponse = input(annonce)
...         if reponse in ('o', 'oui','O','Oui','OUI'):
...             return 1
...         if reponse in ('n','non','N','Non','NON'):
...             return 0
...         print(please)
...         essais = essais-1
...
>>>
```

Cette fonction peut être appelée de différentes façons, telles par exemple :

```
rep = question('Voulez-vous vraiment terminer ? ')
```

ou bien :

```
rep = question('Faut-il effacer ce fichier ? ', 3)
```

ou même encore :

```
rep = question('Avez-vous compris ? ', 2, 'Répondez par oui ou par non !')
```

Prenez la peine d'essayer et de décortiquer cet exemple.

Arguments avec étiquettes

Dans la plupart des langages de programmation, les arguments que l'on fournit lors de l'appel d'une fonction doivent être fournis *exactement dans le même ordre* que celui des paramètres qui leur correspondent dans la définition de la fonction.

Python autorise cependant une souplesse beaucoup plus grande. Si les paramètres annoncés dans la définition de la fonction ont reçu chacun une valeur par défaut, sous la forme déjà décrite ci-dessus, on peut faire appel à la fonction en fournissant les arguments correspondants *dans n'importe quel ordre, à la condition de désigner nommément les paramètres correspondants*. Exemple :

```
>>> def oiseau(voltage=100, etat='allumé', action='danser la java'):
...     print('Ce perroquet ne pourra pas', action)
...     print('si vous le branchez sur', voltage, 'volts !')
...     print("L'auteur de ceci est complètement", etat)
...

>>> oiseau(etat='givré', voltage=250, action='vous approuver')
Ce perroquet ne pourra pas vous approuver
si vous le branchez sur 250 volts !
L'auteur de ceci est complètement givré

>>> oiseau()
Ce perroquet ne pourra pas danser la java
si vous le branchez sur 100 volts !
L'auteur de ceci est complètement allumé
```

Exercices

7.14 Modifiez la fonction **volBoite(x1,x2,x3)** que vous avez définie dans un exercice précédent, de manière à ce qu'elle puisse être appelée avec trois, deux, un seul, ou même aucun argument. Utilisez pour ceux ci des valeurs par défaut égales à 10.
Par exemple :

`print(volBoite())`	doit donner le résultat : `1000`
`print(volBoite(5.2))`	doit donner le résultat : `520.0`
`print(volBoite(5.2, 3))`	doit donner le résultat : `156.0`

7.15 Modifiez la fonction **volBoite(x1,x2,x3)** ci-dessus de manière à ce qu'elle puisse être appelée avec un, deux, ou trois arguments. Si un seul est utilisé, la boîte est considérée comme cubique (l'argument étant l'arête de ce cube). Si deux sont utilisés, la boîte est considérée comme un prisme à base carrée (auquel cas le premier argument est le côté du carré, et le second la hauteur du prisme). Si trois arguments sont utilisés, la boîte est considérée comme un parallélépipède. Par exemple :

`print(volBoite())`	doit donner le résultat : `-1` (indication d'une erreur)
`print(volBoite(5.2))`	doit donner le résultat : `140.608`
`print(volBoite(5.2, 3))`	doit donner le résultat : `81.12`
`print(volBoite(5.2, 3, 7.4))`	doit donner le résultat : `115.44`

7.16 Définissez une fonction **changeCar(ch,ca1,ca2,debut,fin)** qui remplace tous les caractères **ca1** par des caractères **ca2** dans la chaîne de caractères **ch**, à partir de l'indice **debut** et jusqu'à l'indice **fin**, ces deux derniers arguments pouvant être omis (et dans ce cas la chaîne est traitée d'une extrémité à l'autre). Exemples de la fonctionnalité attendue :

```
>>> phrase = 'Ceci est une toute petite phrase.'
>>> print(changeCar(phrase, ' ', '*'))
Ceci*est*une*toute*petite*phrase.
>>> print(changeCar(phrase, ' ', '*', 8, 12))
Ceci est*une*toute petite phrase.
>>> print(changeCar(phrase, ' ', '*', 12))
Ceci est une*toute*petite*phrase.
>>> print(changeCar(phrase, ' ', '*', fin = 12))
Ceci*est*une*toute petite phrase.
```

7.17 Définissez une fonction **eleMax(liste,debut,fin)** qui renvoie l'élément ayant la plus grande valeur dans la liste transmise. Les deux arguments **debut** et **fin** indiqueront les indices entre lesquels doit s'exercer la recherche, et chacun d'eux pourra être omis (comme dans l'exercice précédent). Exemples de la fonctionnalité attendue :

```
>>> serie = [9, 3, 6, 1, 7, 5, 4, 8, 2]
>>> print(eleMax(serie))
9
>>> print(eleMax(serie, 2, 5))
7
>>> print(eleMax(serie, 2))
8
>>> print(eleMax(serie, fin =3, debut =1))
6
```