10 projets à réaliser dès 10 ans

3e édition

Programmer avec Python

EN S'AMUSANT

Apprendre à coder dans un langage de pro

Concevoir des jeux et des programmes

Personnaliser ses projets

BRENDAN SCOTT

pour les nuls

Programmer avec Python en s'amusant

pour les nuls

3e édition

Brendan Scott

Version française : Olivier Engler

FIRST INTERACTIVE

Programmer avec Python en s'amusant 3e édition pour les Nuls

Collection dirigée par Jean-Pierre Cano
Traduction : Olivier Engler
Mise en page : Marie Housseau

Edition française publiée en accord avec Wiley Publishing, Inc.

Éditions First, un département d'Édi8
12 avenue d'Italie
75013 Paris
Tél. : 01 44 16 09 00
Fax : 01 44 16 09 01
E-mail : firstinfo@efirst.com
Web : www.editionsfirst.fr
ISBN : 978-2-412-05608-0
ISBN numérique : 9782412058848
Dépôt légal : 2e trimestre 2020

Ce livre numérique a été converti initialement au format EPUB par Isako www.isako.com à partir de l'édition papier du même ouvrage.

Introduction

Bienvenue dans ce livre ! Tu vas y découvrir tout ce qu'il faut savoir au sujet du langage Python. Une fois que tu auras réalisé les projets, tu seras devenu autonome ; tu pourras poursuivre seul dans la découverte de ce langage.

Tout au long du livre, tu vas apprendre par la pratique, en saisissant le code source, puis de plus en plus, en réfléchissant au code que tu vas écrire (de préférence avant de regarder ma solution).

Conventions du livre

Voici les règles que je me suis imposé pour rendre la lecture plus facile. Tu peux les parcourir dès maintenant, et surtout y revenir au fur et à mesure de ta progression dans le livre.

Conventions dans le texte

Toutes ces conventions sont là pour te rendre la lecture plus facile.

Mots en italique

Quand je présente un mot en *italique*, c'est qu'il s'agit un concept que je vais expliquer peu de temps après. Supposons que j'écrive que les objets d'une liste sont des *éléments*. Dès que tu vois ce genre de présentation, prépare-toi à trouver une définition un peu plus loin.

Code source

Tout ce qui est du code source Python utilise une `police non proportionnelle`. Lorsque c'est au milieu d'un paragraphe (en ligne), cela se présente comme ceci : `print('Salut !')`, mais des codes plus précis permettent de distinguer les noms de `fonctions` et de `variables`.

Mais la plupart du temps, le code source est présenté sous forme d'une série de lignes indépendantes du texte :

```
print('Salut les amis !')
```

Lorsqu'il s'agit de tout ou partie du code source d'un projet, le code source est précédé d'un titre de listing :

Listing x.y : nomfic.py

```
mot-clé nomfonction():
variable = fonctionprédéfinie("chainedetexte")
# Fin du listing
```

Invite >>>

Les extraits de code source commencent parfois, surtout dans les premiers chapitres, par trois signes `>>>`. Cela signifie que nous utilisons le mode interactif de Python. Ce que tu dois saisir va apparaître après ce symbole `>>>` qui se nomme l'invite, car cela invite à saisir des données.

```
>>> mon_texte = "Salut !"
>>> print(mon_texte)
```

Nombre d'espaces

Le texte source commence normalement contre la marge gauche, mais parfois, il faut indenter des lignes de code source pour qu'elles forment un sous-bloc. Le nombre d'espaces du décalage par rapport à la marge a une énorme importance dans le langage Python. Nous verrons cela plus loin.

Longueur des lignes de code

Techniquement, la longueur d'une ligne de code n'a pas d'importance, mais les règles d'écriture Python conseillent de ne pas utiliser plus de 70 caractères environ (lettres, chiffres, espaces et signes de ponctuation). Dans ce livre, je n'ai pas assez de place en largeur pour imprimer des lignes aussi longues ; je suis limité à environ 60 caractères. J'ai donc réparti les lignes les plus longues sur plusieurs lignes apparentes. Pour que Python comprenne qu'il s'agisse bien de la suite de la même ligne technique, il y a des règles à respecter. J'utilise deux méthodes pour subdiviser une autre langue, les méthodes implicite et explicite :

» **La méthode implicite.** On peut poursuivre sur la ligne suivante lorsqu'on est dans un jeu de parenthèses, mais au niveau d'une virgule. Python va comprendre qu'il s'agit de la suite de la même ligne technique. Il faut que les lignes de suite soient indentées au niveau de la parenthèse ouvrante. Cet exemple est tiré du Projet 9 :

```
valeurs = (e.prenom, e.famille,
           e.naissance, e.email)
```

Dans le livre imprimé, nous voyons deux lignes physiques, mais Python comprend qu'il s'agit d'une seule ligne très longue. La seconde ligne ci-dessus a été créée par appui sur la touche Entrée juste après la virgule de la première ligne, puis par insertion du nombre suffisant d'espaces au début de la nouvelle ligne pour que le premier caractère autre qu'une espace tombe au bon endroit dans le sens vertical.

» **La méthode explicite.** L'autre technique consiste à utiliser le caractère spécialement conçu pour subdiviser une ligne. Il s'agit de la barre oblique inverse \, que j'appelle antibarre. Voici un autre exemple du Projet 9 :

```
invite_saisie = "Choisissez : 1 pour s'entraîner,"+\
                " 2 pour tester, 3 pour quitter.\n"
```

Le signe antibarre à la fin de la première ligne oblige Python à raccorder la ligne suivante à la première.

Lignes longues dans l'interpréteur

Dans les chapitres deux et trois seulement, nous utilisons le mode interactif de l'interpréteur Python. Les nouvelles lignes commencent soient par les trois signes d'invite >>>, soit par trois points . . . . Lorsqu'une ligne ne commence pas par l'un de ces deux symboles d'invite, c'est qu'il faut continuer à saisir à la suite de la ligne précédente. Voici un exemple :

```
>>> mon_long_message = 'Cette phrase va se montrer
    trop longue. La raccourcir ne devrait pas prendre
    trop de temps.'
```

Il n'y a pas dans cet exemple de symbole d'invite au début de la deuxième et de la troisième ligne. Cela signifie qu'il faut tout saisir d'affilée avant de frapper la touche Entrée pour valider. Il ne faut valider qu'après avoir saisi l'apostrophe fermante de la troisième ligne. Il ne faut pas frapper Entrée à la fin des première et deuxième lignes.

Longueur des affichages

Parfois, ce qui est affiché à l'écran diffère un peu de ce qui est imprimé dans ce livre. Dans les derniers projets plus avancés, tu verras parfois une ligne de redémarrage comme la suivante. Sur mon écran, les deux lignes de texte suivantes ne sont affichées que sur une seule :

```
>>> ============================== RESTART ================
```

Espaces et indentation

Nous verrons dans le Projet 4 comment automatiser l'indentation des lignes (pour ajouter des espaces au début). Dans les Projets 2 et 3, je te demanderais d'utiliser la barre d'espace pour insérer des séries de quatre espaces dans les lignes qu'il faut indenter. Lorsqu'il faut indenter de deux niveaux, il faut insérer huit espaces, et ainsi de suite. Toutes les lignes indentées doivent posséder le nombre d'espaces approprié.

Syntaxe et code source

Lorsque je dois expliquer comment fonctionne un élément du langage, je me sers souvent d'un modèle qui s'écrit à peu près comme ceci :

```
help([nom_objet])
```

Dans cet exemple :

- Le mot réservé est `help`.
- Ce mot doit être suivi par une paire de parenthèses.
- Lorsque j'utilise des crochets, ce qu'ils contiennent est facultatif.
- Ce qui est écrit en *italique* doit être remplacé par une valeur ou un nom réel.
- Ce qui n'est pas en italique doit être saisi tel quel.

D'après ce modèle, nous pouvons écrire `help(print)` pour obtenir de l'aide sur la fonction `print()`. On peut même écrire `help()`, donc sans rien entre les parenthèses, puisque le contenu des parenthèses était entre crochets, et donc facultatif. Les noms des variables sont imprimés `ainsi`.

Adresses Web

Les adresses Web sont écrites dans une police particulière. Ceux qui utilisent la version électronique de ce livre peuvent cliquer ces adresses pour accéder directement à la page correspondante, comme par exemple `www.pourlesnuls.fr`.

Commandes des menus

Parfois, il faut choisir une commande dans un menu (et ce n'est pas un menu de restaurant). Je demanderai par exemple de choisir File/New (Fichier/Nouveau).

Touches clavier

Les noms de touches du clavier sont indiqués sous une forme spécifique : par exemple Ctrl+A, désigne la combinaison de la touche Ctrl et de la touche A. Sous Mac OS, il y a également une touche Cmd.

Conventions dans les listings

Les listings montrent la totalité ou un extrait du code source du projet en cours de création. Voici les conventions spéciales dans les listings.

- Les mots clés de Python sont écrits `ainsi`.
- Les mots choisi par le programmeur pour ses noms de fonctions et de variables sont écrits `comme ceci`.
- Les textes des chaînes de caractères sont écrits "`Je suis un texte`".
- Les messages qui s'affichent dans l'interpréteur sont `présentés comme ceci`.

L'éditeur IDLE affiche le code source avec des couleurs différentes, mais je ne suis pas exactement son modèle, surtout parce que les identifiants (les noms des fonctions et des variables) ne sont pas du tout mis en valeur dans IDLE, alors que c'est le plus important.

Prérequis

J'ai fait tout mon possible pour ne pas réclamer trop de connaissances initiales. Tu dois être capable d'allumer l'ordinateur et d'utiliser le menu Démarrer sous Windows, l'Explorateur de fichiers et le menu sous Linux ou le Finder et la barre d'applications sous Mac OS. Pour l'installation de Python, il faudra disposer des droits d'administrateur sur la machine concernée.

Pour apprendre, il faut de l'enthousiasme et un tout petit peu de patience au départ. Je suis sûr que tu possèdes tout cela.

Les icônes utilisées dans la marge

Cette icône te rappelle une information utile ou bien signale des conseils ou des raccourcis utiles.

Cette icône signale une information ou un commentaire technique qui concerne les langages en général. Elle peut aussi te prévenir de faire attention ou de ne pas faire quelque chose.

Les fichiers d'exemples

Le livre présente plusieurs projets complets. Les textes source correspondant aux versions complètes et aux versions intermédiaires sont rassemblés dans un fichier archive que je t'invite à télécharger sur le site de l'éditeur en précisant le nom de ce livre dans le champ de recherche :

https://www.pourlesnuls.fr/telechargement

Et maintenant ?

Fort logiquement, je te propose de commencer par le Projet 1 qui correspond au Chapitre 1. J'y explique les grandes lignes de ce langage de programmation et nous verrons comment installer l'interpréteur Python. Avant de passer au Projet 2, nous saurons utiliser la combinaison clavier Ctrl+C ou Ctrl+Z pour arrêter un programme devenu fou (on ne sait jamais !).

À partir de là, tu seras prêt à écrire ton premier vrai programme Python. Tu n'es pas obligé de suivre l'ordre des chapitres, mais c'est tout de même conseillé. Chaque projet est autonome, mais je me sers de certains concepts présentés dans les chapitres antérieurs pour ne pas me répéter.

C'est parti !

Semaine 1

Partons dans Python !

AU MENU DE CETTE SEMAINE

- Projet 1 : **Pour bien démarrer**
- Projet 2 : **Salut les Terriens !**

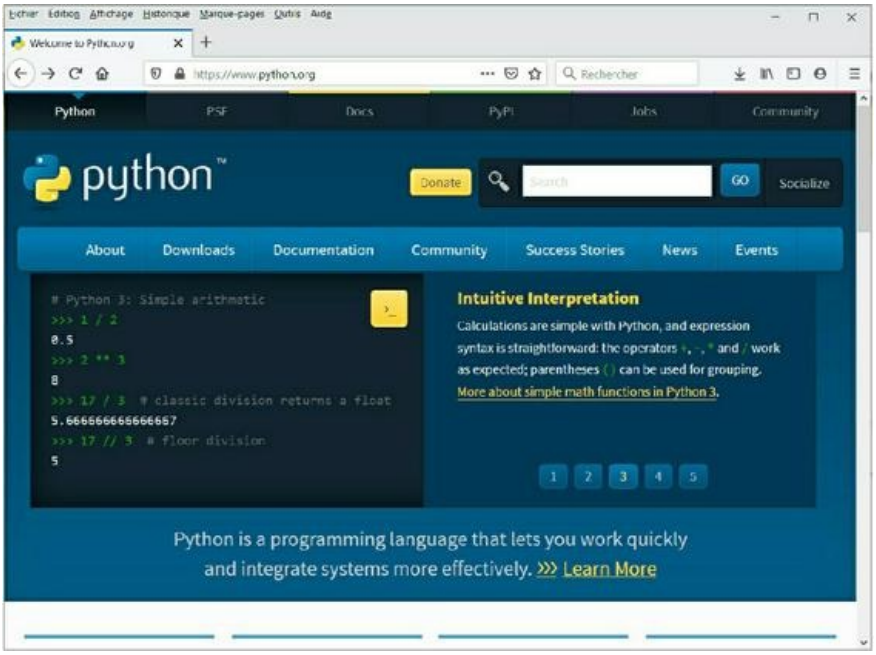
Projet 1

Pour bien démarrer

Dans ce premier chapitre, nous allons découvrir ce qu'est le langage Python et dans quels domaines il est utilisé. Il existe deux générations de Python, mais nous ne nous intéresserons qu'à la seconde, plus exactement la version Python 2.7. Je vais expliquer pourquoi ce choix. À la fin du chapitre, tu auras installé le langage Python 2 et tu sauras démarrer et arrêter l'interpréteur Python.

Dans ce chapitre, nous n'allons pas programmer. Mais ne t'inquiète pas, la vraie programmation commence dans le chapitre suivant, le Projet 2.

Si Python est déjà installé sur ta machine, et si tu sais le démarrer et l'arrêter, tu peux passer directement au chapitre suivant.



Python, c'est magique

Le langage de programmation Python a été conçu dans les années 1990 par le Hollandais Guido Van Rossum. Un langage de programmation, ou langage informatique, sert à créer des programmes, des applications et des logiciels.

Passons en revue quelques-uns des nombreux points forts du langage Python.

- Le code source Python est facile à lire et à comprendre. Personnellement, je trouve même le code source Python sublime (c'est mon avis personnel). Le langage est tellement beau qu'on ne se rend pas compte à quel point il simplifie les choses les plus complexes. Autrement dit, Python est particulièrement facile à apprendre, ce qui en fait un langage souvent sélectionné pour faire découvrir la programmation aux enfants.
- Python est très productif. Avec Python, automatiser la solution d'un problème de traitement est plus simple qu'avec bien d'autres langages. D'ailleurs, Python est considéré comme un langage de conception rapide RAD (*Rapid Application Development*).
- Python peut être dangereux. Qui dit puissance, dit responsabilités (comme pour Spiderman). Les pouvoirs que Python te donne, tu vas t'en servir pour faire le bien. (Si tu comptes t'en servir pour faire des bêtises, arrête immédiatement la lecture de ce livre !)

- Python est un langage interprété (langage de script). Le code source que tu écris est interprété ligne après ligne. Il n'y a pas besoin de lancer un traitement pour générer le code machine à partir du code source (ce qui s'appelle la compilation). De nombreuses erreurs sont ainsi détectées beaucoup plus tôt qu'avec un langage compilé. Le cycle de production Python est beaucoup plus rapide, ce qui rend la programmation plus interactive et plus amusante !
- Python n'est pas sectaire. Python fonctionne sur tous les systèmes d'exploitation répandus (notamment Windows, Mac OS et Linux), mais aussi sur les gros serveurs, et sur les nano-ordinateurs tels que le Raspberry Pi (un ordinateur sur une carte à trente euros). Tu peux même utiliser un programme Python sur les tablettes et smartphones Android aussi bien que sur un iPhone et un iPad. Je me suis d'ailleurs essayé à l'écriture de code source avec ma tablette Android pour les premiers projets de ce livre.
- Python simplifie la création des variables. Pour l'instant, cela ne te dit pas grand-chose, mais sache que les variables correspondent aux cases mémoire dans lesquelles tu stockes tes données. Python devine le type de la donnée en fonction de ce que tu lui fournis. Si tu stockes des chiffres, il devine que c'est une donnée numérique ; s'il voit des lettres, il comprend qu'il s'agit d'une chaîne de caractères, qui est d'un autre type. Cela correspond au mécanisme de *typage dynamique des données*. Nous y reviendrons, bien sûr.
- Python est accompagné de nombreux modules d'extension. Il existe pour Python des milliers de librairies complémentaires qui ont été conçues pour des besoins particuliers. Une librairie (certains appellent cela une bibliothèque) est un groupe de fonctions prédéfinies, et directement utilisables. Cela permet de gagner énormément de temps lors de l'écriture de tes programmes. C'est un peu comme avec les logiciels de création musicale par collage. Tu récupères des échantillons de tes morceaux favoris et tu les redistribues à ta manière. Il existe par exemple une librairie pour automatiser des actions dans le jeu MineCraft.
- Python est un logiciel libre et gratuit. La licence d'utilisation de Python respecte la liberté de chacun. Tu peux télécharger puis utiliser Python sans rien devoir à personne. Tous les programmes que tu vas écrire restent à toi ; tu peux les partager si tu le désires. Le code source du langage Python lui-même est librement disponible. (Cela dit, l'interpréteur Python est écrit dans un autre langage que Python, je dois l'avouer.)

PYTHON N'A RIEN D'UN SERPENT !

Le nom du langage de programmation Python n'a rien à voir avec le reptile, mais avec le groupe d'humoristes anglais des années 1970 Monty Python. On continue à rire dans le monde entier des blagues dont leurs films et émissions télévisées fourmillent. Tu as peut-être entendu parler ou vu quelques-uns de leurs films : *Sacré Graal !*, *la Vie de Brian* ou *le Sens de la vie*.

Qui utilise Python ?

Le langage Python est utilisé pratiquement dans tous les domaines.

- Dans la conquête spatiale. Le robot Robonaut de la station spatiale internationale ISS utilise Python pour son système de commandement central. Le langage doit également servir pour gérer la collecte d'échantillons du sol de Mars lors d'une mission européenne en 2020.
- En physique des particules. Python sert analyser les données recueillies lors des expériences de collisions d'atomes dans le grand accélérateur du CERN à Genève.
- En astronomie. Le radiotélescope MeerKat (le plus grand de l'hémisphère sud) est contrôlé en langage Python.
- Dans le cinéma. L'entreprise responsable des effets spéciaux de *Star Wars*, Industrial Light And Magic, a automatisé la production de ses films. Le logiciel de génération d'images 3D de Side Effects Software a adopté Python pour l'interface de programmation et pour piloter le moteur graphique.
- Dans les jeux vidéo. La société Activision utilise Python pour produire ses jeux, les tester, les analyser et même pour détecter les joueurs qui trichent en échangeant des bonus.

- Dans l'industrie musicale. Le serveur de musique en flux (streaming) Spotify recourt à Python pour diffuser ses contenus.
- Dans l'industrie vidéo. Le fournisseur de vidéo Netflix se sert de Python pour garantir que la lecture des films ne soit jamais interrompue. Python est également beaucoup utilisé par YouTube.
- Dans les recherches sur le Web. Au départ, Google utilisait énormément Python.
- En médecine. La société Nodality gère ses informations de recherche contre le cancer avec Python.
- Dans les systèmes d'exploitation. Les systèmes comme Linux et Mac OS X réalisent certaines de leurs tâches d'administration en langage Python, souvent sous forme de scripts.
- Dans la domotique à la maison. On peut tout à fait automatiser la plupart des fonctions d'une maison en ajoutant des capteurs et des moteurs. On peut ainsi faire fermer et ouvrir les volets ou allumer les lumières en rentrant chez soi.

Je pourrais continuer ainsi longtemps. Retiens simplement que dès que tu t'intéresses à un domaine, tu peux utiliser Python pour trouver une solution.

Les projets de ce livre

Voici un aperçu des projets que tu vas réaliser dans ce livre :

- un outil pour crypter du texte et créer des phrases secrètes ;
- un programme d'entraînement pour mémoriser les tables de multiplication ;
- une application de gestion de contacts, sorte de carnet d'adresses.

À la fin du livre, tu sauras écrire en langage Python, et tu pourras poursuivre dans différentes voies :

- Avec lalibrairie d'interface graphique *TkInter*, tu pourras écrire des applications fonctionnant dans l'environnement graphique (avec des fenêtres) auquel tu es habitué. Dans ce livre, nous ne travaillons qu'en mode texte, car cela suffit pour apprendre les grands principes.
- Python est utilisé par de nombreuses applications. En écrivant des scripts Python, tu vas pouvoir augmenter les capacités de nombreux programmes : *Blender* (modélisation 3D), *GIMP* (retouche photo), *LibreOffice* (suite bureautique) et bien d'autres. Je me souviens par exemple avoir eu besoin de retoucher toute une série de modèles 3D Blender. Manuellement, cela m'aurait pris des journées entières. J'ai donc décidé d'écrire un script Python qui a automatisé le travail.
- Tu pourras concevoir des jeux vidéo graphiques avec *TkInter* ou une autre librairie comme *Pygame* ou *Kivy*. Je n'aborde pas les jeux en mode graphique dans ce livre.
- La librairie *mathplotlib* permet de dessiner des graphiques complexes pour des applications en mathématiques et en physique.
- Avec la librairie *openCV*, tu peux explorer la vision par ordinateur. Les passionnés de robotique s'en servent pour permettre à leurs créatures se détecter ce qui se trouve dans leur champ visuel, et donc éviter les obstacles.

Quelle que soit ton idée, il y a de grandes chances que quelqu'un d'autre ait déjà écrit du code dont tu pourrais te resservir, ou que quelqu'un puisse t'aider à avancer plus vite.

L'approche pédagogique du livre

J'ai choisi ce titre pour impressionner tes parents. S'ils ne lisent pas cette section, confirme-leur que le livre possède une approche pé-da-go-gique (cela signifie qu'il a été conçu pour l'apprentissage).

Le but du livre est de te guider pour apprendre à mettre en pratique les grands principes du langage Python. Ce livre est donc d'abord destiné aux jeunes qui veulent découvrir ce langage.

Doucement, les bases !

Je vais être direct : je ne parlerai pas de tout. Il y a tellement d'options dans Python que si je voulais tout décrire, tu t'endormirais avant la fin du prochain chapitre. Et si tu t'endors, tu ne peux pas continuer à lire et à mettre en pratique.

J'ai essayé de présenter suffisamment de choses pour que tu deviennes autonome en programmation Python, sans aller jusqu'à te demander trop d'efforts. Tu approfondiras tout seul en te renseignant dans l'aide, la documentation et sur le Web.

POUR LES PLUS CURIEUX

Si au cours de la lecture, tu te retrouves assailli par un pressant besoin d'en savoir plus sur un point particulier de Python, commence par l'aide de Python et la documentation en ligne. Je reparle de ses possibilités en annexe. Python est également doté d'un manuel de référence, mais il est en anglais. Le livre n'est pas une référence, mais un guide d'apprentissage concret.

Nous démarrons tout en douceur avec les grands principes. Si tu trouves que cela ne va pas assez vite, tu peux passer au chapitre suivant. Les exemples sont presque toujours indépendants les uns des autres. À la fin du livre, tu auras créé plusieurs petits projets. Tu peux donc ne pas suivre l'ordre normal du livre. Cela dit, il y a quand même une certaine logique dans l'ordre naturel. Il n'est pas inutile d'accepter de suivre cette logique.

Dans les premiers projets, j'utilise le moins possible de jargon technique. Dans la suite, j'en adopte un peu, car c'est beaucoup plus pratique. Je te guide également de moins en moins au fur et à mesure que nous progressons. Il faudra bien sûr rester concentré.

Les projets que je propose sont réalistes, mais sans jamais risquer de t'ennuyer. Lorsque tu décideras de créer tes propres programmes, il te faudra posséder des méthodes et des techniques. Je présente ces éléments indispensables à la réalisation des projets, en te guidant pas à pas. Ne saute aucune étape dans ce cas.

Pour certains projets, tu peux passer directement à la fin du chapitre pour étudier le programme complet puis l'essayer en le récupérant dans le fichier des exemples (disponible sur le site de l'éditeur). Cela dit, pour apprendre à programmer en Python, il vaut mieux commencer à la première page du projet et ne pas être trop impatient. Il faut surtout faire l'effort de saisir le code source, car tu vas mieux mémoriser avec des gestes qu'avec la seule pensée.

POURQUOI J'AI CHOISI PYTHON 2

Le langage Python se distingue de la plupart de ses collègues par le fait que deux générations, la 2 et la 3, évoluent en parallèle. Le basculement vers la version 3 prend beaucoup de temps, notamment parce que les programmes écrits en Python 3 ne sont pas directement compatibles avec ceux écrits en Python 2. Les programmes que tu vas écrire en Python 2 devront être en général retouchés pour qu'ils fonctionnent en Python 3 (et inversement).

Il m'a été difficile de décider d'adopter Python 2 ou Python 3. J'ai finalement choisi Python 2 (dans sa plus récente version, Python 2.7) parce que les librairies de fonctions complémentaires sont très nombreuses pour cette génération. C'est par exemple le cas du projet MineCraft Pi, qui n'existe que pour Python 2 (pour le moment). Dès que l'on a besoin d'une librairie complémentaire, il y a plus de chances qu'elle existe en Python 2. D'ailleurs, la plupart des modules Python 3 sont également proposés dans une version pour Python 2, alors que l'inverse n'est pas vrai. Il faudra encore quelques années pour que tout bascule vers la nouvelle génération.

Ce qui distingue la génération 3 de la génération 2 se situe surtout au niveau des fonctions sophistiquées. C'est une autre raison pour laquelle il n'y a pas d'intérêt à adopter Python 3 dans un livre pour débutants qui n'aurait pas abordé ces fonctions sophistiquées.

Enfin, c'est Python 2 qui est installé par défaut sur les ordinateurs sous Mac OS. Tous les lecteurs qui ont un Mac pourront donc directement faire les exercices sans devoir télécharger et installer quoi que ce soit. Sous Linux, c'est différent, il faudra vérifier que c'est bien la version 2 qui est installée.

Je suis pragmatique

J'espère que tu pourras utiliser les projets d'exemples comme points de départ pour réaliser tes propres projets, par exemple pour t'aider dans tes devoirs ou pour gérer tes notes. Je commence en douceur, mais je rêve grand. Viens rêver avec moi en découvrant les grands principes de ce langage !

Installons Python sous Mac OS

Normalement, il n'y a pas besoin d'installer Python sous Mac OS. Pour vérifier cela, nous allons essayer de démarrer l'interpréteur Python :

1. **Au clavier, utilise la combinaison clavier** Commande+Espace **pour ouvrir le moteur de recherche Spotlight.**
2. **Saisis le mot** `terminal`**.**

   Dans le Finder, tu peux aussi utiliser la commande Aller/Utilitaires et activer Terminal.app.

   Tu dois voir s'ouvrir une fenêtre de terminal.
3. **Dans cette fenêtre (clique-la pour l'activer si nécessaire), saisis la commande** `python` **et valide par la touche** Entrée **(**Retour**).**

   Normalement, l'interpréteur Python fourni avec Mac OS démarre.

   Si ce n'est pas le cas, va sur le site de Python en t'inspirant de ce qui est écrit dans les instructions pour la version Windows et installe l'application.

Installons Python sous Windows

Quelques fabricants d'ordinateurs PC installent Python dans le système, car ils utilisent des scripts pour la configuration. Nous allons considérer que Python n'est pas installé sous Windows. Il faut donc d'abord aller sur le site officiel, télécharger un fichier puis en lancer l'installation. Si tu sais récupérer un fichier depuis un site, tu sauras installer Python.

La fondation Python (qui est responsable du langage) met à disposition des fichiers d'installation pour tous les systèmes.

Voici comment installer Python sous Windows :

1. **Ouvres ton navigateur Internet, et rends-toi à dans la page suivante :**

   `www.python.org/downloads`

2. **Clique le gros bouton** Downloads **et fait** défiler le contenu de la fenêtre vers le bas**.**

   Tu vois apparaître des options, comme dans la Figure 1.1.

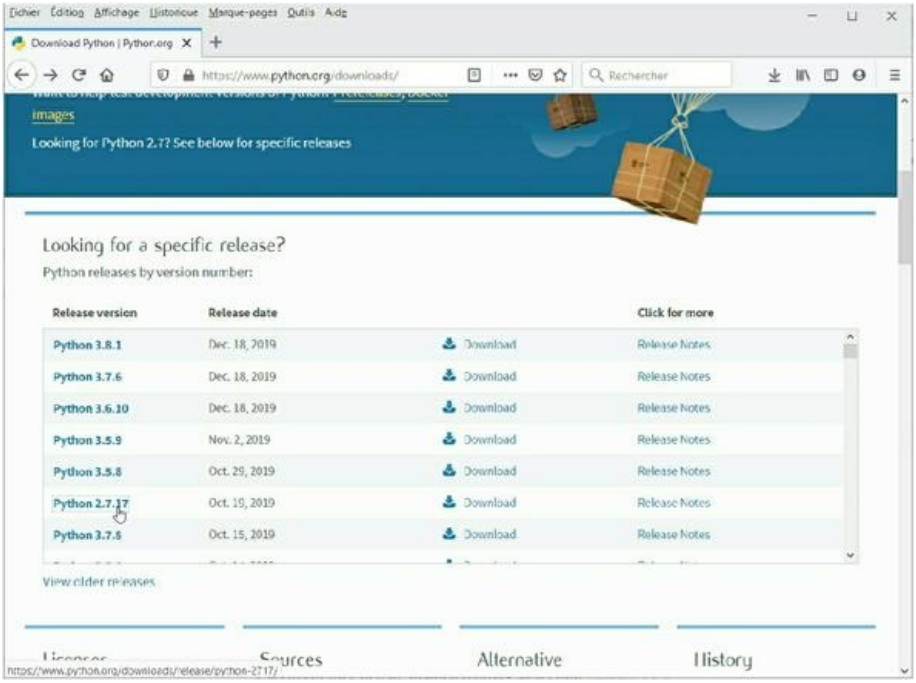


Figure 1.1 : Téléchargement de la version 2.7 de Python.

3. **Fais défiler la liste jusqu'à trouver la section** Python 2.7.x, comme sur la Figure 1.2**.**

   Le x correspond à la version la plus récente de la génération 2 (ici, c'est donc la version 2.7.17). Normalement, Python est toujours dans une sous-version de la 2.7.

4. **Clique sur ce lien. Tu accèdes à une nouvelle page, clique le lien portant le nom** Windows x86 MSI Installer**.**

   Même si la plupart des PC récents ont un processeur 64 bits, je te conseille de choisir la version 32 bits (pas celle nommée *x86-64)*, car cela te garantit l'utilisation d'un maximum de librairies complémentaires (Figure 1.3).

5. **Si le programme te demande de choisir entre** Exécuter **et** Enregistrer le fichier**, choisis** Exécuter**. Sinon, il suffit d'accepter d'enregistrer le fichier, puis de choisir de l'ouvrir, autrement dit de l'exécuter (Figure1.4).**

   Ce choix initie l'installation de Python 2.

6. **Si tu vois apparaître un message de demande d'autorisation au début de l'installation (ou plusieurs fois au cours de celle-ci), choisis** Exécuter **ou** Accepter**.**

7. **Laisse toutes les options d'installation telles qu'elles sont proposées.**

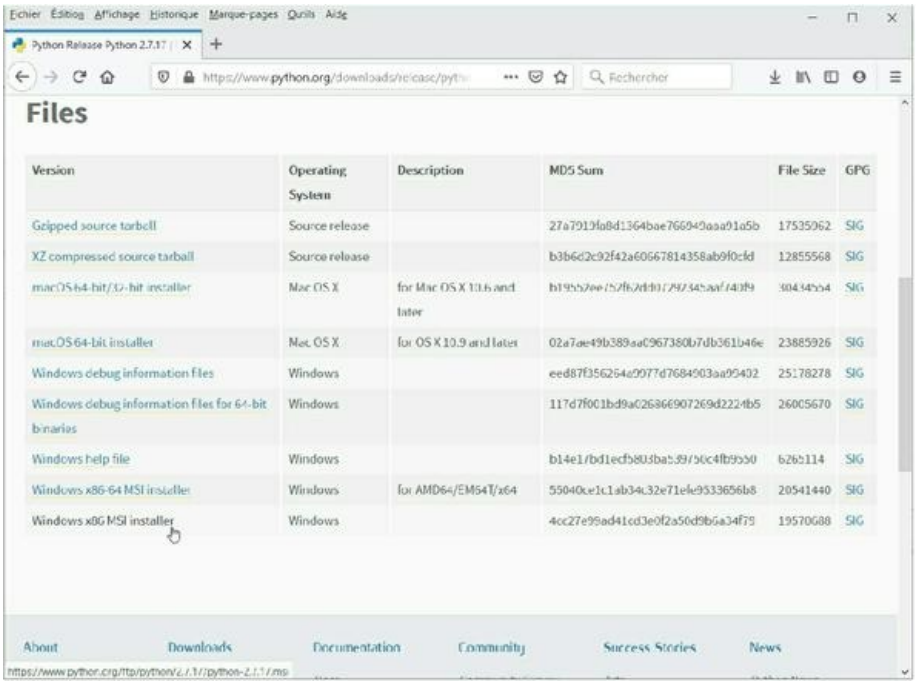


Figure 1.2 : L'installateur de Python x86.

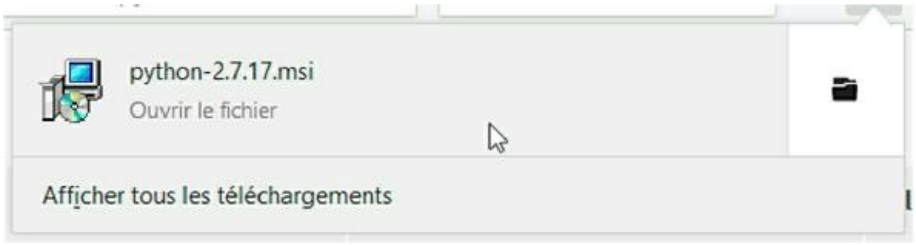


Figure 1.3 : Lancer l'installateur de Python x86 après l'avoir enregistré.

Installons Python sous Linux

Toutes les distributions Linux sont dotées dès le départ de l'interpréteur Python. Il s'agit donc de vérifier que la version installée est bien de la génération 2 et non 3. Python 2.7 est installé par défaut dans les versions récentes d'openSUSE, d'Ubuntu et de Red Hat Fedora.

Si tu constates que c'est Python 3 qui est installé, je t'invite à regarder la documentation de ta distribution ou à utiliser le gestionnaire de paquets pour installer Python 2.7 et son éditeur IDLE.

Une fois l'installation faite, tu trouveras les icônes de Python dans le dossier Développement du menu principal.

Épinglons Python (Windows)

Une fois Python installé, un dossier d'application a été créé sous le nom *Python 2.7*. Il doit contenir les quatre entrées suivantes :

- IDLE (Python GUI)
- Python (command line)
- Python Manuals
- Uninstall Python

Je te conseille d'épingler les deux icônes indiquées en gras au menu Démarrer (Windows 7 et 10) ou à l'écran d'accueil (Windows 8.1) pour pouvoir plus facilement accéder aux programmes.

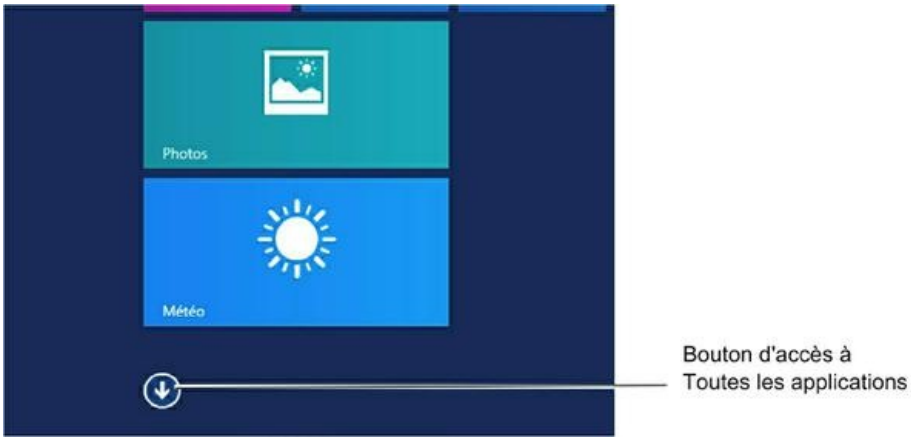


Figure 1.4 : Entrées du dossier Python sous Windows.

Windows 8

Sous Windows 8, accède à l'écrand'accueil puis tout en bas à gauche au panneau montrant toutes les applications grâce à la flèche vers le bas (Figure 1.5).

Saisis alors le début du mot `python`. Tu obtiens le groupe d'icônes installées par Python (Figure 1.6).

Clique-droit tour à tour dans l'icône IDLE (Python GUI) et Python (command line) en choisissant Épingler à l'écran d'accueil ou Épingler à la barre des tâches.

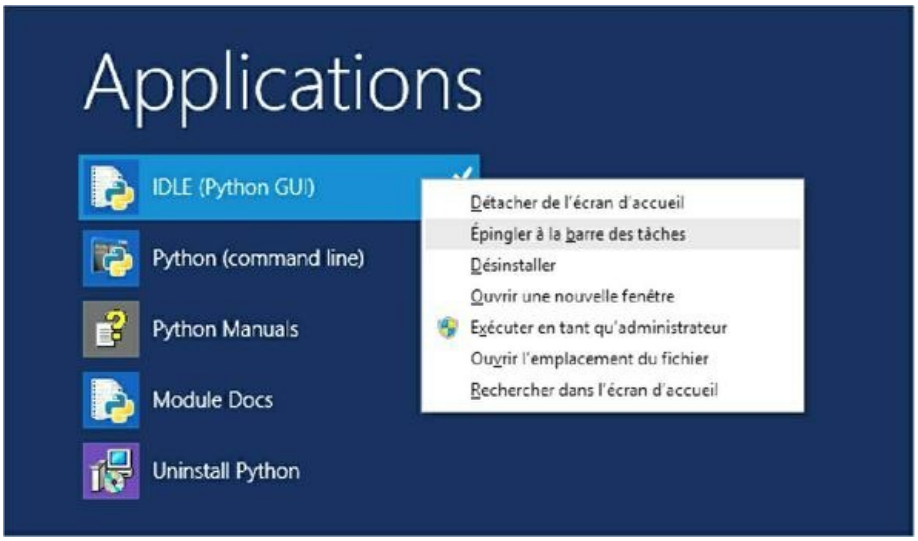


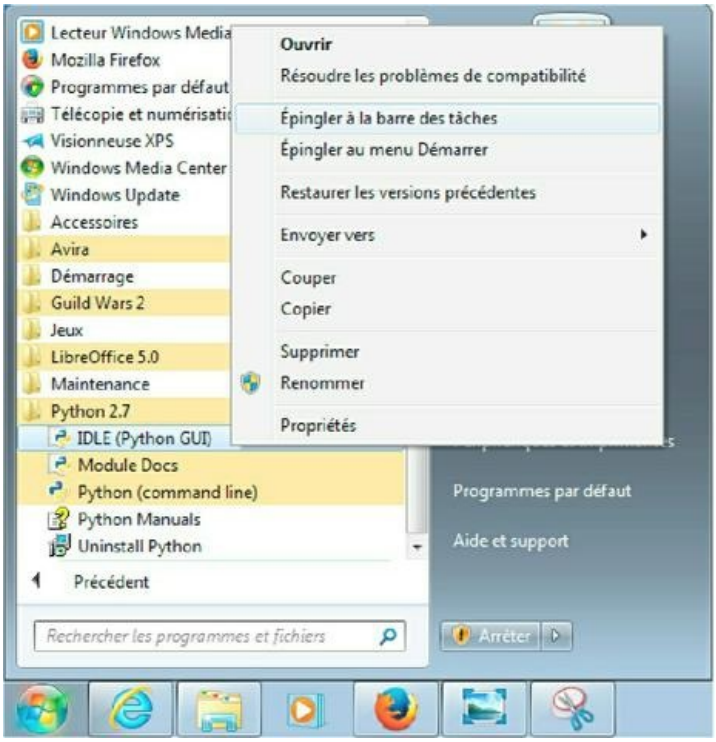
Figure 1.5 : Les cinq entrées du dossier Python sous Windows.

Windows 7 et 10

Sous Windows 7 et Windows 10, la procédure est quasiment identique.

1. **Ouvre le menu** Démarrer**. Cherche le dossier Python 2.7 et clique pour ouvrir ses détails. Pour voir toutes les applications dans le menu, tu peux avoir besoin de cliquer** Tous les programmes **ou** Toutes les applications **dans le menu** Démarrer**.**
2. **Par clic-droit, ouvre le menu local et épingle au menu Démarrer ou à la barre des tâches chaque raccourci dont nous avons besoin :** IDLE (Python GUI) **et** Python (command line) **(Figures 1.6 et 1.7)**

   Les deux entrées doivent dorénavant se trouver dans le menu Démarrer de Windows 7 et 10 ou dans l'écran d'accueil de Windows 8.1 (ou dans la barre des tâches si tu en a décidé ainsi).



[Figure 1.6](#) : Accès aux icônes de Python sous Windows 7.

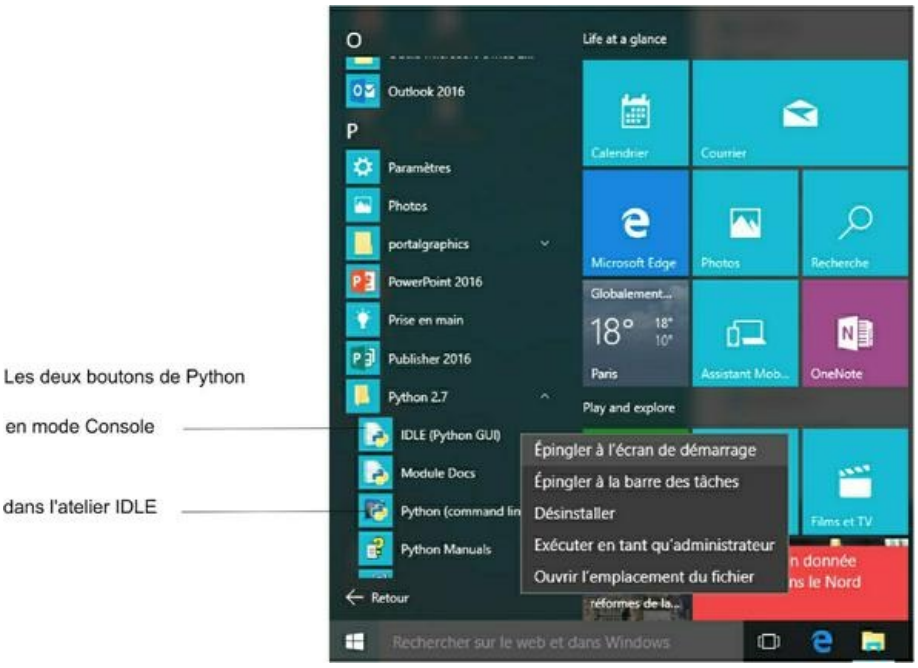


Figure 1.7 : Accès aux icônes de Python sous Windows 10.

PYTHON SUR UNE TABLETTE ?

Est-ce que tu aimerais programmer avec Python ou exécuter des programmes Python sur ta tablette ? Pour écrire un programme fonctionnant sur tablette, je te conseille de te renseigner sur la librairie de fonctions nommée Kivy. J'ai par ailleurs installé SLA4 (*Scripting Layer for Android*) avec l'interpréteur Python, et j'ai ainsi pu travailler sur les premiers chapitres du livre avec ma tablette. Tu peux aller voir dans la boutique des applications de ta tablette à la recherche des interpréteurs Python disponibles.

Les modes d'affichage graphiques varient d'une tablette à l'autre. C'est pourquoi Python doit être enrichi de certaines librairies pour que les programmes fonctionnent en mode graphique. Dans ce livre, les projets ne fonctionnent qu'en mode texte, et fonctionneront donc correctement sur une tablette. Lorsque tu voudras créer un programme en mode graphique, il faudra trouver les librairies que j'ai mentionnées et modifier les programmes en conséquence. Mais ce n'est plus un travail de débutant.

Sur tablette, je conseille vraiment d'acquérir un clavier physique. Les claviers logiciels qui apparaissent à l'écran ne permettent pas de travailler aisément. On ne trouve pas facilement les signes de ponctuation dont Python a grand besoin (et dont on a également besoin pour la correspondance habituelle en français).

Pour démarrer l'interpréteur Python

Pour mettre en pratique les projets de ce livre, il faut apprendre à démarrer Python. Vérifions que nous arrivons à ouvrir une fenêtre de terminal pour la version en ligne de commande de Python. Il suffit de cliquer l'icône Python (command line) que je t'ai invité à épingler au menu Démarrer (ou à l'écran d'accueil).

Si tu n'as pas encore épinglé le raccourci, fais-le maintenant. Au pire, tu peux également chercher le mot `Python` dans la barre de recherche puis sélectionner Python (command line) dans la section Programmes des résultats.

La fenêtre de terminal qui s'ouvre doit avoir l'aspect de celle de la Figure 1.8.

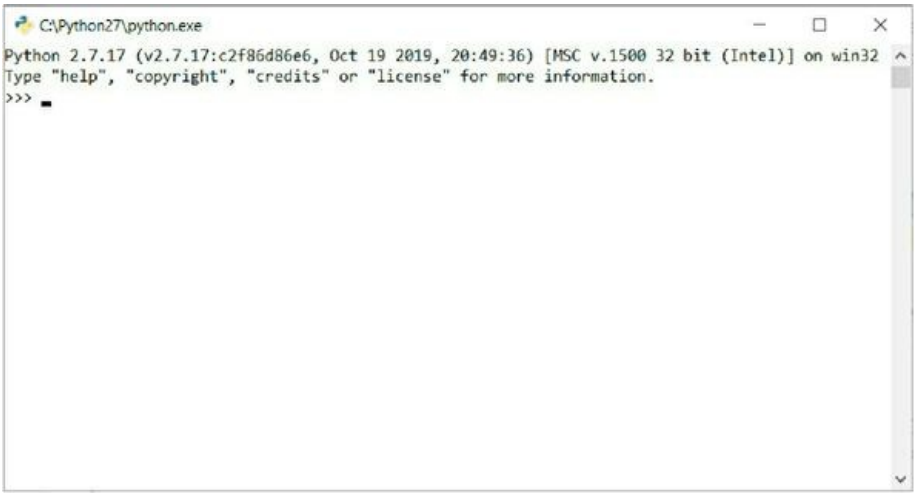


Figure 1.8 : La fenêtre de l'interpréteur Python.

Pour quitter l'interpréteur Python

Il faut bien sûr savoir démarrer Python pour pouvoir avancer dans ce livre, mais il faut également savoir quitter le programme. Si tu sais déjà le faire, tu peux même passer directement au Projet 2 dès maintenant.

La version en ligne de commande de Python peut être arrêtée par les moyens suivants :

- Tu peux taper la commande `exit()` puis valider par la touche Entrée.
- Tu peux cliquer l'icône de fermeture de l'angle supérieur droit (ou gauche) de la fenêtre de terminal.

» Sous Windows, tu peux taper la combinaison Ctrl+Z puis valider par Entrée.

» Sous Linux et Mac OS, c'est la combinaison Ctrl+D qu'il faut utiliser.

» Sous Mac OS, vérifie que tu utilises bien la touche Ctrl et pas la touche Commande. De plus, quand je parle de la touche Entrée dans la suite du livre, tu traduiras en touche Retour sur Mac.

Tes erreurs sont tes amies

Quand on écrit des programmes informatiques, on rencontre deux catégories d'erreurs, les erreurs de syntaxe et les erreurs de logique.

» Les erreurs de syntaxe. Elles sont la plupart du temps des bourdes de ponctuation. Ce sont les erreurs les plus fréquentes au début. Elles peuvent être détectées par l'interpréteur Python, qui peut donc t'aider à les corriger. Pour les éviter, il faut bien relire ce que l'on vient d'écrire et le comparer au modèle ou à ton brouillon.

Dès que tu as un souci pour faire exécuter un des projets de ce livre, la première chose à faire est de vérifier que ce que tu as saisi est exactement identique à ce que je montre dans le livre.

» Les erreurs de logique. Elles se produisent lorsque ce que tu écris n'est pas exactement ce que tu voulais écrire, et Python n'agit pas comme tu t'y attendais. Ce sont les erreurs les plus difficiles à détecter et à corriger. Il faut comparer le résultat réel au résultat prévu. Souvent, cela suppose de faire toute une série d'essais avec des valeurs différentes. Dans le Projet 6, je ferai volontairement une erreur de logique pour montrer comment l'identifier et la corriger.

Imagine que tu aies écrit dans un logiciel de traitement de texte le titre suivant :

```
lesenclumesvolente
```

Il y a dans ce titre trois erreurs (volontaires). Le correcteur orthographique du logiciel va détecter que « volente » s'écrit avec un « a » et pas un « e ». Le correcteur grammatical va détecter que l'adjectif « volante » doit être au pluriel puisqu'il s'accorde avec « enclumes ». En revanche, le logiciel ne dira rien sur le fait que pour l'instant, des enclumes volantes, ça n'existe pas ! Ce n'est pas logique.

Lorsque Python rencontre une erreur, il fait tout son possible pour donner un indice en affichant un message (hélas en anglais). Ce message comporte généralement le numéro de la ligne qu'il pense être coupable. Commence toujours par essayer de comprendre ce que ce message veut dire.

Certaines erreurs sont liées au fait que ce que tu crois vrai est faux. Si tu ne parviens pas à savoir ce qui cloche, réfléchis à ce que tu penses être la situation initiale. Si cela ne te sert à rien, va prendre l'air pour remettre ton cerveau à zéro. Va boire un verre d'eau, nourrir ton poisson rouge ou marcher un peu sans penser à autre chose pour aérer ton cerveau. (C'est une mauvaise idée que d'en profiter pour aller sur les réseaux sociaux.) À la fin de ta pause, reviens te plonger sur ton problème Python. Tu devrais pouvoir profiter d'un regard neuf sur le problème, ce qui te rapprochera de la solution.

Si cela ne résout rien, fait venir ton animal de compagnie à côté de toi pendant que tu programmes. Lorsque tu bloques sur un problème, explique le problème à ton animal ou à ton objet fétiche.

Décrire à voix haute une situation est une technique de débogage (correction d'un problème) éprouvée. Cela fonctionne parce que pour expliquer le problème à quelqu'un d'autre, il faut d'abord avoir réussi à le comprendre un minimum.

En utilisant des mots du français pour présenter ton problème, tu fais travailler une autre partie de ton cerveau. Et si tu es du même genre que moi, c'est-à-dire pas très extraverti, tu peux adopter la technique du journal de programmation. Le principe reste le même : tu notes par écrit la description du problème et les causes apparentes du blocage. Je te garantis que cela va t'aider à résoudre la plupart de tes soucis.

Apprends à apprendre

Lire ce livre, c'est bien, car cela te permet de gagner du temps, mais cela ne suffit pas. Il faut pra-ti-quer. Approprie-toi le livre, cherche à faire un avec, et que la force soit avec toi !

Il faut pratiquer !

Sérieusement. Si tu veux vraiment apprendre, tu dois faire. Personne n'a pu apprendre efficacement simplement en lisant. C'est donc le cas pour le langage Python, comme pour toute autre matière.

Tape toutes les lignes de code source tout en progressant dans le livre. Interdis-toi de faire des copier/coller. Adopte cette discipline au moins pour tes premiers projets. Tu verras que tu comprendras bien plus vite à quoi sert un code source simplement en le parcourant des yeux. Tu ne seras jamais intime avec le code si tu ne fais que copier/coller. Et n'hésite pas à partir en *freestyle* en ajoutant tes propres modifications. N'aie pas peur de faire des erreurs. Tu apprendras bien plus vite en t'appropriant le code.

Fais des erreurs

Toi, tu n'es pas un ordinateur. N'hésite pas à te tromper. Si tu pars en *freestyle*, n'aie pas peur de créer des erreurs dans le code. Tu peux toujours utiliser la commande d'enregistrement sous un autre nom (Save as/Sauver sous) pour faire des copies de sécurité.

Tous les programmeurs se trompent au départ, que ce soit des erreurs de syntaxe ou de logique. Ce sont des erreurs tout simplement.

Il est aussi important d'oser faire des erreurs que de pratiquer.

C'est tout à fait normal. L'écriture d'un programme est un processus qui comporte des pièges. Même les programmeurs professionnels font tous les jours des erreurs. Cela ne leur fait pas peur parce qu'ils travaillent par petites portions qu'ils peuvent gérer, et testent chaque passage avant d'aller plus loin.

Prends le temps de penser

Lorsque le programme ne fonctionne pas comme prévu ou pas du tout, fais une pause pour chercher à comprendre ce qui se passe. Si tu y réfléchis suffisamment, tu trouveras la réponse.

Apprendre consiste à combiner les erreurs, la recherche de solutions et la réflexion pour comprendre. C'est la courbe d'apprentissage. Les différents éléments se combinent, un peu comme le langage Python lui-même.

Ganbatte ne !

Il existe en japonais le mot *ganbatte*. Il suggère un mélange des idées suivantes : bonne chance, fais au mieux, sois fort, sois tenace, aie du courage, n'abandonne jamais et tu y arriveras. Quand on commence à apprendre un langage de programmation, on a de grands rêves, mais peu d'expérience. On pourrait se laisser décourager si on regarde les fantastiques logiciels que créent des équipes entières de personnes dans des projets de plusieurs mois ou années.

De nombreux programmes de nos jours fonctionnent avec une interface graphique ou manipulent des objets graphiques. Ce sont des domaines que je n'aborde pas dans ce livre.

Accroche-toi dans ces premiers pas et tes efforts seront récompensés !

Récapitulons

Pour ce premier chapitre, nous avons abordé les points suivants.

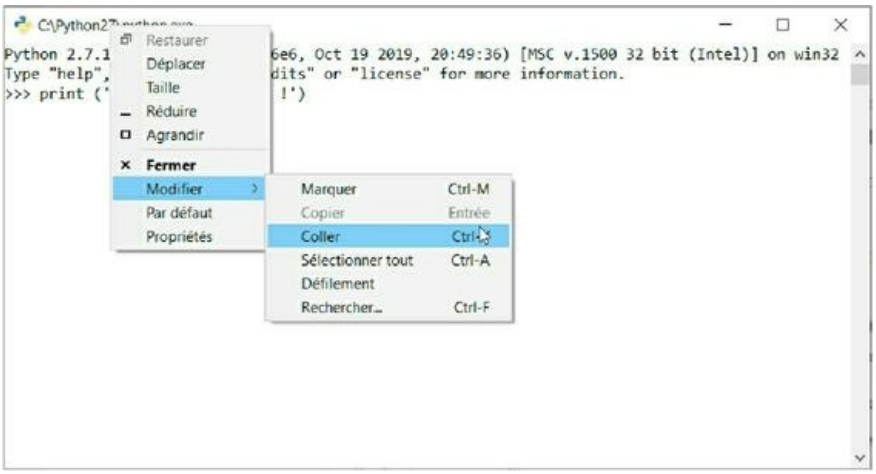
- Nous avons vu dans quels domaines est utilisé le langage Python.
- Nous avons découvert les deux générations Python 2 et Python 3 et compris pourquoi ce livre utilise Python 2. Nous avons téléchargé Python, l'avons installé et appris à le démarrer et à l'arrêter.
- Nous avons découvert l'approche tolérante aux erreurs qu'il faut adopter pour bien progresser.

Projet 2

Salut les Terriens !

Dans le Projet 1, nous avons installé Python et appris à démarrer l'interpréteur. Dans ce projet, nous plongeons vraiment dans la découverte de Python. Le but est de faire de chaque lecteur un programmeur Python en formation.

En programmation informatique, une tradition s'est établie depuis plusieurs dizaines d'années. Elle consiste à choisir comme objectif pour le tout premier programme l'affichage d'un simple message de bienvenue du style « Salut les Terriens ! » (en anglais, "Hello, world!"). En honorant cette tradition, tu pourras toi aussi dire plus tard que tu as commencé par *Hello world!*



Dans ce projet, nous allons voir comment transmettre une information depuis le programme vers l'écran de l'utilisateur, c'est-à-dire afficher un message. Tu verras ainsi comment le langage se souvient d'une information et comment tu peux choisir quelle information Python doit mémoriser avant de l'afficher. C'est ce qui permet de réutiliser cette information en d'autres endroits du même programme.

Nous allons découvrir comment Python progresse du début à la fin d'un programme, comment arrêter de force un programme qui deviendrait incontrôlable, et même comment faire faire une petite bêtise au programme (afin de voir comment l'arrêter). Nous terminerons en découvrant une des techniques fondamentales de programmation : la répétition ou *boucle*. Nous nous en servirons pour remplir l'écran de dizaines de messages de bienvenue.

Affichons un message !

Voici comment écrire notre premier programme de bienvenue :

1. **Démarre l'application** Python (command line)**. Sous Windows, tu la trouves dans le menu** Démarrer **puisque nous l'avons rendue accessible directement dans le menu au cours du Projet 1.**

   Une fois le programme lancé, tu dois voir s'afficher une fenêtre avec un fond sombre et en haut un peu de texte et les fameux trois signes `>` qui incarnent l'invite.

   Tous les extraits de code dans ce livre sont imprimés avec une `police non proportionnelle`, comme celle des machines à écrire. Lorsque tu vois cette police, c'est du code source Python.

   Pour le moment, nous allons utiliser Python dans son mode interactif grâce à l'interpréteur. C'est dans ce mode qu'apparaît la série de trois petits symboles dans la fenêtre. Python affiche ces symboles pour t'inviter à saisir

une commande.

2. **Au niveau de cette invite, tu vas saisir la ligne de code qui suit ce paragraphe. Il faut prendre soin d'utiliser le signe apostrophe. (C'est la touche 4 sur ton clavier.)**

```
print ('Salut les Terriens !')
```

3. **Valide l'exécution de ta première commande au moyen de la touche Entrée (ou Retour sur Mac).**

   Python exécute immédiatement le code que tu viens de lui fournir.

   Tu dois voir apparaître le même résultat que dans la Figure 2.1. Si tu vois la même chose, félicitations : tu as réussi à écrire ton premier programme ! Bienvenue dans le club des programmeurs Python en herbe !

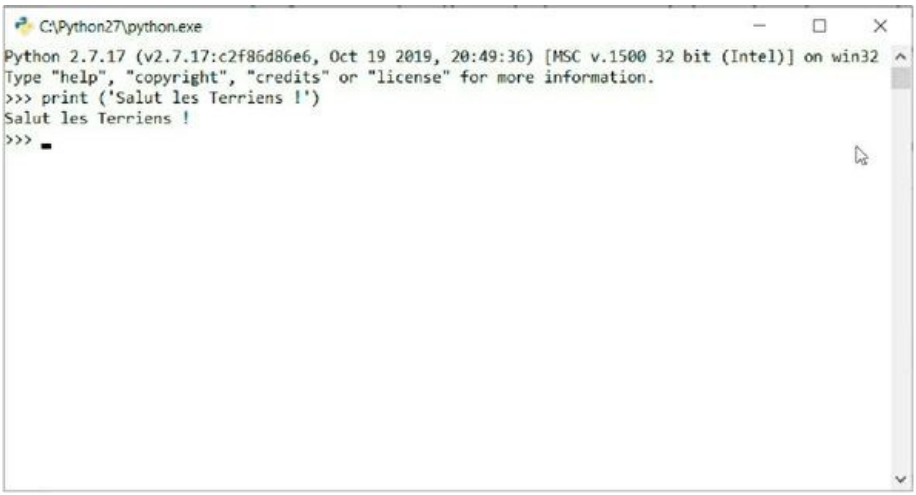


Figure 2.1 : Le programme Salut les Terriens après exécution.

Si tu ne vois pas la même chose que dans la figure, il faut vérifier ce que tu as saisi à l'étape 2 :

- Vérifie la présence et la position des parenthèses et des apostrophes.
- Vérifie que tu n'as pas oublié la parenthèse fermante, car c'est une bourde fréquente.
- Vérifie qu'il y a bien une apostrophe de part et d'autre du message à afficher.

Chaque langage de programmation possède des règles de grammaire et de ponctuation qui constituent la syntaxe du langage. Les langages humains restent compréhensibes maime lorsqu'il sont bourés de fôtes de grandmaire (la preuve : tu as réussi à lire ce que je viens d'écrire !). En programmation, la plus petite erreur provoque une réaction négative de l'interpréteur.

Cherchons et réparons nos erreurs

Comme un interprète humain, celui de Python prend chaque phrase (ici, chaque ligne de code source) pour la traduire (l'exécuter) immédiatement, dès que tu as validé avec la touche Entrée. Dans notre premier programme, nous utilisons une instruction prédéfinie du langage Python qui s'écrit `print()`. Cette instruction récupère le contenu des parenthèses et l'affiche sur la prochaine ligne vide de la fenêtre de la console ou terminal.

J'ai dit que Python était chatouilleux avec la grammaire et la ponctuation. Si tu fais une erreur de frappe, le programme ne pourra pas fonctionner.

Lorsque Python attend un caractère spécial à un certain endroit, il faut le lui fournir. Passons en revue quelques exemples d'erreurs volontaires. Essaie de les repérer avant de lire la solution.

```
>>> pritn('Salut les Terriens !')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
NameError: name 'pritn' is not defined
```

Voici un deuxième exemple :

```
>>> print('Salut les Terriens !)
  File "<stdin>", line 1
    print('Salut les Terriens !)
SyntaxError: EOL while scanning string literal
```

En voici un dernier :

```
>>> print 'Salut les Terriens !')
  File "<stdin>", line 1
    print 'Salut les Terriens !')
                         ^
SyntaxError: invalid syntax
```

Python fait de son mieux pour fournir un indice sur la cause probable de l'erreur (avec les titres NameError et SyntaxError).

Voici la solution de ces trois erreurs :

- Les noms des commandes et instructions doivent être écrits correctement (le premier exemple indique `pritn` au lieu de `print`).
- Chaque apostrophe ou guillemet ouvrant doit faire référence à une apostrophe ou à un guillemet fermant (l'apostrophe fermante manque dans le deuxième exemple).
- La parenthèse ouvrante est indispensable (oubliée dans l'exemple 3).

Les valeurs littérales

Dans notre programme de bienvenue, le texte qui est envoyé par l'instruction `print` vers les circuits d'affichage de l'écran est ce que l'on appelle un *littéral*. Un littéral est quelque chose qui est délimité par des apostrophes ou des guillemets. Ne confonds pas les apostrophes (') avec les guillemets (").

Un littéral est une information qui ne peut jamais changer au cours de l'exécution. Le nom complet est d'ailleurs *constante littérale*. Un littéral est une donnée, qui ne peut donc pas avoir le rôle d'une action dans le programme. Voici un exemple :

```
>>> 'Salut les Terriens !'
'Salut les Terriens !'
```

PRINT OU PRINT() ?

L'instruction `print` est la seule que nous ayons utilisée pour l'instant. Elle n'a normalement pas besoin d'un jeu de parenthèses. La syntaxe de `print` n'est pas la même dans Python 2 et dans Python 3. Dans Python 2, `print` est le mot réservé d'une instruction. Avant l'apparition de Python 3, le programme traditionnel de bienvenue offrait l'aspect suivant :

```
print "Salut les Terriens !"
```

Il n'y a pas de parenthèse dans cette ligne. Les créateurs et les responsables de la fondation Python ont décidé de changer la syntaxe dans la version 3 de Python en ajoutant un jeu de parenthèses obligatoire. En effet, alors que c'était une instruction, `print()` est devenu une fonction. Nous expliquerons dans un autre projet ce qu'est une fonction.

Tous les exemples de ce livre pourraient fonctionner en utilisant l'ancienne version de `print`, c'est-à-dire sans les parenthèses. (Ceux qui ne me croient pas peuvent faire des essais.) Étant donné qu'il va falloir s'habituer aux parenthèses pour utiliser Python 3 un jour ou l'autre, je les utilise dès maintenant puisqu'elles sont acceptées, même si elles ne sont pas obligatoires dans Python 2. Un conseil : fais comme moi.

Cette ligne de code source se résume à un littéral entre deux apostrophes. Le résultat est presque le même que dans le premier programme de bienvenue. Souviens-toi que, dans le programme, l'affichage ne comportait pas les apostrophes. Dans le petit essai ci-dessus, les apostrophes sont affichées.

L'interpréteur Python ne cherche absolument pas à analyser le contenu d'une valeur littérale. Tu peux donc faire des fautes d'orthographe, ajouter des gros mots et donner libre cours à ton imagination à l'intérieur des deux apostrophes. Cela ne provoquera pas de message d'erreur de la part de Python.

Pourtant, ces apostrophes sont indispensables. Si tu en oublies une, Python va chercher à analyser ce texte comme une commande ou une instruction.

Dans l'exemple volontairement faux qui suit, Python ne sait que faire des trois mots salut, les et Terriens :

```
>>> Salut les Terriens !
    File "<stdin>", line 1
        Salut les Terriens !
                           ^
SyntaxError: invalid syntax
```

Le message que nous utilisons depuis le début du projet est un texte de bienvenue, constitué de plusieurs mots. Une telle valeur porte le nom de *chaîne littérale*, car il s'agit de l'enchaînement de plusieurs caractères alphabétiques. Un littéral chaîne se distingue d'un littéral numérique. Nous y reviendrons.

Pour définir une chaîne littérale, il suffit, comme tu sais maintenant, d'ajouter une apostrophe à gauche et une autre à droite :

```
Salut les Terriens !
'Salut les Terriens !'
```

Évidemment, le fait que le langage se serve du signe apostrophe pour savoir où commence et/ou se termine le littéral peut poser un problème lorsque le contenu (une phrase, par exemple) contient ce signe, comme dans l'exemple suivant :

```
>>> 'Je t'aime'
  File "<stdin>", line 1
    'Je t'aime'
        ^
SyntaxError: invalid syntax
```

Dans l'exemple, quand Python trouve le deuxième signe apostrophe, il croit que cela marque la fin de la chaîne littérale, alors que ce n'est pas le cas.

La solution est très simple, car on peut utiliser comme délimiteur de début et de fin de chaîne soit des apostrophes, soit des guillemets. (D'ailleurs, les guillemets sont bien plus souvent utilisés comme délimiteurs dans les autres langages.) Voici deux exemples corrects :

```
>>> "Je t'aime"
"Je t'aime"

>>> '"Je suis trop content" dit-il alors.'
'"Je suis trop content" dit-il alors.'
```

Le langage Python est très généreux au niveau des délimiteurs de littéraux. On peut tout à fait utiliser trois apostrophes ou trois guillemets consécutifs pour marquer le début et la fin d'un littéral chaîne. Vois plutôt :

```
>>> '''Une phrase entre triples apostrophes.'''
'Une phrase entre triples apostrophes.'
>>> """Une autre entre triples guillemets."""
'Une autre entre triples guillemets.'
```

Tu constates que la réponse affichée en écho n'est délimitée que par des apostrophes simples, ce qui prouve que Python a compris que ce sont des littéraux.

Tu peux faire des essais avec les différents délimiteurs. Essaye d'afficher au moins un message délimité par des apostrophes, un autre avec des guillemets, et un autre contenant une apostrophe.

Stockons du texte dans une variable

Tu maîtrises désormais les chaînes littérales. Mais une fois que tu as écrit un littéral dans ton code source, tu ne peux rien en faire d'autre que l'utiliser immédiatement. Le langage Python ne conserve pas les littéraux, car il a besoin de bien gérer l'espace mémoire. Un processus automatique, qui s'appelle le *ramasse-miettes* (*garbage collector*), se charge de faire le ménage régulièrement dans la mémoire vive de l'ordinateur. Il supprime notamment l'espace encore occupé par les chaînes littérales qui ne servent plus.

Mais comment faire pour empêcher Python d'oublier une chaîne littérale ? La solution est simple : il faut lui donner un nom. Python ne cherchera plus à le supprimer. Tout ce qui existe dans l'univers porte un nom ou alors n'existe pas. Il en va de même dans les programmes informatiques. C'est un peu comme si tu colles une étiquette sur le petit message.

Voici comment donner un nom à un littéral :

1. **Commence par chercher un nom qui obéit aux règles de nommage que j'indique un peu plus bas. Il faut que ce nom soit significatif (ou au moins amusant).**
2. **Saisis le nom puis une espace (avec la touche Espace) et un signe égal (=).**
3. **Saisis le littéral qu'il faut stocker sous ce nom à droite du signe égal (sans oublier ses délimiteurs).**

Voici deux exemples de littéraux stockés avec un nom :

```
>>> mon_message = 'Salut les Terriens !'
>>> nom_variable_trop_long = 'Ce nom est Ok, mais un peu trop
long.'
```

Voici les règles à respecter pour que Python accepte le nom que tu veux créer :

- Le nom doit commencer par une lettre et jamais par un chiffre. Il est possible de commencer par le signe de soulignement (_), mais cela a une signification spéciale dans le langage. Il faut donc éviter de s'en servir au début.
- Un nom ne peut jamais contenir d'espace.
- Aucun nom ne doit être identique à un mot réservé du langage Python. Je donne la liste des mots réservés un peu plus loin dans ce même projet.

Les règles suivantes ne sont pas obligatoires, mais elles sont très conseillées :

- Le nom doit donner une idée de ce à quoi il sert. Par exemple, `texte_ question` est un nom bien choisi pour un littéral qui contient le texte d'une question affichée à l'écran. C'est une mauvaise idée que d'appeler ce littéral `toto47`.
- On peut profiter du caractère de soulignement _ pour aérer les différents morceaux d'un nom, puisqu'il est interdit d'utiliser l'espace.
- En dehors du premier caractère, le nom peut comporter des chiffres.
- Tu peux combiner des lettres minuscules et des lettres majuscules dans un nom, et certains en profitent pour rendre les noms plus compréhensibles. Par exemple, `NombreEssais` est plus facile à lire que `nombreessais`.
- En revanche, les noms entièrement en majuscules sont réservés aux noms des constantes (on verra cela plus loin).

Pour utiliser la valeur littérale qui est associée au nom, il suffit dorénavant de citer ce nom. Un nom de constante peut donc aussi se comprendre comme une sorte d'abréviation ou de synonyme. Il est plus rapide d'écrire `msg2` que `'Ceci est un trop long message'`.

Presque toutes les valeurs qui sont utilisées dans les programmes sont associées à un nom. Jusqu'à maintenant, nous n'avons utilisé que des valeurs fixes (littérales). Nous en verrons d'autres par la suite.

Quand on donne un nom à une valeur, cela s'appelle une *affectation*. Dans l'exemple suivant, nous affectons la valeur qui est une chaîne littérale au nom indiqué à gauche du signe égal :

```
mon_message = "Salut les Terriens !"
```

Voici comment nous pouvons maintenant reformuler notre programme de bienvenue :

```
>>> mon_message = "Salut les Terriens !"
>>> print(mon_message)
Salut les Terriens !
```

La première ligne associe (affecte) la chaîne littérale au nom `mon_message`. Souviens-toi que le nom est toujours à gauche du signe égal et la valeur littérale à droite. La deuxième ligne affiche le contenu de `mon_message`.

Une fois que tu as défini un nom, tu peux en général changer la valeur qui est associée à ce nom en répétant le processus d'affectation. Inversement, tu peux choisir un autre nom. Repartons du précédent exemple qui créait deux noms pour deux chaînes :

```
>>> mon_message = 'Salut les Terriens !'
>>> nom_variable_trop_long = 'Ce nom est OK, mais un peu trop
long.'
```

Nous pouvons copier la valeur associée au deuxième nom dans le premier nom puis demander l'affichage du contenu pour voir ce qui s'est passé :

```
>>> mon_message = nom_variable_trop_long
>>> print(mon_message)
Ce nom est Ok, mais un peu trop long.
>>> mon_message = 'Un autre message'
>>> print(mon_message)
Un autre message

>>> print(nom_variable_trop_long)
Ce nom est Ok, mais un peu trop long.
```

Dans mon dernier exemple, la valeur associée à `mon_message` a changé. Les noms des variables sont en fait des adresses de stockage dans la mémoire. Dans le lieu de stockage, on place une valeur, mais cette valeur peut changer au cours du temps. Voilà pourquoi on appelle cela une *variable*. Le nom est le nom de la variable. Ce qui est stocké à l'adresse en mémoire qui est symbolisée par le nom est la valeur de la variable. Le fait de stocker une valeur se nomme l'affectation.

Nous remarquons également que la valeur du deuxième nom `nom_variable_ trop_long` n'a pas changé. Nous n'avons fait que lire la valeur, puisque ce nom se trouve du côté droit du signe égal. L'écriture se fait vers la variable dont le nom est mentionné du côté gauche du signe égal (qui est un opérateur).

Des littéraux numériques

L'autre grande catégorie de valeurs à côté des valeurs littérales (des bribes de texte) est la catégorie des valeurs numériques. Les valeurs numériques peuvent être manipulées grâce à des opérateurs mathématiques (addition, soustraction, *etc.*) Voici quelques essais :

```
>>> a = 1
```

```
>>> b = 2
>>> print(a)
1
>>> print(b)
2
>>> print(a + b)
3
>>> print(b - a)
1
>>> print(a < b)
True
```

Cette fois-ci, nous utilisons des noms très courts, comme lorsque l'on écrit des expressions mathématiques. Le symbole `<` est celui d'un opérateur qui permet de comparer la valeur située à sa gauche et celle située à sa droite. La mention `a < b` demande à Python de tester s'il est vrai ou pas que la valeur `a` est inférieure à la valeur `b`. Dans l'exemple, la valeur `a` est égale à 1 et la valeur b est égale à 2. Donc, `a` est inférieure à 10. En résultat, Python considère que ce test est vrai. C'est pourquoi il affiche la mention `True` (qui signifie « vrai » en anglais).

Il est possible de citer le même nom des deux côtés du signe égal. Cela permet de modifier la valeur stockée à l'adresse correspondant à ce nom. Lorsque le nom est cité à droite, on procède à la lecture de la valeur actuelle, et lorsque le nom est cité à gauche du signe, on écrit la valeur lue au même endroit. Voici par exemple comment augmenter de 1 une variable numérique nommée `var`. Je commence par lui donner la valeur 1, ce que je vérifie en l'affichant.

```
>>> var = 1
>>> print(var)
1
>>> var = var + 1
>>> print(var)
2
```

Le langage commence par lire la valeur actuelle de `var` puis il l'augmente de 1. Il stocke ensuite le résultat dans la même variable.

Une boucle infernale avec while

Dans la vie courante, il n'est pas bien vu d'interrompre quelqu'un, mais ici, c'est indispensable. Amusons-nous à créer un programme qui n'arrive plus à s'arrêter. On appelle cela une boucle infinie ou boucle infernale. Nous allons créer ce programme défectueux pour apprendre à provoquer la fin de l'exécution.

Il est indispensable de savoir provoquer la fin d'un programme lorsque celui-ci ne veut plus répondre. L'astuce consiste à utiliser le raccourci clavier composé des deux touches Ctrl et C (inutile d'utiliser la touche Maj pour faire C majuscule). Vérifie d'abord que la fenêtre de terminal de Python est la fenêtre active (clique dans la fenêtre si nécessaire). Il ne faudrait pas fermer un autre programme sans le vouloir !

Nous allons d'abord découvrir comment écrire un programme contenant une boucle infinie, c'est-à-dire qu'une fois démarré, il ne s'arrêtera jamais de lui-même. Ce sera l'occasion de découvrir un mot réservé important de Python : `while`.

1. **Tu es devant l'invite de l'interpréteur (les `>>>`). Saisis les deux mots suivants, sans oublier le signe deux-points (`:`) final puis valide par la touche Entrée :**

   ```
   while True :
   ```

2. **Appuie quatre fois sur la barre d'espace.**
3. **Saisis le mot réservé `pass` puis valide avec la touche Entrée.**
4. **Appuie une seconde fois sur la touche Entrée pour indiquer que tu ne veux plus rien saisir d'autre.**

Voici ce qui doit apparaître à l'écran :

```
>>> while True:
...     pass
...
```

La seconde frappe de la touche Entrée a provoqué le démarrage du programme, puis l'interpréteur a cessé de répondre. Tu ne peux plus saisir de commande. Le programme semble ne rien faire et ne plus accepter de données. L'invite habituelle ne réapparaît pas, ce qui prouve que l'interpréteur est occupé à exécuter quelque chose. D'ailleurs, tu peux peut-être percevoir une certaine activité de l'ordinateur (le ventilateur qui accélère, par exemple).

Pour sortir sur-le-champ de cette boucle infernale, utilise la combinaison clavier Ctrl+C (Windows) ou Ctrl+Z (Linux et Mac OS). Voici ce qui s'affiche :

```
^CTraceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyboardInterrupt
```

QUE SIGNIFIENT CES POINTS ?

Lorsque tu utilises Python en ligne de commande, donc en mode texte dans une fenêtre de terminal, tu es en mode interactif. En effet, Python répond immédiatement à la commande saisie dès que tu valides par la touche Entrée. Dans ce mode interactif, les symboles d'invite sont soit les trois points (`...`), soit les trois signes Supérieur à (`>>>`). Nous connaissons déjà la série de trois signes Supérieur à. Lorsque ce sont des points qui sont affichés, cela signifie que Python attend que tu saisisses autre chose. Il continuera à accepter d'autres éléments saisis jusqu'à ce que tu appuie deux fois de suite sur la touche Entrée et non une seule fois.

Il n'y a pas de limite théorique au nombre de lignes de suite que tu peux saisir dans ce mode, sauf des raisons de logique que nous verrons plus loin. Il est essentiel que toutes les lignes de suite soient décalées de la marge gauche du même nombre d'espaces, pour que tout continue à faire partie du même bloc de code. Les conventions d'écriture de Python consistent à faire des décalages de quatre espaces et de multiples de quatre. Même si c'est tentant, il ne faut pas utiliser la touche de tabulation.

Le message qui s'affiche lors de l'interruption est un message de trace de pile (*Traceback*). Il donne des informations sur ce qui était en cours (le dernier appel, *most recent call*) juste avant que le programme rencontre l'erreur ou le blocage. Dans un programme plus complexe, ce message permet d'avoir un indice sur la cause probable du problème. Ici, on te dit que cela concerne le fichier (*file*) *stdin* qui est en fait le clavier.

Intéressons-nous au mot `while` du précédent exemple. Il s'agit d'un mot réservé par le langage. Il sert à écrire une boucle de répétition conditionnelle pour exécuter de façon répétée toutes les lignes qui en dépendent tant que la condition de la boucle est satisfaite.

La condition est une formule ou une expression que l'instruction va tester. Le résultat du test (une évaluation) est soit vrai, soit faux, ce qui correspond aux deux mentions `True` et `False`. Si la condition est vraie, on exécute une fois la série de lignes décalées qui suivent la ligne du `while`. C'est un « tour de boucle ». Si la formule n'est pas vraie, on sort de la boucle. Dans cet exemple, la condition `True` est toujours vraie (elle l'est par définition).

Maintenant que nous savons briser une boucle infernale, passons à un exemple concret. Pour le saisir, il faut tenir compte des règles suivantes :

1. **Tu dois appuyer quatre fois sur la barre d'espace au début des deux lignes qui forment le bloc conditionnel appartenant à la ligne `while`.**
2. **Tu dois appuyer une fois sur la touche Entrée après avoir saisi chaque ligne.**
3. **Tu dois frapper une seconde fois la touche Entrée après avoir validé la ligne `print(a)`.**

```
>>> a = 2
>>> while a < 10:
...     a = a+1
...     print(a)
...
3
4
5
6
7
8
9
10
```

Que se passe-t-il lorsque ce programme s'exécute ? Il affiche les nombres de 3 à 10. En effet, Python exécute la ligne `print(a)` huit fois (tu peux vérifier). Pourtant, nous n'avons mentionné qu'une fois l'instruction d'affichage `print()`.

Analysons ce petit programme de la même manière que l'interpréteur Python. Après la première ligne dans laquelle nous affectons la valeur de la variable `a`, nous arrivons à la ligne contenant `while`. Ce mot réservé demande à Python de répéter toutes les lignes qui la suivent et qui sont décalées de quatre espaces par rapport à elle. La répétition se produit tant que (*while*) la condition mentionnée à droite du mot `while` est vraie.

Dans l'exemple, la condition s'écrit `a < 10`. Le bloc de code qui sera répété est constitué des deux lignes suivantes. À la différence d'autres langages, il suffit que les lignes qui constituent ce bloc à répéter soient décalées du même nombre d'espaces par rapport à la marge gauche. Dans d'autres langages, on délimite le bloc avec des accolades. Remarque bien le signe deux-points à la fin de la ligne de tête de `while`. Il signifie qu'un bloc de code doit suivre. Par convention, les lignes des blocs sont décalées par multiples de quatre espaces.

En règle générale, l'interpréteur Python lit le code source dans le sens normal de lecture, de la première à la dernière ligne, et exécute tour à tour chacune des instructions qu'il rencontre. Parfois, certaines sections du code source sont répétées, et c'est le cas dans cette boucle `while`. Nous verrons plus loin qu'il est aussi possible de sauter directement vers une autre ligne.

Dans les détails d'une boucle

Avant le début du premier tour, la variable `a` reçoit la valeur 2. Le test est vrai puisque 2 est plus petit que 10. On réalise donc un premier tour de bloc.

La première ligne du bloc demande d'ajouter 1 à la valeur de `a` (`a = a+1`). C'est pourquoi la première valeur qui est affichée par l'instruction `print(a)` est 3. La fin du bloc correspond à cette ligne d'affichage fondée sur `print()`. Puisque c'est la fin du bloc et que nous sommes dans une boucle, Python revient à la tête de boucle pour tester si la condition est toujours vraie. Au début du deuxième tour, `a` vaut 3, ce qui est toujours inférieur à 10. La condition est donc satisfaite.

On repart pour un tour de boucle. La valeur de la variable passe à 4, et cette valeur est affichée. Nous continuons ainsi jusqu'à ce que la variable contienne la valeur 10. Après avoir affiché la valeur, nous remontons faire le test, mais la condition n'est plus satisfaite, car 10 n'est pas inférieur à 10. Python sort alors de la boucle, et passe à l'exécution de la ligne qui suit la dernière faisant partie du bloc. Dans cet exemple, il n'y en a pas, donc le programme s'arrête.

On appelle ce mécanisme une *boucle* ou un bloc de répétition, parce que Python répète les lignes du corps de boucle (le bloc) tant que la condition reste vraie.

pass, pass, passera...

Revenons brièvement au premier exemple de boucle `while`, la boucle infernale. Revoici à quoi elle ressemble :

```
>>> while True:
...     pass
```

Cet exemple n'avait aucun effet sur quoi que ce soit, mais c'était tout de même notre premier programme de plus d'une ligne. La boucle ne faisait absolument rien, à part bloquer le programme, mais pour qu'elle soit acceptée par l'interpréteur Python, nous avons utilisé le mot réservé `pass`.

Ce mot réservé `pass` demande tout simplement à l'interpréteur de passer la ligne. C'est un peu étonnant. Prévoir une instruction qui n'a aucun effet, à quoi cela pourrait-il servir ? C'est obligatoire parce que Python, lorsqu'il détecte un mot réservé de boucle comme `while`, s'attend à rencontrer ensuite une expression (la condition) puis un signe deux-points (`:`) qui marque le début du bloc de code à répéter. Ce bloc doit exister et comporter au moins une ligne avec une instruction. Si Python ne trouve pas ce bloc, il refusera d'exécuter la commande `while`. Pour le tromper et le satisfaire, j'ai utilisé cette botte secrète, c'est-à-dire le mot `pass`.

Voici comment l'interpréteur se plaint si on n'utilise pas cette astuce pour le satisfaire (appuie deux fois sur la touche Entrée après le signe deux-points `:`) :

```
>>> while True:
...
File "<stdin>", line 2
  ^
IndentationError: expected an indented block
```

Tu n'utiliseras que très rarement ce mot réservé `pass`, qui n'a servi ici qu'à permettre de lancer l'exécution du programme volontairement défectueux. Certains programmeurs s'en servent comme marqueur de position pour revenir plus tard au même endroit pendant la rédaction du code source.

Les mots réservés de Python

Python compte 31 mots dont la signification est figée, c'est-à-dire réservée :

'and'	'as'	'assert'	'break'	'class'	'continue'	'def'
'del'	'elif'	'else'	'except'	'exec'	'finally'	'for'
'from'	'global'	'if'	'import'	'in'	'is'	'lambda'
'not'	'or'	'pass'	'print'	'raise'	'return'	'try'
'while'	'with'	'yield'				

Dans cette liste, neuf mots réservés sont assez complexes, et je n'en parlerai pas du tout dans ce livre :

'assert', 'del', 'except', 'exec', 'finally', 'global', 'raise', 'try' et 'yield'.

Pour l'instant, sache que tu ne peux utiliser aucun de ces mots réservés comme nom pour tes variables. En revanche, un nom de variable peut comporter ce mot dans son nom. Tu ne peux pas, par exemple, utiliser le mot `return` comme nom, mais tu peux choisir comme nom de variable `mavaleur_return`.

Voici ce qui se produit lorsque l'on essaye d'utiliser le mot réservé `return` pour un nom de variable :

```
>>> return = 4
File "<stdin>", line 1
return = 4
^
SyntaxError: invalid syntax
```

Dorénavant, dès que tu verras un message d'erreur (`SyntaxError`) qui mentionne une syntaxe invalide (*invalid syntax*) dans une ligne d'affectation d'une valeur à une variable, tu sauras pourquoi.

Des bienvenues plein l'écran

Une fois que tu sais écrire une boucle, tu peux par exemple afficher des dizaines de messages de bienvenue. Saisis le programme de trois lignes suivant. N'oublie pas la virgule bizarre tout à la fin de la dernière ligne. Comme la fenêtre de terminal va se remplir sans fin, prépare-toi à arrêter l'exécution infinie avec la combinaison magique Ctrl+C !

```
>>> mon_message = 'Salut les Terriens !'
>>> while True:
...     print(mon_message),
...
```

Mais à quoi sert cette petite virgule de rien du tout qu'on croirait abandonnée là par mégarde ? Il faut savoir que l'instruction `print()` provoque normalement le passage à la ligne suivante après l'affichage de son message. En ajoutant cette virgule à la fin de la ligne d'affichage, tu empêches `print()` de forcer le saut de ligne. Le message suivant est donc collé à la fin du précédent.

Lorsque j'étais gamin, je me souviens que j'aimais me donner du courage. Étudie le programme suivant et le résultat affiché à l'écran dans la Figure 2.2 :

```
>>> msg_bravo = 'Brendan est formidable !!!'
>>> while True:
...     print(msg_bravo)
...
```

Tu peux bien sûr indiquer ton propre prénom à la place de `Brendan`. Toi aussi tu es formidable !

Figure 2.2 : Nous sommes tous formidables.

Se répéter, mais pas trop

Couvrir l'écran de messages d'encouragement n'est pas désagréable, mais il est fatigant de devoir utiliser la combinaison clavier Ctrl+C pour arrêter le programme. De plus, si tu le laisses fonctionner ainsi trop longtemps, tu risques de provoquer une erreur au niveau du système de ton ordinateur.

Dans la pratique, on décide dès le départ combien de tours de boucle il faut faire. C'est le rôle de l'instruction `while`, mais il est également possible d'utiliser la fonction `range()` qui sert à générer une série de valeurs numériques (entières uniquement) :

```
>>> range(3)
[0, 1, 2]
```

`range()` permet d'obtenir toutes les valeurs à partir de zéro et jusqu'à la dernière valeur de l'intervalle inférieure à celle indiquée comme butée (3, ici), sans y inclure cette valeur. Dans l'exemple, la valeur 3 n'est pas générée, mais trois valeurs le sont.

On commence à zéro !

Tu as peut-être froncé les sourcils devant le comportement apparemment étrange de la fonction `range()` : elle commence à compter à partir de zéro et non à partir de un.

En fait, il s'agit de la manière habituelle de compter des ordinateurs, la première valeur est zéro et non un. Il y a une bonne raison : cela rend les tests beaucoup plus simples, car il est plus facile de voir qu'il n'y a plus rien (plus aucun fruit dans un panier, par exemple) que de vérifier qu'il n'en reste exactement plus qu'un et non deux ou trois.

RANGE OU XRANGE

Dans Python 2, on évitait d'utiliser la fonction `range()` parce qu'elle consomme plus de mémoire que sa variante `xrange()`. Dans Python 3, `range()` a été améliorée et il n'y a plus besoin de `xrange()`. Prends donc l'habitude d'utiliser uniquement `range()` pour être prêt à basculer vers la génération Python 3. De plus, le problème mémoire ne survient que pour de très grands intervalles, largement au-delà de 1 000.

Faisons compter Python avec range()

Tu peux demander à la fonction `range()` de compter d'un nombre jusqu'à un autre en indiquant le premier nombre et le nombre qui sert de butée dans les parenthèses, en séparant les deux valeurs par une virgule :

```
>>> range(3,10)
[3, 4, 5, 6, 7, 8, 9]
```

Python compte à partir du nombre de départ (3), et s'arrête juste avant le second.

On peut aussi faire compter de deux en deux ou de trois en trois. Par exemple, pour générer les seules valeurs paires de 3 à 10, demande à Python de commencer à trois et de s'arrêter juste avant 10 en avançant de deux à chaque pas :

```
>>> range(3, 10, 2)
[3, 5, 7, 9]
```

Si le pas de progression (la troisième valeur entre parenthèses) est négatif, tu obtiens un compte à rebours. Dans ce cas, il faut bien sûr que la première valeur soit supérieure à la deuxième :

```
>>> range(10, 0, -1)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Nous sommes les plus for

La fonction `range()` devient intéressante quand on l'utilise dans une boucle de répétition, puisqu'elle va générer une valeur pour chaque tour de boucle. Découvrons à cette occasion une autre instruction fondamentale pour répéter des actions, l'instruction `for` :

```
>>> for cptr in range(3):
...     print(cptr)
...
0
1
```

Tu remarques que la commande d'affichage `print()` est indentée (décalée de quatre espaces par rapport à la ligne précédente), afin de faire partie du bloc de répétition.

Cette boucle possède la même structure que celle que nous avions conçue avec la boucle `while`, sauf que cette fois-ci il n'y a pas de conditions à tester pour arrêter de boucler. L'instruction `for` est associée à une variable qui s'appelle `cptr`, ici (pour compteur). Cette variable prend à chaque tour une des valeurs de l'intervalle généré par `range()` (nous les affichons).

Le bloc de code, qui se résume ici à la seule commande d'affichage `print()`, est exécuté trois fois, pour les valeurs numériques 0, 1 et 2. La variable `cptr` est une variable jetable, qui ne sert que de compteur dans la boucle. Le choix de son nom est donc beaucoup moins critique, mais il faut faire attention à ne pas lui donner le même nom qu'une variable existant ailleurs dans le programme.

Il faut toujours essayer de choisir des noms bien significatifs pour les variables. (Les variables jetables sont une exception.)

La valeur de la variable change effectivement à chaque tour de boucle. Normalement, tu ne te serviras pas de cette variable ailleurs que dans la boucle. (Avec un peu plus d'expérience, il y aura des exceptions.)

En général, les programmeurs utilisent les lettres minuscules `i`, `j` et `k` pour les variables des boucles `for`. Ce genre de variable est également appelé un compteur ou un indice.

Nous pouvons maintenant utiliser la technique basée sur une variable de boucle pour revoir la boucle `while` qui était infinie afin qu'elle s'arrête au bout d'un certain temps.

```
>>> msg_salut = "Salut les Terriens !"
>>> for i in range(300):
...     print(msg_salut),
...
```

Tu devrais obtenir la même chose qu'à la Figure 2.3.

Dans cet exemple, la valeur numérique `300` est un nombre magique qui semble provenir de nulle part. Les nombres magiques sont normalement déconseillés. Il est toujours préférable d'associer chaque valeur, même celles qui sont fixes (les constantes), à un nom de variable. Le nom permet de savoir à quoi sert la valeur, ce qui évite des erreurs.

Dans l'exemple, nous pourrions améliorer la situation en ajoutant au départ la ligne suivante (c'est une déclaration de constante) :

```
NBRE_MESSAGES = 300
```

puis modifier l'appel à `range()` pour qu'il s'écrive ainsi :

```
range(NBRE_MESSAGES)
```

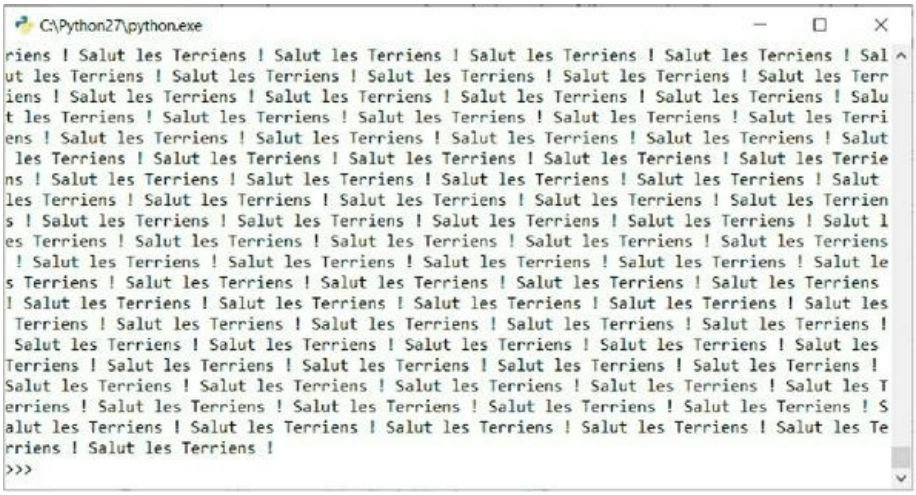


Figure 2.3 : Un message affiché 300 fois, ni plus, ni moins.

Dans un petit programme de test comme celui de cette page, cela n'a pas d'effet, mais dans les programmes réels, cette création d'un synonyme peut tout changer.

Au départ, tu peux croire que tu n'auras jamais besoin de changer ce nombre, mais c'est rarement vrai. Lorsque le nombre est « codé en dur » (indiqué littéralement), c'est-à-dire non associé à un nom qui le symbolise, s'il faut le changer, il faut chercher tous les endroits où il est utilisé. Et comme c'est une valeur numérique, il n'y a aucun moyen de savoir à quel genre de chose correspond la valeur.

Par exemple, il peut tout à fait y avoir un autre endroit dans le programme où on a besoin de spécifier 300 unités en largeur ou en hauteur. S'il faut un jour passer de 300 à 100 unités, on pourrait croire qu'il suffit de rechercher et remplacer toutes les valeurs 300 en 100. Évidemment, cela fera également changer le `300` qui sert à compter le nombre de tours de boucle, alors qu'il ne devait pas changer.

Lorsque ces valeurs fixes sont associées à des noms, par exemple `NBRE_MESSAGES = 300` et `LARGEUR = 300`, il suffit de changer en un seul endroit, là où est faite l'affectation du nombre au nom. Tous les endroits où la valeur est utilisée par l'intermédiaire du nom de la variable, utilisent la bonne nouvelle valeur. Les autres variables qui contiennent la même valeur au départ ne sont pas perturbées.

Récapitulons

Voici les sujets abordés au cours de ce projet :

- Ce qu'est une valeur littérale, de type chaîne et de type numérique.
- L'affectation d'une valeur à une variable en choisissant un nom de variable valide et en utilisant le signe égal qui est un opérateur.
- L'utilisation de quelques mots réservés du langage (`while`, `pass`, `for`, `in`) de deux fonctions (`print()` et `range()`) et des valeurs prédéfinies `True` et `False`.
- Nous avons enfin appris à provoquer la fin d'exécution d'un programme devenu fou au moyen de la combinaison clavier Ctrl+C.

Semaine 2

Jouons aux devinettes

AU MENU DE CETTE SEMAINE

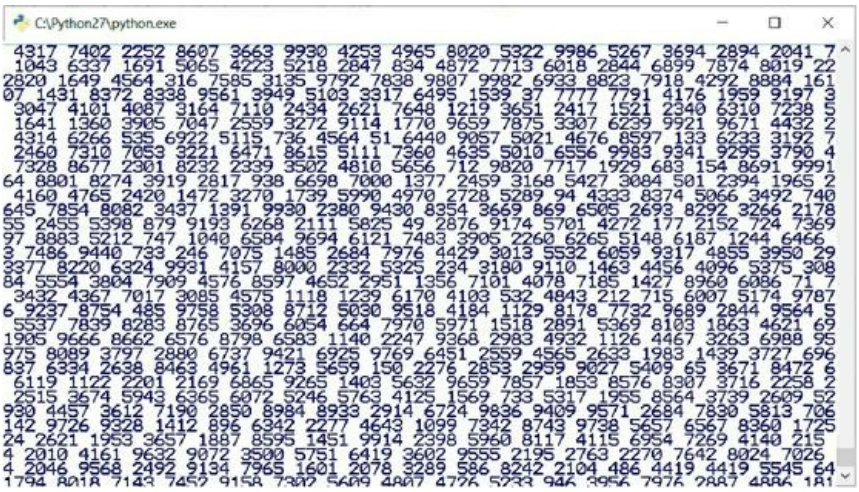
- Projet 3 : Un jeu de devinette
- Projet 4 : Découvrons l'atelier IDLE
- Projet 5 : Ta fonction : deviner()

Projet 3

Un jeu de devinette

Nous allons créer un jeu dans lequel l'ordinateur choisit un nombre entre 1 et 100, et c'est à toi de deviner lequel.

Au cours de ce projet, tu vas apprendre comment obtenir une information saisie au clavier et comment sont gérées les valeurs numériques dans Python (ce n'est pas la même chose que les chaînes de caractères). Tu vas apprendre à faire générer un nombre au hasard par le programme, en réutilisant une fonction déjà écrite par quelqu'un d'autre. Nous verrons même un peu comment corriger les erreurs de logique, c'est-à-dire faire du débogage.



Analysons d'abord

Dans ce jeu de devinette, Python choisit un nombre que le joueur devra trouver.

- Si le joueur réussit à deviner, Python affiche un message de félicitations.
- Si le joueur n'a pas trouvé le bon nombre, Python lui donne un indice en indiquant si le nombre à deviner est plus grand ou plus petit que celui déjà proposé.
- Le joueur continue à proposer des valeurs jusqu'à avoir trouvé la réponse.
- Le programme compte le nombre d'essais.

Si tu as bien pratiqué le Projet 2, tu sais comment afficher un message pour le joueur, avec la commande `print()`. Pour l'instant, tu ne sais pas comment récupérer une valeur numérique saisie au clavier, ni comment demander à Python de générer une valeur au hasard.

Ce cycle en trois étapes (1] sortie de données, 2] entrée de données et 3] traitement de ces données) est une technique fondamentale dans tous les langages informatiques.

Récupérons des données saisies

Dans la pratique, quasiment tous les programmes ont besoin de demander à l'utilisateur de saisir quelque chose. Pour l'instant, nous travaillons en mode texte (avec la ligne de commande) ; la saisie se fait au

clavier, et rien n'est sélectionné avec la souris. Pour récupérer des données, nous disposons d'une commande prédéfinie qui porte le nom `raw_input()`, c'est-à-dire « entrée brute ». Cette commande attend que l'utilisateur saisisse quelque chose puis valide avec la touche Entrée. Elle récupère alors les données et les renvoie.

Voyons en mode interprété comment fonctionne cette commande. Commence par ouvrir la fenêtre de terminal pour la version Python (Command line). Au cours du Projet 1, nous avons vu comment accéder facilement aux deux raccourcis de Python.

1. **Saisis la ligne suivante :**

```
raw_input()
```

2. **Valide avec la touche Entrée.**

   Tu arrives sur une ligne vide.

3. **Tape n'importe quoi en guise de réponse.**

4. **Valide par la touche Entrée.**

   Ce que tu as saisi lors de la troisième étape est affiché tel quel :

```
>>> raw_input()
J'ai saisi un petit texte dans le terminal.
"J'ai saisi un petit texte dans le terminal."
```

Ce qui a été saisi au clavier est réaffiché après ajout de guillemets ou d'apostrophes comme délimiteurs autour des données. Python a donc affiché en écho ce que tu as saisi, comme vu dans le Projet 2. As-tu déjà une idée de comment nous pourrions récupérer ce qui est saisi dans le programme ?

Pour éviter tout risque de confusion, mon exemple de saisie a été délimité par des guillemets (") et non par des apostrophes ('), car l'interpréteur a détecté une apostrophe dans le texte saisi (`j'ai`).

Explique ce que tu veux !

Pour que l'utilisateur sache quel genre de données il doit saisir, il faut afficher un message d'invite. Tu as remarqué que `raw_input()` était suivi d'une paire de parenthèses vide `()`. Ces parenthèses ne sont pas inutiles puisque tu peux y placer un texte qui sera affiché juste avant que la fonction se mette en attente de ce que tu vas saisir. Ce texte s'appelle une *invite*, puisqu'il invite à réagir.

Dans les exemples en mode interprété, lorsque la ligne commence par trois signes `>>>` ou `...`, cela signifie qu'il faut taper la commande qui se trouve sur la même ligne.

```
>>> raw_input("Propose un nombre :")
Propose un nombre :17
'17'
```

En fait, Python fait exactement ce que tu lui demandes de faire. Il affiche le message d'invite, mais n'ajoute pas d'espace entre la fin du message et le premier caractère que tu saisis en réponse. Pour que la saisie soit plus confortable, il suffit de prévoir une espace à la fin de ton texte d'invite.

```
>>> raw_input("Propose un nombre : ")
Propose un nombre : 17
'17'
```

Après cette correction, l'utilisateur peut lire et relire sa réponse plus aisément.

Nous avons placé ici un littéral entre parenthèses avec des guillemets. Nous pouvons aussi indiquer le nom d'une variable dans laquelle ce littéral aura été stocké au préalable :

```
>>> raw_input("Propose un nombre : ")
>>> raw_input(mon_invite)
Propose un nombre : 17
'17'
```

Mais comment récupérer ce qui a été saisi ? En fait, le principe est le même que pour toutes les autres données : il suffit de créer une variable dans laquelle nous pourrons stocker les données. Dans l'exemple suivant, nous avons choisi comme nom de variable `nbr_joueur`.

```
>>> mon_invite = 'Propose un nombre : '
>>> nbr_joueur = raw_input(mon_invite)
Propose un nombre : 17
'17'
>>>
>>> nbr_joueur
'17'
```

Nous renvoyons dans cette nouvelle variable la valeur qui est renvoyée par la fonction de demande de saisie. C'est magique, et c'est le mode habituel d'utilisation d'une fonction. Lorsque tu fais un essai, tu constates que la valeur numérique que tu saisis n'est pas immédiatement réaffichée. Elle est stockée dans la variable. Il faut demander l'affichage du contenu de cette variable pour confirmer qu'elle contient bien ce qui a été saisi.

Apprenons à comparer

Dans notre projet, nous devons pouvoir vérifier si le nombre proposé par l'utilisateur est bien celui que connaît le programme. Il s'agit d'une opération de comparaison. Nous voulons vérifier qu'il y a égalité entre deux nombres. C'est à ce niveau qu'il y a un piège. Dans Python comme dans de nombreux autres langages informatiques, il ne faut pas utiliser le signe égal auquel on est habitué en classe d'arithmétique. Tu ne peux pas écrire "`SI a = 1`" pour vérifier qu'un nombre est égal à 1. Le signe égal unique est un *opérateur d'affectation* ; il sert à copier une valeur dans une variable. En écrivant avec un seul signe égal pour comparer, tu demandes à Python de « vérifier si copier la valeur dans la variable », ce qui ne veut rien dire (n'est-ce pas ?). Python rétorque à juste titre qu'il n'a rien compris.

Pour comparer deux choses et savoir si elles sont égales, il faut utiliser dans Python deux signes égal consécutifs (`==`). Dans notre exemple, pour savoir si une variable `a` contient bien la valeur 1, il faut écrire `a == 1`. Faisons quelques essais :

```
>>> a = 1
>>> a == 1
True
>>> a = 2
>>> a == 1
False
```

Pour rappel, *true* signifie « vrai » en anglais, et *false*, « faux ».

Voici trois autres essais, pour que les choses soient bien claires :

```
>>> 1 == 1
True
>>> 1 == 2
False
>>> 1 = 2
```

```
File “<stdin>”, line 1
SyntaxError: can’t assign to literal
```

Les deux premiers essais sont bien des comparaisons. Nous demandons si 1 est bien la même chose que 1, puis si 1 est égal à 2. Dans le premier cas, c’est vrai (`True`) ; dans le second cas, c’est faux (`False`). Dans le troisième essai, nous demandons à Python de réaliser quelque chose d’absurde : copier la valeur 2 dans la valeur 1. Cela n’a aucun sens, et Python affiche en réponse qu’il y a une erreur de syntaxe, parce que la valeur à gauche du signe `=` n’est pas le nom d’une variable. D’ailleurs, le texte de l’erreur explique en anglais que Python ne peut pas affecter de valeur à un littéral (une valeur).

Dès que Python voit les deux signes égal `==`, il compare la valeur qu’il trouve à droite à celle qu’il trouve à gauche des signes. Si les valeurs sont identiques, Python remplace toute l’expression par la valeur prédéfinie `True`. Si la comparaison échoue, Python remplace toute l’expression par la valeur prédéfinie `False`.

Notre panoplie d’opérateurs

Le signe `=` et le double signe `==` sont des *opérateurs*. Ils opèrent sur des valeurs. Dans la plupart des cas, il y a une valeur à gauche de l’opérateur et une autre à droite. Les principaux opérateurs sont présentés dans le Tableau 3.1. La plupart servent surtout avec des valeurs numériques (addition, soustraction, *etc.*). Les opérateurs sont indispensables à tous les programmeurs au quotidien.

Tableau 3.1 : Principaux opérateurs arithmétiques de Python.

Opérateur	Nom	Effet	Exemples

`+`	Plus	Pour additionner deux nombres ou coller une chaîne de caractères à la fin d’une autre.	Addition
			`>>> 1 + 1`
			`2`
			Jointure de chaînes :
			`>>> ‘a’ + ‘b’`
			`‘ab’`

`-`	Moins	Pour soustraire un nombre d’un autre. Non utilisable avec des chaînes.	`>>> 1 - 1` `0`

`*`	Multiplication	Sert à multiplier deux nombres ou à faire une copie d’une chaîne.	Nombres :
			`>>> 2 * 2`
			`4`
			Copie de chaîne :
			`>>> ‘a’ * 2`
			`‘aa’`

`/`	Division	Sert à diviser un nombre par un autre. Non utilisable avec des chaînes.	Voir la section suivante

`%`	Modulo	Renvoie le reste de la division du nombre de gauche par celui de droite. Sert également à contrôler le format des chaînes de caractères (nous verrons cela dans le Projet 8).	Nombres :
			`>>> 10 % 3`
			`1`
			Chaînes :
			`>>> print (“Mme %s”%”JOLY”)`
			`Mme JOLY`

**	Puissance	L'écriture x**y demande d'élever x à la puissance y.	>>> 9 **2 81

Tableau 3.2 : Opérateurs conditionnels de Python.

Opérateur	Nom	Effet	Exemples
=	Affectation	Copie la valeur de droite dans la variable indiquée à gauche.	>>> a = 1

==	Test d'égalité	Sert à comparer si les deux côtés sont identiques. Renvoie True si c'est le cas et False dans le cas contraire.	>>> 1 == 2
			False
			>>> 'a' == 'a'
			True

! =	Test de différence	Permet de vérifier que la valeur de gauche est différent de celle de droite et renvoie True si c'est le cas.	>>> 1 ! = 2
			True
			>>> 'a' ! = 'A'
			True

>	Supérieur à	Pour vérifier si la valeur de gauche est supérieure à celle de droite.	>>> 3 > 23 False
>=	Supérieur ou égal à	Pour vérifier si la valeur de gauche est supérieure ou égale à celle de droite.	>>> 2 >= 1 True
<	Inférieur à	Pour vérifier si la valeur de gauche est inférieure à celle de droite.	>>> 3 < 23 True
<=	Inférieur ou égal à	Pour vérifier si la valeur de gauche est inférieure ou égale à celle de droite.	>>> 2 <= 1 False

and (ou &)	ET logique	Renvoie True seulement si les deux valeurs (gauche et droite) sont vraies (True) toutes les deux. Opérateur utilisé dans les conditions complexes.	>>> True & False
			False
			>>> True and (1 == 2)
			False

or (ou ¦) Alt+Maj+L	OU logique	Renvoie True dès qu'une des deux valeurs (gauche ou droite) est vraie (True). Opérateur utilisé dans les conditions complexes.	>>> True or True
			True
			>>> True or False
			True
			>>> False ¦ False
			False
			>>> True ¦ (1 == 2)
			True

La division dans Python

Dans Python, les divisions méritent une petite explication complémentaire. En effet, le langage essaie de faire de son mieux, mais avec parfois plus de mal que de bien. Si tu n'y prends pas garde, tu vas obtenir des calculs faux. Méfiance !

En informatique, il y a deux grandes catégories de valeurs numériques : les nombres *entiers*, c'est-à-dire ceux qui n'ont pas de chiffres après la virgule, et les nombres *fractionnaires*, avec des chiffres après la

virgule. On appelle ces derniers les nombres *flottants* (parce que la virgule peut flotter vers la gauche ou vers la droite).

IMPORTANT : Puisque Python est un langage inventé en anglais, on n'utilise pas une virgule (`,`), mais un point (`.`) pour séparer la partie entière de la partie décimale.

Le piège de la division entière est celui-ci : lorsque tu divises un entier par un autre entier dans Python 2 et que le résultat ne donne pas un entier, Python oublie la partie décimale et arrondit le résultat à l'entier le plus proche.

Voici un exemple (en utilisant bien sûr l'opérateur de division `/` de Python) :

```
>>> 3/2
1
>>> -3/2
-2
```

Ces deux essais auraient dû donner les valeurs 1.5 et -1.5, mais on obtient 1 et -2. La valeur -2 provient du fait que Python trouve la valeur négative - 1.5 puis l'arrondit à l'entier suivant dans le sens négatif, c'est-à-dire -2.

Pour ne plus souffrir de ce problème de calcul faux, il suffit de spécifier au moins un des nombres comme n'étant pas entier : il suffit d'ajouter au nombre un point et un chiffre zéro. Cela en fait automatiquement un nombre à virgule flottante.

Mettons cela immédiatement en pratique en ajoutant un point et un chiffre zéro à l'une des valeurs (même si cela semble inutile) :

```
>>> 3 / 2.0
1.5
```

Parfait ! Mais si tu as stocké le nombre dans une variable en tant que nombre entier, tu ne peux plus ajouter le point et le zéro à la fin du nom de la variable. Dans ce cas, utilise la technique radicale fondée sur la fonction de conversion d'un entier vers un flottant qui porte le nom `float()`. Elle force la valeur à devenir un nombre flottant.

Voici un exemple :

```
>>> a = 2
>>> 3 / a
1
>>> 3 / float(a)
1.5
```

Des chiffres et des lettres

Je rappelle un petit extrait que nous avons vu quelques pages auparavant. Inutile de le ressaisir :

```
>>> mon_invite = 'Propose un nombre : '
>>> raw_input(mon_invite)
Propose un nombre : 17
'17'
```

Dans la dernière ligne, la valeur que Python affiche est délimitée par des apostrophes `'`, ce qui indique que Python considère que la valeur saisi est une chaîne de caractères (un texte), et non une valeur numérique. Python ne considère pas qu'il s'agit de la valeur numérique 17, mais des caractères 1 et 7.

Une technique très simple pour savoir si Python considère une valeur comme numérique ou pas consiste à essayer de lui ajouter une autre valeur numérique :

```
>>> a = 1
>>> a + 1
2
```

Dans cet extrait, nous stockons d'abord la valeur numérique 1 dans la variable `a` puis nous y ajoutons la valeur 1. Cela ne pose pas de problème puisque c'est déjà une valeur numérique. En revanche, nous ne pouvons pas ajouter la valeur 1 au contenu de la variable qui contient le nombre saisi par le joueur :

```
>>> nbr_joueur = raw_input(mon_invite)
Propose un nombre : 17
>>> nbr_joueur + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

Il y a quelque chose qui cloche avec la variable contenant la valeur saisie. D'ailleurs, Python donne un indice puisque son message d'erreur signifie qu'il ne peut fusionner une chaîne avec un entier, qui sont deux objets (deux types de données) différents. Tu vois qu'il ne tente même pas d'additionner les valeurs, mais de mettre au bout de la première chaîne ce qui aurait dû être une autre chaîne (mais nous avons fourni la valeur numérique 1). Nous pouvons confirmer ce que contient la variable en demandant l'affichage du contenu :

```
>>> nbr_joueur
'17'
```

La variable contient bien une chaîne puisque l'affichage est entouré d'apostrophes. Cela empêche Python de comprendre qu'il s'agit d'une valeur numérique.

La fonction `raw_input()` renvoie systématiquement une chaîne de caractères, ce qui explique tout. Même si tu saisis au clavier une valeur purement numérique, c'est une chaîne que la fonction récupère dans le programme. Il faut donc convertir cette valeur texte en valeur numérique.

Supposons que, comme par hasard, l'ordinateur ait choisi de faire deviner la valeur 17. Pour réussir la comparaison entre le contenu de la variable et cette valeur, il faut faire comme ceci :

1. **On stocke la valeur à deviner dans une variable :**

   ```
   >>> nbr_secret = 17
   ```

2. **On compare la valeur proposée par le joueur à cette valeur connue du programme en utilisant l'opérateur de comparaison `==` :**

   ```
   >>> nbr_secret == nbr_joueur
   False
   ```

   La ligne source précédente demande de comparer une variable numérique à une variable chaîne telle que saisie au clavier, c'est-à-dire `'17'`. Cette comparaison renvoie donc très logiquement la valeur `False`, puisqu'elle est fausse. Si la saisie était une carotte, la valeur numérique serait un navet.

3. **Revoyons la ligne précédente en forçant la conversion de ce qui a été saisi vers le type numérique au moyen de la fonction de conversion `int()` :**

   ```
   >>> nbr_secret == int(nbr_joueur)
   True
   ```

Cette fois-ci, cela fonctionne ! La fonction standard `int()` analyse la chaîne de caractères que tu lui fournis en cherchant à produire une valeur numérique. Elle échoue dans son traitement si la chaîne contient autre

chose qu'une séquence de chiffres sans point décimal. En revanche, elle n'est pas gênée s'il y a des espaces inutiles avant ou après le ou les chiffres.

Voici un exemple qui échoue :

```
>>> int('1.0')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '1.0'
```

Cet essai échoue parce que l'écriture `'1.0'` n'est pas une valeur numérique entière. La fonction `int()` ne sait pas convertir une valeur décimale. Dans ce second exemple en erreur, Python ne parvient pas à convertir les lettres dans la chaîne qu'on lui propose :

```
>>> int('1 jour parfait')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '1 jour
parfait'
```

Finissons avec un essai fructueux. La conversion réussit même s'il y a des espaces autour de la valeur :

```
>>> int('   17          ')
17
```

Comparons avec des si (if)

Après avoir recueilli au format d'un nombre ce que le joueur a tenté de deviner, il faut maintenant pouvoir comparer cette valeur au nombre que connaît le programme.

Rappelons d'abord les deux grandes règles :

- Le signe `:` (deux-points) implique que, juste après lui, il doit y avoir un bloc d'au moins ligne de code source décalée par rapport à la marge.
- Dans toutes les lignes qui forment ce bloc, il faut commencer par quatre espaces ou un multiple de quatre.

Voici justement notre première instruction conditionnelle. Elle est fondée sur le mot réservé `if` (mot anglais signifiant « si ») :

```
>>> if nbr_secret == int(nbr_joueur):
...     print("Correct !")
...
Correct !
```

Voici notre premier contact avec cet important mot réservé `if`. Le nom de l'instruction est suivi immédiatement par la condition d'exécution. Ici, c'est une comparaison entre la valeur que connaît l'ordinateur et la valeur saisie au clavier (après conversion vers le type numérique entier). La fin de la ligne doit obligatoirement comporter le signe deux-points `:`, qui prévient Python que ce qui va suivre est un bloc dépendant de cette instruction, un bloc de code.

Le contenu du bloc qui dépend de `if` n'est exécuté que si la comparaison renvoie un résultat vrai (`True`). Si la condition n'est pas satisfaite, le bloc contrôlé par `if`, en décalage de quatre espaces, n'est jamais exécuté. Python continue à la prochaine ligne sans décalage, donc après la fin du bloc de code. Le signe `:` de début de bloc correspond en français au « alors » de la construction « si… alors ».

Pour le vérifier, je te propose de faire les deux essais interactifs suivants, mais je rappelle ceci :

- Le signe `:` marque le début d'un bloc de code en décalage par rapport à la marge.
- Toutes les lignes qui doivent faire partie du bloc après le signe `:` doivent être décalées de quatre espaces.

```
>>> if 1 == 2:
...     print(“Correct !”)
...
>>> if 1 == 1:
...     print(“Correct !”)
...
Correct !
```

Dans le premier essai, la comparaison échoue parce que 1 n’est pas égal à 2, donc rien n’est affiché : le bloc de code est totalement ignoré. Dans le deuxième essai, 1 étant bien égal à 1, l’instruction d’affichage est exécutée, et tu vois apparaître le message demandé (`Correct !`).

LES BLOCS DE CODE

En Python, une instruction conditionnelle est composée d’un mot réservé, d’une expression, du marqueur de bloc (le signe `:`) et du bloc dépendant. Ce sont des éléments classiques dans un langage de programmation structuré. Les blocs de code constituent l’une des techniques les plus importantes pour contrôler et faire varier l’exécution d’un programme. Le chemin que prend l’interpréteur Python parmi toutes les lignes d’un programme correspond au flux d’exécution. Tu vas rapidement prendre l’habitude de comprendre et de suivre ce flux mentalement.

Si oui ou sinon (else)

Que faut-il faire lorsque le joueur n’a pas deviné le bon chiffre ? Il faut que Python affiche un message pour l’en informer.

Il faut toujours penser à informer l’utilisateur du résultat de ses actions, même dans les programmes les plus simples.

Découvrons donc un autre mot réservé : `else` (« sinon » en anglais). C’est le partenaire privilégié de `if`. Il n’y a pas besoin de fournir de condition pour `else`. C’est une branche d’exécution qui est empruntée si la condition du `if` n’est pas satisfaite. Avec une construction à deux branches `if`/`else`, tu fais exécuter un bloc de code ou un autre, mais jamais les deux en même temps.

Note que `else` doit lui aussi être suivi du marqueur de début de bloc, le signe deux-points `:` .

Continuons avec le même exemple. Nous supposons que l’ordinateur a choisi la valeur 17. Le joueur est chanceux, puisqu’il a saisi la même valeur 17, mais sous forme de chaîne, `‘17’`. Mettons un grain de sable dans les rouages :

```
>>> if nbr_secret == int(nbr_joueur)+1:
...     print(‘Correct !’)
... else:
...     print(‘Faux ! Recommence’)
...
Faux ! Recommence
```

Dans l’exemple, j’ai volontairement ajouté 1 avec la mention `+1` pour que les valeurs ne soient pas identiques, ce qui permettra de déclencher l’exécution du bloc qui dépend de `else`.

Tu remarques que `else` est doté d’un signe `:` et donc d’un bloc de code comme `if`. Ce bloc est effectivement en décalage par rapport à la marge gauche. Il faut que ce décalage dans la branche du `else` soit le même que dans la branche `if` correspondante.

Si la condition du `if` est vraie, le bloc du `if` est exécuté, et celui du `else` (« sinon ») ne l'est pas. Si la condition du `if` est fausse, son bloc est ignoré, et c'est celui du `else` qui est exécuté. Il n'y a là rien de bien sorcier.

Si, sinon si, sinon (elif)

Pour rendre ce jeu de devinette un peu moins difficile, nous allons donner un indice au joueur en lui précisant si ce qu'il a deviné est supérieur ou inférieur à la valeur attendue.

Pour y parvenir, nous allons créer une troisième branche de test avec un nouveau mot réservé qui s'écrit `elif`. Nous pourrons ainsi tester successivement si la valeur est bonne, si elle est trop grande ou si elle est trop petite.

```
>>> if nbr_secret == int(nbr_joueur):
...     print('Correct !')
... elif nbr_secret > int(nbr_joueur):
...     print('Trop petit')
... else:
...     print('Trop grand')
...
Correct !
```

Le mot réservé `elif` est une abréviation de *else-if* (« sinon-si »). Il permet de créer des branches de conditions intermédiaires. En théorie, tu peux insérer autant de branches `elif` que nécessaire.

Prenons un exemple tout à fait différent. Supposons que nous désirions lancer une action différente selon que l'utilisateur a tapé le chiffre 1, 2 ou 3, comme dans un menu. Il suffit d'ajouter deux branches `elif` :

```
>>> a = 3
>>> if a == 1:
...     print('a vaut 1 !')
... elif a == 2:
...     print('a vaut 2 !')
... elif a == 3:
...     print('a vaut 3 !')
... else:
...     print("Je ne sais pas ce que vaut a.")
...
a is 3!
```

En pratique, quand tu auras besoin de créer plus de trois branches `else` et `elif`, il vaudra mieux te tourner vers une autre solution, dont je parlerai plus loin.

Voyons si cette structure fonctionne. Nous allons l'utiliser dans notre exemple de devinette :

```
>>> nbr_joueur = 17
>>> nbr_secret = 16
>>> if nbr_secret == int(nbr_joueur):
...     print('Correct !')
... elif nbr_secret > int(nbr_joueur):
...     print('Trop petit')
... else:
...     print('Trop grand')
...
Trop grand
>>> nbr_secret = 18
>>> if nbr_secret == int(nbr_joueur):
...     print('Correct !')
... elif nbr_secret > int(nbr_joueur):
...     print('Trop petit')
```

```
... else:
...     print('Trop grand')
...
Trop petit
```

Puisqu'il y a deux essais successifs ci-dessus, tu peux croire que je te demande de saisir deux fois les sept lignes de code source de l'exemple. En fait, tu peux profiter d'une fonction copier/coller loin d'être inutile. Voyons comment l'utiliser.

1. **Avec le curseur, remonte dans la fenêtre et place-toi à la fin de la ligne `nbr_secret = 18` puis frappe** Entrée**.** La ligne est copiée dans la dernière ligne, celle de l'invite. Tu peux modifier le contenu. Ici, remplace la valeur `16` par `18` puis valide par Entrée.
2. **Remonte le curseur à la fin de la première ligne du grand bloc conditionnel commençant par `if` et frappe** Entrée**.** C'est magique : tout le bloc (six lignes) est recopié en bas ! Tu peux encore appliquer des retouches avant de valider pour lancer l'exécution. Pratique, non ?

L'édition sera encore plus facile à partir du Projet 4, car nous pourrons sauvegarder le code source et le recharger pour le modifier.

Voici quelques observations importantes concernant le précédent exemple :

» Toutes les lignes d'affichage `print()` sont décalées du même nombre d'espaces.

» Toutes les lignes des mots réservés `if`, `elif` et `else` commencent sur la marge gauche.

» Dans l'exemple, le nombre à deviner a d'abord été choisi plus petit, puis plus grand que le nombre soi-disant saisi par le joueur (17).

» Dans le premier essai, le programme a bien affiché que le nombre était trop grand, et qu'il était trop petit dans le second essai.

Pour tourner en boucle (while)

Nous savons demander à l'utilisateur de saisir une valeur, de la convertir vers un type numérique et de la comparer à celle que connaît le programme. Il nous faut maintenant prévoir un mécanisme pour que le programme recommence à poser la question tant que le joueur n'a pas trouvé la réponse. Voici comment faire :

» Nous allons créer une grande boucle qui va englober tous les tests en utilisant le mot réservé `while` (« tant que »).

» Pour sortir de cette boucle qui ne se termine normalement jamais, nous allons utiliser un autre nouveau mot réservé nommé `break` (« sortir »).

```
>>> nbr_secret = 17
>>> invite = 'Quel est ton nombre ? '
>>> while True:
...     nbr_joueur = raw_input(invite)
...     if nbr_secret == int(nbr_joueur):
...         print('Correct !')
...         break
...     elif nbr_secret > int(nbr_joueur):
...         print('Trop petit')
...     else:
...         print('Trop grand')
...
Quel est ton nombre ? 3
Trop petit
Quel est ton nombre ? 93
Trop grand
```

```
Quel est ton nombre ? 50
Trop grand
Quel est ton nombre ? 30
Trop grand
Quel est ton nombre ? 20
Trop grand
Quel est ton nombre ? 10
Trop petit
Quel est ton nombre ? 19
Trop grand
Quel est ton nombre ? 16
Trop petit
Quel est ton nombre ? 18
Trop grand
Quel est ton nombre ? 17
Correct !
```

Dans l'exemple imprimé, tu vois que le mot `break` est en gras. Il provoque la sortie du bloc de répétition `while`, sans tenir compte des conditions des branches suivantes. Attention : `break` ne fait sortir que d'un niveau s'il y en a plusieurs imbriqués.

Prenons un autre exemple. Tu disposes d'une longue liste de noms de couleurs et tu veux vérifier que la couleur rouge est mentionnée. Il suffit de créer une boucle comme celle de l'exemple et de passer en revue chacune des lignes de la liste. Dès que tu détectes le mot « rouge », tu peux sortir de la boucle avec `break`, car il ne sert à rien de continuer à tester les cas suivants. C'est à cela que sert le mot réservé `break`.

Eh oui ! `break` est encore un mot réservé, ce qui commence à faire un certain nombre de mots à retenir. Mais ne t'inquiète pas, il n'y en a pas tant que cela.

Mais `break` n'est utilisable que dans une boucle. Ceci ne fonctionne pas :

```
>>> break
File "<stdin>", line 1
SyntaxError: 'break' outside loop
```

Lorsque `break` est indiqué dans une sous-boucle (une boucle à l'intérieur d'une autre), il ne fait sortir que du niveau de boucle courant.

Pour mieux comprendre comment on sort d'un niveau de boucle, étudions l'exemple suivant en tenant compte des remarques suivantes.

Dans cet exemple, nous utilisons la fonction standard `str()`. Elle sert à convertir une valeur numérique en une chaîne de caractères (*STRing* signifie « chaîne »). Nous utilisons aussi l'opérateur `+` pour accoler une chaîne à la fin d'une autre (nous avons vu les opérateurs dans le Tableau 3.1).

Dans cet exemple, il y a bien deux boucles : la boucle interne travaille avec la variable `j` et la boucle externe avec la variable `i`, cette dernière avançant moins vite.

Le `break` est activé lorsque `y` prend la valeur 1 en partant de zéro. Du fait que le `break` est dans la boucle intérieure, il ne fait sortir que de cette boucle à la fin du premeir tour. On en a la preuve parce que la variable `i` continue à progresser jusqu'à 2 alors que `j` revient à zéro après avoir atteint la valeur 1.

```
>>> for i in range(3):
...     for j in range(3):
...         print(str(i)+", "+str(j))
...         if i == 1:

...             break
0, 0
0, 1
0, 2
```

```
1, 0
2, 0
2, 1
2, 2
```

Pour pouvoir sortir de la boucle externe, il faut déjà se trouver à ce moment dans cette boucle. Ce sera le cas si le test `if` est remonté d'un niveau, donc décalé de quatre espaces par rapport à la marge et non huit.

Voici ce qui se passe lorsque la boucle externe est concernée par le `break`. La variable `i` n'atteint plus la valeur 2 :

```
>>> for i in range(3):
...     for j in range(3):
...         print(str(i)+", "+str(j))
...     if i == 1:
...         break
0, 0
0, 1
0, 2
1, 0
1, 1
1, 2
```

Prends le temps de bien comprendre comment fonctionnent les boucles et le mot réservé `break`, et n'hésite pas à faire d'autres essais.

Un nombre au hasard avec randint()

Le joueur doit deviner un nombre, mais comment ton programme Python peut-il faire pour choisir le nombre à deviner ?

Lorsque tu installes Python, un certain nombre de modules complémentaires s'installent également, dont un qui porte le nom `random` et qui sert uniquement à générer des nombres aléatoires (quasiment pris au hasard). Ce module contient notamment la fonction nommée `randint()` qui génère une valeur numérique entière située à une position imprévisible dans l'intervalle constitué des deux nombres que tu lui fournis (les arguments). Les deux nombres servant de bornes sont compris dans la plage de valeurs dans laquelle la fonction choisit sa valeur. Par exemple, pour obtenir un nombre imprévisible entre 6 et 10, tu peux écrire : `random.randint(6,10)`

Voici un essai plus réaliste :

```
>>> import random
>>> random.randint(1,100)
67
>>> help(random.randint)
```

Nous n'avons encore jamais rencontré cette fonction. Tu peux donc utiliser la commande d'aide `help(randint)` au niveau de l'invite de l'interpréteur pour deviner (car c'est en anglais) ce que permet cette fonction. Si tu ne reviens pas à l'invite automatiquement, utilise la touche `q` pour sortir de l'aide.

Faisons quelques essais pour voir que la valeur générée varie bien d'une fois à l'autre :

```
>>> random.randint(1,100)
15
>>> random.randint(1,100)
72
>>> random.randint(1,100)
25
>>> random.randint(1,100)
36
```

```
>>> random.randint(1,100)
90
>>> random.randint(1,100)
81
>>> random.randint(1,100)
23
```

Tu constates que la valeur générée n'est jamais la même, qu'elle ne suit pas un motif reconnaissable et qu'elle reste bien dans la plage définie par les deux bornes.

Tu ne peux pas utiliser `random.randint()` sans demander au préalable d'importer le module dans lequel cette fonction est définie. Ce n'est pas une fonction principale comme `raw_input()` ou `str()`. Il faut indiquer à Python le nom du module dans lequel il doit chercher la fonction portant le nom indiqué. C'est à cela que sert la déclaration `import random`.

Le mot `import` est un nouveau mot réservé qui correspond à une directive d'importation d'un module complémentaire. Une fois que cette déclaration est faite, tu peux utiliser toutes les fonctions et définitions qui ont été regroupées dans le module mentionné.

Les *modules* sont très utilisés dans les langages de programmation. Ils permettent de regrouper des fonctions par domaine, et de les utiliser seulement si elles sont nécessaires. Tout ce qui concerne les nombres aléatoires est regroupé dans le module `random`, et ce qui concerne les opérations mathématiques se trouve dans le module `math`. Les opérations concernant les dates et les heures sont dans le module `datetime` et ce qui concerne l'enregistrement et la lecture de données dans des fichiers se trouve dans le module `pickle`. (Drôle de nom, n'est-ce pas ?)

Nous avons déjà utilisé plusieurs instructions et fonctions, comme `int()`, `range()` et `raw_input()`. Nous n'avons jamais besoin d'utiliser `import` pour pouvoir les utiliser. C'est parce que ces éléments font partie intégrante de l'interpréteur Python (ils s'appellent des *built-ins*, des éléments intégrés). Ces fonctions sont toujours disponibles.

En revanche, le module `random` est un module complémentaire. Tu vas déclencher une erreur si tu oses utiliser la fonction `randint()` sans demander l'importation du module `random` au préalable :

```
>>> random.randint(1,100)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'random' is not defined
```

Le module `random` est installé d'office avec l'interpréteur Python, parce qu'il fait partie de la librairie standard de Python. En revanche, il n'est pas activé par défaut. Lorsque tu en as besoin, il suffit d'ajouter la déclaration `import` au début de ton programme.

Pourquoi Python ne charge pas tous les modules dès le démarrage ? Pour deux raisons : tout d'abord, si de nombreux modules sont présents, cela occupe beaucoup plus d'espace en mémoire. Deuxièmement, s'il y a de nombreux modules, cela représente un grand nombre de fonctions, et l'interpréteur passe plus de temps à localiser celle que tu lui demandes d'utiliser. Pourquoi installer en mémoire des fonctions que tu n'utiliseras pas du tout dans ton programme ?

De plus, la directive `import` permet de faire charger un module complémentaire extérieur à la librairie standard, c'est-à-dire un module tiers (qui aura été conçu par un autre programmeur ou par toi-même). Pour installer un module tiers, il faut bien sûr l'avoir téléchargé et installé dans le bon sous-dossier de l'interpréteur Python.

PYTHON EST PLUS FORT QUE LA GRAVITÉ !

Attache tout ce qui peut tomber dans ta maison, les membres de ta famille, tes animaux domestiques puis toi-même à des points d'ancrage solides avant d'utiliser la directive suivante :

```
import antigravity
```

Les espaces de noms et le qualificateur

Il y a une autre nouveauté très importante dont je n'ai pas encore parlé dans l'écriture `random.randint()` : le nom du module `random` et le nom de la fonction `randint()` sont séparés par un point qui sert de séparateur.

Les fonctions intégrées que nous avons vues dans les projets précédents peuvent être mentionnées directement sans préfixe. Ce n'est pas le cas pour les fonctions des modules. Il faut dans ce cas adopter la technique des espaces de noms.

Un espace de noms permet d'éviter tout problème qui pourrait survenir lorsque deux éléments portent le même nom dans deux modules différents. L'interpréteur ne saurait pas lequel choisir. Tu connais sans doute le problème qui se pose lorsque deux élèves de la même classe portent le même prénom. Python doit être guidé pour trouver la fonction que tu veux qu'il utilise, et la technique consiste à spécifier le nom du module avant le nom de la fonction pour le qualifier.

La technique qui consiste à faire référence à un élément B appartenant à un objet A (`A.B`) est très utilisée en programmation. Tu t'en serviras énormément lorsque tu auras progressé dans le langage Python.

L'OPÉRATEUR D'ATTRIBUT

Personnellement, j'appelle le point dans l'écriture `random.randint()` l'opérateur d'attribut (j'explique pourquoi dans le Projet 6). De nombreux programmeurs parlent de *qualificateur* de l'espace de noms. J'ai tenté de trouver le nom officiel pour cette utilisation du signe point, mais je n'ai rien trouvé. Tu peux choisir ce que tu préfères.

Terminons nos devinettes

Nous pouvons maintenant terminer notre projet de devinette. Il n'y a plus grand-chose à faire pour arriver à la version complète. Nous allons réaliser trois opérations au début du programme.

1. **Nous allons ajouter une directive pour importer le module `random`.**
2. **Nous allons faire générer un nombre au hasard par le programme en utilisant `randint()`.**
3. **Nous allons stocker dans une variable la valeur générée.**

Voici donc le programme complet. Fais bien attention aux doubles indentations de certaines lignes (huit espaces donc).

Les trois premières lignes sont nouvelles par rapport à la précédente version.

```
>>> import random
>>>
>>> nbr_secret = random.randint(1,100)
>>> invite = 'Propose un nombre : '
>>> while True:
...     nbr_joueur = raw_input(invite)
```

```
...     if nbr_secret == int(nbr_joueur):
...         print('Correct !')
...         break
...     elif nbr_secret > int(nbr_joueur):
...         print('Trop petit')
...     else:
...         print('Trop grand')
...
Propose un nombre : 24
Trop petit
Propose un nombre : 86
Trop petit
Propose un nombre : 94
Trop petit
Propose un nombre : 98
Trop grand
Propose un nombre : 96
Trop grand
Propose un nombre : 95
Correct !
```

UN PYTHON ZEN

Essaie ceci :

```
>>> import this
```

Récapitulons

Au cours de ce projet, nous avons vu :

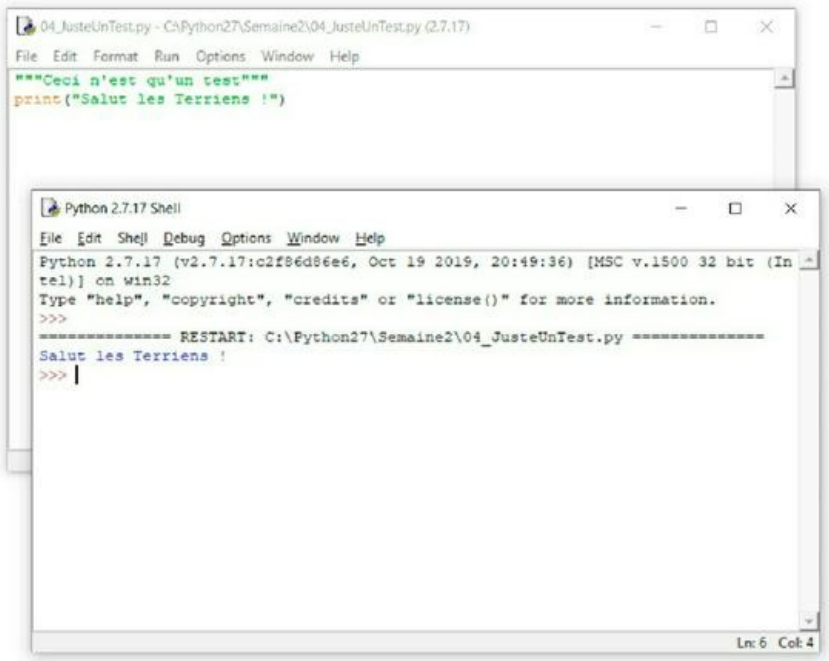
- comment afficher un message invitant l'utilisateur à saisir une information.
- que Python ne traitait pas de la même manière les valeurs numériques et les valeurs textuelles (les chaînes de caractères). Nous avons appris comment obtenir une valeur numérique à partir d'une chaîne avec `int()` ou faire la conversion inverse avec `str()`.
- comment écrire une expression avec l'opérateur `==` pour vérifier que deux valeurs sont identiques.
- la liste des opérateurs de Python, en mettant l'accent sur le point délicat concernant la division dans Python (quand les valeurs sont entières ou non).
- cinq nouveaux mots réservés : `break`, `if`, `elif`, `else` et `import`. Il n'en reste que 21 à découvrir.
- la technique des blocs de code qui consiste à ajouter des espaces par rapport à la marge gauche pour indenter certaines lignes afin qu'elles constituent un bloc dont l'exécution sera conditionnelle. Par convention, les indentations se font par quatre espaces.
- deux fonctions intégrées, `int()` et `str()`.
- comment importer un module de la librairie standard.
- la fonction de génération de valeur aléatoire du module `random` nommée `randint()`.
- une valeur prédéfinie correspondant à une condition non satisfaite, False. Il n'en reste qu'une à découvrir.

Projet 4

Découvrons l'atelier IDLE

Le Projet 3 était long et fatigant pour les doigts. Celui-ci sera bref et ne réclamera pas d'efforts digitaux. Bien au contraire, tu vas découvrir l'environnement IDLE de Python, qui va notamment te permettre de sauvegarder les lignes de code source dans un fichier pour pouvoir les réutiliser.

Tu n'auras plus besoin de faire des copier/coller pour récupérer des lignes déjà saisies. Nous utiliserons cet éditeur dans tous les autres projets du livre.



L'application IDLE simplifie la création et la relecture du code source en coloriant les mots différemment selon leur catégorie. Il contient également des outils pour ajouter des commentaires (nous verrons plus loin de quoi il s'agit) et indenter (c'est-à-dire décaler) les blocs de ligne.

Ton atelier de création

Un environnement de développement intégré est une sorte de traitement de texte spécialement conçu pour créer du code source de programmation.

Dans la mesure du possible, évite d'utiliser un traitement de texte (comme Word, WordPad, Writer ou Pages) ou un éditeur de texte (comme Bloc-notes ou TextEdit) pour écrire ou copier-coller du code source. En effet, certains caractères peuvent s'avérer inutilisables dans le langage Python. Par exemple, les guillemets français « » ne sont pas reconnus car ils doivent être de la forme “. Il en va de même pour les apostrophes, qui doivent être de la forme ‘ (et non ‘ ou ’), du moins dans la fenêtre de l'éditeur de IDLE, comme nous le verrons plus loin dans ce projet. Et de toute façon, c'est un excellent exercice de t'entraîner à écrire toi-même les lignes de code directement dans Python !

L'atelier IDLE n'est pas utilisable sur une tablette. Tu peux éventuellement trouver dans un site d'applications un atelier permettant d'avoir un environnement de développement sur tablette.

LE RETOUR DES MONTY PYTHON !

IDLE est l'acronyme de *Integrated DeveLopment Environnement*, c'est-à-dire « Environnement de développement intégré » (EDI). Comme par hasard, un des acteurs de la troupe des Monty Python (qui a donné son nom au langage de programmation, comme tu le sais déjà) se nommait… Eric Idle. Il y a même un autre environnement de développement pour Python qui a été baptisé Eric ! On reste dans la famille…

L'éditeur IDLE de Python est constitué de deux fenêtres :

- la fenêtre de l'interpréteur Shell, dans laquelle tu retrouves l'invite Python que tu connais déjà avec la fenêtre de terminal (Figure 4.1) ;
- la fenêtre de l'éditeur, dans laquelle tu écris le code source. Tu peux l'enregistrer sur disque et en lancer l'exécution.

Figure 4.1 : Aspect général de la fenêtre de l'interpréteur Shell dans IDLE.

Démarrons IDLE

Dans le Projet 1, tu as découvert comment lancer l'application IDLE (Idle), soit en épinglant l'icône dans le menu Démarrer sous Windows, soit en retrouvant l'icône dans la barre des applications ou le menu.

1. **Démarre IDLE en choisissant l'icône appropriée dans ton système.**

   Tu dois voir apparaître la fenêtre de l'interpréteur Shell avec l'invite que tu connais déjà (Figure 4.1.). Le format exact de la fenêtre varie d'un système à l'autre.

   Sous Mac OS, ouvre l'atelier IDLE en ouvrant d'abord une fenêtre de terminal puis en saisissant la commande IDLE.

   Dans tous les autres projets de ce livre, dès que tu vois l'invite habituelle >>>, tu peux démarrer l'atelier IDLE et utiliser la fenêtre de l'interpréteur Shell. Quand je te demanderai de démarrer Python, cela voudra dire qu'il faut démarrer l'atelier IDLE.

   Tu peux donc dorénavant oublier l'autre application qui est la fenêtre de terminal avec la ligne de commande, Python (command line). Tu ne t'en serviras plus que pour faire de petits essais.

2. **Pour commencer, saisis le premier programme du Projet 2 (qui tient sur une ligne) sur la ligne de l'invite de l'interpréteur :**

   ```
   print('Salut les Terriens !')
   ```

   Dès que tu valides (et même à mesure que tu saisis du texte), tu devrais voir le code prendre plusieurs couleurs (Figure 4.2).

Figure 4.2 : La coloration syntaxique permet de distinguer les différents éléments dans le code source.

L'atelier IDLE utilise des couleurs différentes selon le genre de mots dans le code source. Cela permet de distinguer les mots réservés comme `print`, les chaînes littérales comme `'Salut les Terriens'` et les valeurs numériques. Cette différenciation est très utile pour relire un programme.

Dans la Figure 4.3, j'ai saisi une seconde fois la même ligne de code source, mais en oubliant volontairement l'apostrophe fermante de la chaîne. Python affiche la parenthèse en vert et non en noir comme cela devrait l'être, ce qui te prévient qu'il manque quelque chose. Dès que tu valides par la touche Entrée, IDLE émet un message d'erreur et sélectionne le caractère « coupable ».

Autrement dit, en cas d'erreur, tu reçois un message de IDLE, ainsi qu'un une indication visuelle sur la cause probable du problème. Cela dit, il arrive que IDLE se trompe dans sa suggestion. Tu ne lui en tiendras pas rigueur, n'est-ce pas ?

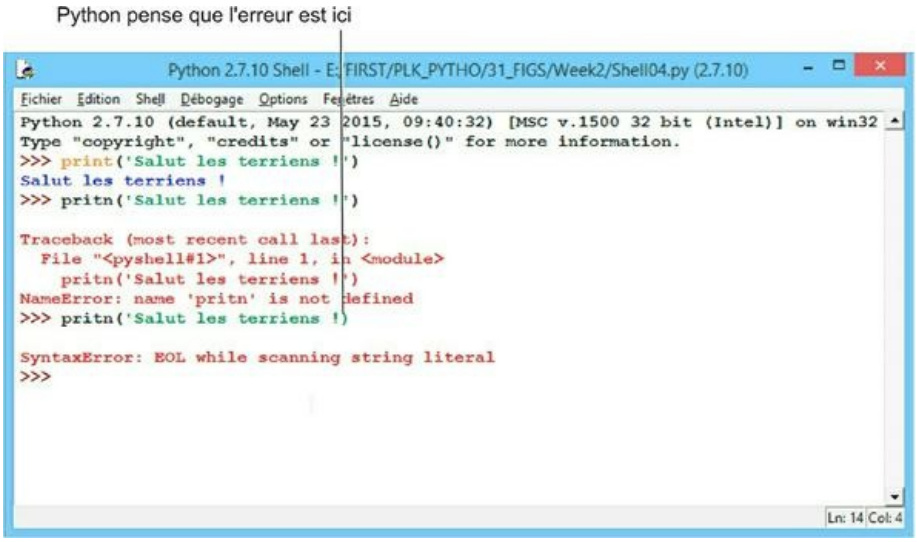


Figure 4.3 : L'atelier IDLE sélectionne ce qu'il croit être le coupable d'une erreur.

Les aides à la saisie de IDLE

L'atelier IDLE offre plusieurs mécanismes pour simplifier l'écriture du code source. Les deux plus intéressants sont l'aide à la saisie par TAB et l'historique de commandes.

Suggestion de saisie par tabulation

Ce mécanisme consiste à utiliser la touche TAB (tabulation) pour faire afficher tous les mots qui commencent avec les lettres que tu as déjà saisies. Essayons cela :

1. **Assure-toi d'être dans une ligne vide de l'interpréteur.**
2. **Saisis la lettre p et frappe la touche TAB pour activer le mécanisme.**

Tu vois apparaître une petite fenêtre donnant sur une liste (Figure 4.4).

3. **Utilise la flèche Bas du clavier pour aller jusqu'à la ligne correspondant à l'instruction `print`.**

   La lettre `p` est remplacée par le mot `print` entier.

4. **Tu peux directement saisir le paramètre de l'instruction (`'Salut les Terriens ! '`) avant de valider par Entrée.**

Si tu ne veux pas utiliser cette aide à la saisie, il suffit de continuer à saisir au clavier sans te soucier de la liste qui est apparue. Mais n'utilise pas la touche Entrée. Si tu utilises deux fois de suite la touche TAB, tu acceptes ce qui est sélectionné dans la liste. C'est un peu déroutant.

Cette aide à la saisie par TAB fonctionne également avec les noms de tes propres variables. Un exemple :

1. **Place-toi dans une nouvelle ligne dans le bas de l'interpréteur Shell.**
2. **Saisis la définition de variable suivante :**

```
ceci_est_un_long_nom_de_variable = 0
```

3. **Valide par Entrée.**
4. **Saisis les trois lettres `cec` puis frappe la touche TAB.**

   Oui, ne saisis que les trois lettres `c`, `e` et `c`.

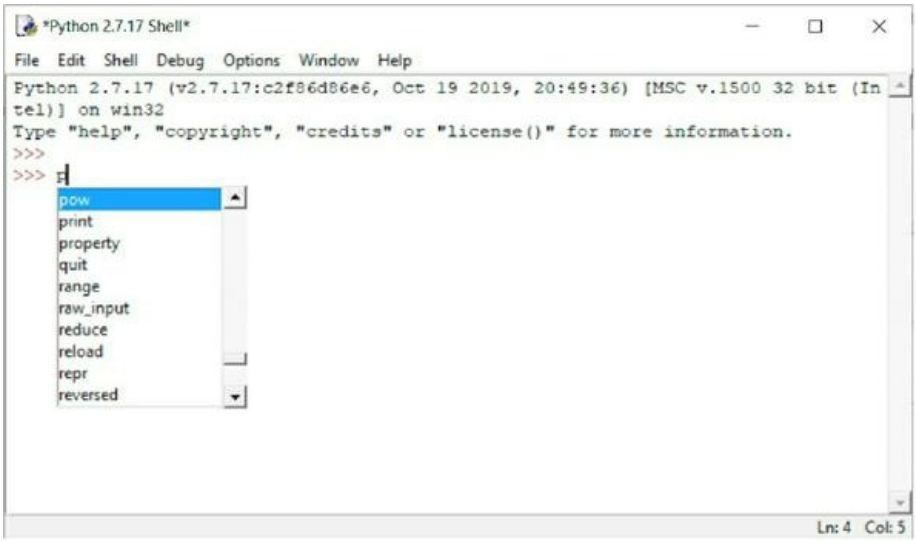


Figure 4.4 : Avec la touche TAB, tu ouvres la liste des mots possibles.

Tu vois apparaître le nom complet de la variable qui s'est placée directement dans la ligne de code. La variable a été sélectionnée d'office puisque c'est la seule qui commence par ces trois lettres dans notre exemple.

Historique de commandes

IDLE conserve dans sa mémoire les lignes déjà saisies, ce qui permet d'y revenir pour les modifier avant de les relancer. Voici comment profiter de l'historique de commandes :

1. **Utilise la flèche Haut du clavier pour remonter jusqu'à la ligne que tu voudrais exécuter.**
2. **Frappe la touche Entrée.**

   La ligne de code est recopiée au niveau de la ligne d'invite, c'est-à-dire la dernière en bas. Lorsque tu te places sur la première ligne d'un bloc en décalage par rapport à la marge, par exemple celui d'une instruction conditionnelle comme `if`, tout le bloc est copié. C'est magique !

3. **Tu peux modifier le contenu de la ou des lignes avec les flèches Gauche et Droite, la touche Ret Arr et la touche Suppr.**

4. **Tu peux alors valider la nouvelle saisie par la touche** Entrée**.**

Passons à un exemple concret :

1. **Saisis au niveau de l'invite la commande suivante, avec l'erreur volontaire consistant à omettre l'apostrophe fermante :**

```
print('Salut bel univers)
```

2. **Valide par** Entrée**.** Python se plaint immédiatement, comme le montre la Figure 4.5.

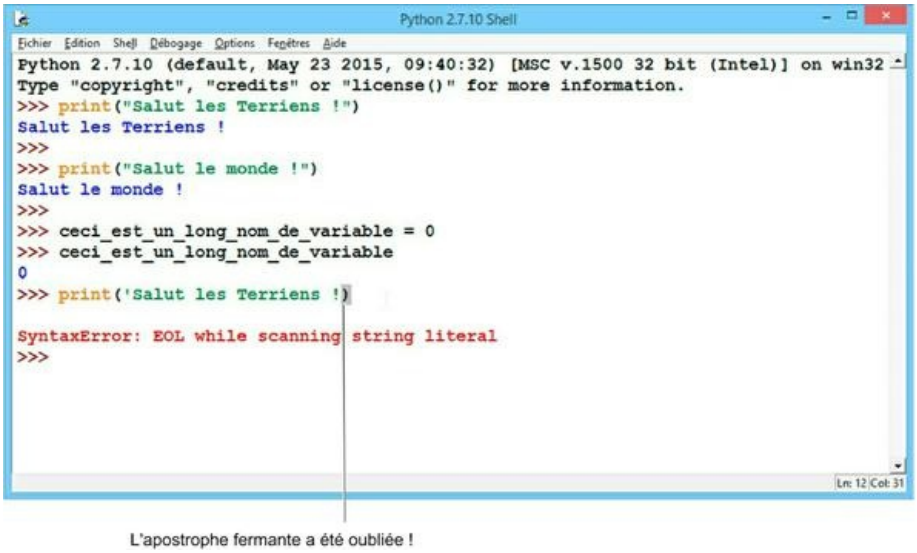


Figure 4.5 : IDLE détecte les erreurs qu'il connaît.

3. **Utilise la flèche** Haut **pour ramener le curseur à la fin de la ligne erronée.**

   Normalement, tu dois remonter de trois lignes.

4. **Valide à nouveau par** Entrée**.**

   Tu récupères au niveau de la ligne d'invite toute la ligne erronée (Figure 4.6).

5. **Avec la flèche** Droite **du clavier du curseur, va jusqu'à la fin de la ligne.**

6. **Ajoute l'apostrophe manquante puis valide par la touche** Entrée**.**

L'historique de commandes va souvent t'épargner de ressaisir quelque chose. Ce mécanisme est très utile dès que l'on a oublié un délimiteur ou fait une faute de frappe.

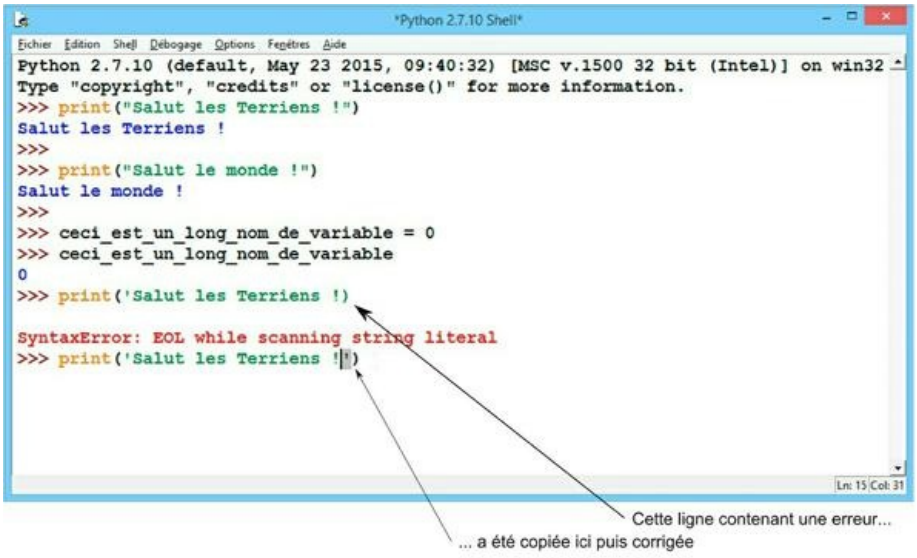


Figure 4.6 : Utilisation de l'historique de ligne de commande pour corriger une erreur.

Découvrons l'éditeur de IDLE

C'est dans la seconde fenêtre de l'atelier IDLE, celle de l'éditeur, que tu vas passer la plupart de ton temps de programmeur. La barre de menus de cette fenêtre contient la commande pour enregistrer ou sauver (Save) ton travail. Tu vas pouvoir ainsi créer ta collection de fichiers de code source, que tu pourras recharger au besoin.

Pour tester une ligne ou un petit bloc de code, tu peux utiliser la fenêtre de l'interpréteur Shell. Celle de l'éditeur sert à rédiger le code source d'un programme complet puis à l'enregistrer.

Pour ouvrir une fenêtre d'éditeur, utilise la barre de menus de la fenêtre principale de l'interpréteur : choisis la commande File/NewFile (Fichier/Nouveau Fichier) ou le raccourci Ctrl+N (sur Windows)/Commande+N (sur Mac OS).

Tu dois voir apparaître une jolie fenêtre vide.

Il n'y a pas d'invite d'interpréteur dans cette fenêtre, ni aucun message rappelant le numéro de la version de Python. Cette fenêtre ne sert pas à exécuter de ligne de code source. Elle sert à écrire les lignes d'un programme puis à l'enregistrer dans un fichier pour l'exécuter dans un deuxième temps.

La possibilité d'enregistrer ton travail dans un fichier est absolument indispensable, et permet d'envisager de grands projets. Le seul inconvénient est que tu n'as pas de retour immédiat de la part de l'interpréteur après chaque ligne saisie.

Voyons comment créer notre premier fichier de code source :

1. **Observe d'abord la barre de titre de la fenêtre d'édition.**

   Pour l'instant, elle indique Untitled (Sans nom de fichier).

2. **Saisis la phrase suivante puis valide par Entrée :**

   ```
   """Ceci n'est qu'un test"""
   ```

   Je vais expliquer ce que sont les commentaires un peu plus loin.

3. **Observe à nouveau ce qui est indiqué dans la barre de titre de la fenêtre.**

   Dorénavant, la mention dans la barre de titre est entourée de deux astérisques, ce qui t'avertit que la fenêtre contient des informations qui n'ont pas encore été enregistrées dans un fichier.

4. **Saisis la seconde ligne suivante et valide par Entrée :**

   ```
   print("Salut les Terriens !")
   ```

   Rien ne se passe, alors que tu as validé ta ligne par Entrée. La fenêtre de l'éditeur doit avoir l'aspect de la Figure 4.7.

5. **Dans la barre de menus, choisis la commande Run/Run Module (Exécuter/Exécuter Module) ou utilise la touche F5.**

Figure 4.7 : Saisie d'un premier programme dans l'éditeur de IDLE.

F5 est une des touches de fonction dans le haut du clavier. C'est ainsi que tu peux lancer facilement l'exécution du programme. Dans l'exemple, l'atelier IDLE répond qu'il faut d'abord enregistrer le fichier source et te demande si tu es OK.

6. **Clique** OK **dans la boîte de dialogue.**

   Tu vois apparaître une boîte de dialogue d'enregistrement de fichier.

7. **Dans la zone de texte du nom de fichier, saisis le nom suivant :** *testidle.py***.**

   Il faut ajouter "l'extension *.py* à la fin du nom dans le fichier si l'atelier ne l'ajoute pas d'office.

   Pour l'instant, ne te soucie pas du dossier dans lequel le fichier est stocké. Laisse l'atelier choisir pour toi.

8. **Valide l'enregistrement au moyen du bouton** Enregistrer**.**

   Si tu vois apparaître une boîte de dialogue comme celle de la Figure 4.8, c'est la preuve que tu as tenté un copier/coller d'apostrophe non droite (depuis Word, par exemple) ou que tu as tapé une lettre accentuée. Le code source ne peut pas fonctionner ainsi. Reviens au texte source et vérifie tous les guillemets, apostrophes et accents (à, è, é, ê, î, ô, û, ç). (Parfois, l'atelier affiche un message d'erreur pour t'aider à trouver les endroits à corriger.)

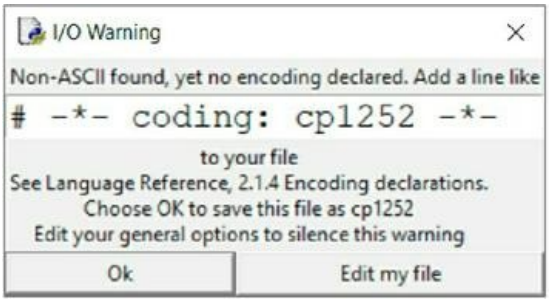


Figure 4.8 : La boîte d'erreur d'encodage des caractères.

Dès que tu acceptes la sauvegarde, l'atelier enregistre le fichier sur disque avec le nom indiqué puis lance l'exécution du fichier dans la fenêtre principale de l'interpréteur Shell. Cette fenêtre doit avoir l'aspect de la Figure 4.9.

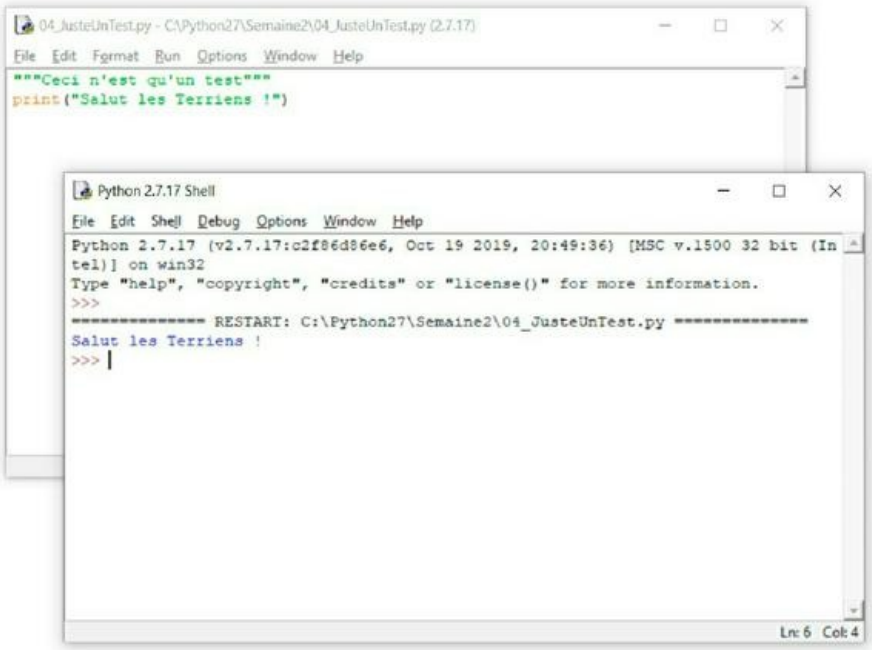


Figure 4.9 : L'exécution d'un programme se trouvant dans la fenêtre de l'éditeur (en haut) se déroule dans la fenêtre de l'interpréteur Shell qui redémarre (en bas).

Tu peux relancer l'exécution autant de fois que désiré avec la touche F5. Cela provoque à chaque fois un redémarrage de la fenêtre de l'interpréteur. À chaque redémarrage, tu perds les valeurs des variables qui étaient définies dans cette fenêtre. Tu perds également la possibilité d'accélérer la saisie par rapport à ces noms de variables, évidemment. Mais cela n'est pas grave du tout.

Deviens ton propre commentateur

Revoyons les deux lignes de code source écrites dans la section précédente :

```
"""Ceci n'est qu'un test"""
print('Salut les Terriens !')
```

La première des deux lignes est un commentaire. Apparemment, c'est une valeur littérale comme celles que nous avons rencontrées dans le Projet 2, sauf qu'elle est toute seule sur une ligne. C'est effectivement une valeur littérale de type chaîne. Quand Python voit une telle valeur toute seule, il n'en tient pas compte. Cela te permet d'ajouter des explications dans ton code source. Elles te serviront plus tard pour te rappeler à quoi sert la partie concernée dans le programme. Elles serviront aussi aux autres quand ils liront le code source de ton programme. Dans ce petit exemple, le commentaire ne nous apprend pas grand-chose, mais ce n'est qu'un essai.

Passons en revue les nombreuses raisons qui rendent très utile l'ajout de commentaires dans le code source :

» **Explications techniques.** Dès que tu commences la rédaction d'un nouveau programme Python, prend le temps d'ajouter au début une description globale de l'objectif. Ne reformule pas en français ce que fait le programme, mais résume sa fonction, rappelle l'objectif. Par exemple, si tu as une instruction qui s'écrit `A = A + 1`, il est inutile d'ajouter un commentaire du genre `#Ajoute un à A`. N'importe quel programmeur débutant a compris cela sans lire le commentaire. Il ne faut ajouter de commentaires que là où la fonction n'est pas évidente en lisant le code source. Voici un meilleur commentaire pour la même ligne :

```
# En changeant la valeur de A, les données sont
# réactualisées dans le bloc suivant.
```

» **Aide-mémoire.** Lorsque tu crées ton programme, tu es très intime avec son contenu, tu sais exactement pourquoi tu as écrit chaque instruction là où elle se trouve. Mais lorsque tu reviens sur le même programme

plusieurs semaines ou mois plus tard, les choses sont bien moins évidentes. Il est très utile de retrouver une description pour se remémorer ce que l'on a fait.

» **Communication entre programmeurs.** Si tu partages ton code source avec d'autres, il est très utile de donner une description de ce qu'il fait. Les commentaires permettent de coopérer plus efficacement. Si tu comptes faire partie un jour d'une équipe de programmeurs, il est conseillé de prendre l'habitude dès le départ de bien commenter son code.

» **Aide au débogage.** S'il y a un souci dans le code source, il faut que quelqu'un puisse résoudre le problème, et ce n'est pas nécessairement toi. C'est une autre raison pour ajouter des commentaires décrivant le fonctionnement du code.

» **Une saine discipline.** Commenter le code, c'est bon pour ton mental de programmeur.

Les commentaires de fin de ligne (dièse #)

En théorie, tous les délimiteurs de valeurs littérales peuvent servir à créer des commentaires (apostrophes et guillemets simples, doubles et triples). Par convention, on réserve le triple guillemet comme délimiteur de commentaires : `"""J'en suis un"""`. L'autre manière de créer un commentaire est réservée aux fins de ligne. Ce genre de commentaire commence par le signe `#` (dièse) et n'a pas besoin de marqueur de fin de zone de commentaires.

Dès que l'interpréteur détecte ce `#`, il ignore tout ce qui le suit jusqu'à la fin de la ligne. Ce marqueur est très pratique pour ajouter une courte explication en plein milieu du code source. Il est plus pratique puisqu'il suffit d'un caractère. Lorsque tu veux créer un commentaire sur plusieurs lignes, le signe `#` est moins pratique, puisqu'il faut en ajouter un au début de chaque nouvelle ligne.

L'exemple suivant montre comment utiliser les deux types de commentaires.

Listing 4.1 : 04_Commenter.py

```
""" Ceci est un simple fichier de test """

print("Salut de l'auteur") # Commentaire monoligne

""" Le signe # sert pour un bref commentaire en fin
de ligne, mais pas pour une longue description.
"""

#############################################################
# On peut cependant se servir du signe # pour ajouter
# des commentaires en bloc pour les mettre en valeur.
#############################################################
print('Commentaires non pris en compte.') # Celui-ci non plus
```

DIÈSE OU OCTOTHORPE (#)

Ce que j'appelle un signe dièse est en réalité un croisillon ou un octothorpe (ou encore carré au Quebec), car les barres horizontales du dièse des musiciens sont légèrement inclinées. L'habitude a été prise d'appeler ce symbole dièse, et les messageries vocales demandent toutes d'« appuyer sur la touche dièse ». Nous allons donc continuer à l'appeler ainsi, tout en sachant que son vrai nom est *octothorpe*. Les anglais parlent souvent de signes *sharp* (qui signifie « dièse »).

QUEL DOSSIER SOUS WINDOWS ?

Dans notre premier essai de sauvegarde, nous avions accepté que le fichier soit créé dans le dossier choisi par Python. Sous Windows, si tu as procédé à l'installation normale, cela correspond au dossier *C:\Python27*. Pour la suite de tes travaux, mieux vaut définir un dossier personnel. Mais pour l'instant, je te laisse stocker les fichiers dans le dossier (répertoire) prédéfini par l'interpréteur pour que tu retrouves plus aisément les fichiers sauvegardés. (Note que sous Mac OS, les choses sont différentes.)

Je rappelle qu'un commentaire délimité par des triples guillemets peut s'étaler sur autant de lignes que nécessaire. Il ne se termine que lorsque l'interpréteur rencontre un autre triplet de guillemets. Dans l'exemple, le bloc construit avec des # est en réalité une suite de quatre lignes commençant par un #.

Sauvegardons la saisie de l'interpréteur

À la fin du Projet 3, tu as peut-être été un peu ennuyé de voir qu'il fallait ressaisir quasiment le même programme pour faire tes essais avec le jeu de devinette. Sache que tu peux dorénavant enregistrer sur disque toute la session de travail puis la relancer.

Dans la fenêtre de l'interpréteur Shell, la commande File/Save stocke dans un fichier toutes les lignes présentes dans la fenêtre. Évidemment, cela enregistre également le message initial qui indique le numéro de version. Autrement dit, tu ne peux pas faire exécuter ce fichier comme un vrai programme Python. Tu sauvegardes la session au cas où tu désires pouvoir récupérer certaines des lignes.

Pour neutraliser des lignes de code

Pendant la création d'un programme, tu auras de temps à autre besoin de conserver certaines lignes de code dans le fichier tout en les rendant temporairement inactives pour éviter de devoir les ressaisir. Ce sera par exemple le cas lorsque tu pressens un problème à un certain endroit, et que la ligne correspondante doive être mise hors service pour pouvoir tester une autre partie du programme. Ou bien tu écris deux façons différentes de résoudre un problème, et tu veux les essayer tour à tour. Dans tous les cas, la solution consiste à neutraliser des lignes de code en les transformant en commentaires.

Lorsque tu veux empêcher certaines lignes d'être exécutées, il n'y a absolument pas besoin de les effacer : il suffit de les neutraliser, ce qui revient à ajouter un # au début de la ligne.

L'atelier IDLE permet de neutraliser plusieurs lignes à la fois. Voyons cela avec un exemple.

Listing 4.2 : 04_CodeNeutra.py

```
""" Fichier : 04_CodeNeutra
Montre comment neutraliser du code source. """

# Pour marquer une section

##print('Imagine que, au lieu de ces instructions print(),')
##print('il y a du code que Python doit temporairement ')
##print('ignorer pour pouvoir tester une autre partie ')
##print('du programme en cours de conception. ')

# Autre section dans laquelle on doit travailler
print('As-tu bien dormi cette nuit ?')

# Suite du programme
```

1. **Nous travaillons à partir d'un exemple que je ne te demande pas de ressaisir.**

2. **Nous sélectionnons les lignes que nous voulons neutraliser.**

   Il suffit de glisser avec la souris pour sélectionner les lignes ou de maintenir la touche Maj tout en se déplaçant avec les flèches Haut et Bas.

   Tu peux voir les lignes que j'ai sélectionnées dans la Figure 4.10.

3. **Nous choisissons alors la commande Format/Comment Out Region (Commenter région).**

   On peut également utiliser le raccourci Alt+Maj+3. Cette commande fait ajouter un signe # au début de chacune des lignes sélectionnées (Figure 4.11).

Si on lance l'exécution du programme, les lignes neutralisées sont totalement ignorées. Tu peux alors chercher à résoudre le problème avant de réinstaurer ces lignes. La procédure est très simple :

1. **Sélectionne les lignes à réinstaurer et choisis la commande Format/ Uncomment Region (Décommenter région).**

   Tu peux également utiliser le raccourci Alt+Maj+4.

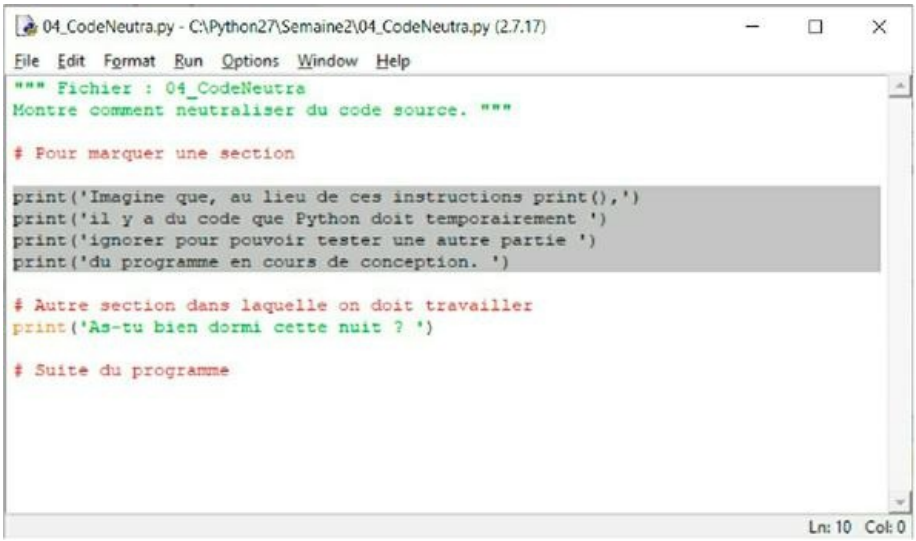


Figure 4.10 : Sélection des lignes de code à neutraliser.

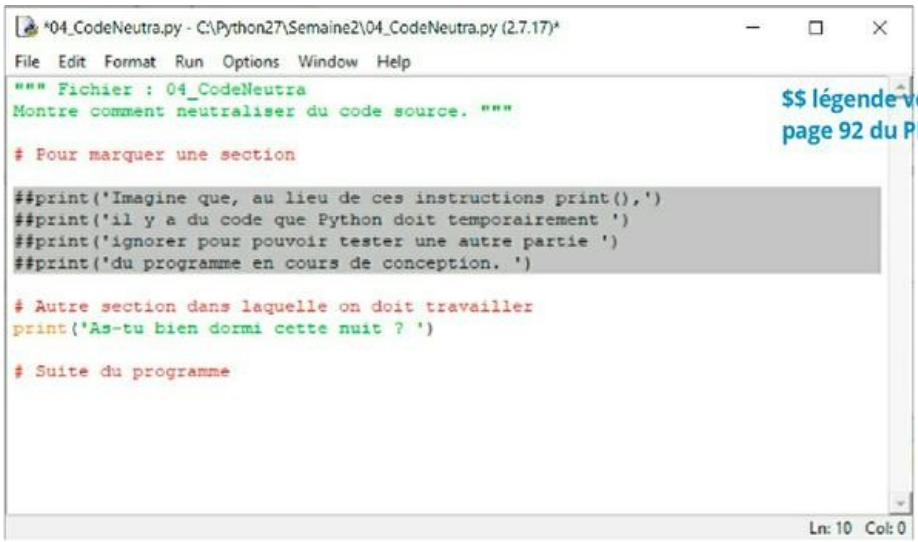


Figure 4.11 : Signes # ajoutés au début des lignes.

Indentons et désindentons

Puisque le langage Python n'utilise aucun symbole pour délimiter les lignes qui constituent des blocs, le nombre d'espaces en début de ligne a une très grande importance. Tu auras donc souvent besoin d'insérer automatiquement des multiples de quatre espaces. En ajouter consiste à *indenter* et en enlever consiste à *désindenter*.

Supposons qu'une instruction d'affichage qui fait partie du corps du programme doive devenir une instruction du bloc d'une boucle ou d'une instruction conditionnelle. Il faut donc indenter la ligne en ajoutant quatre espaces à gauche.

Pour une seule ligne, tu iras plus vite en ajoutant directement les quatre espaces, mais quand il y a toute une série de lignes, la commande suivante est très pratique. Il y a également une commande complémentaire pour supprimer quatre espaces.

Voyons comment fonctionnent ces deux commandes.

1. **Nous partons d'un code exemple tiré de l'avant-dernier exemple.**

Listing 4.3 : 04_CodeIndenter.py

```
""" Ceci est un simple fichier de test """
DEBOGUER = True

JOUR_FERIE = True

print("Salut de l'auteur") # Signe # de commentaire monoligne

""" Le signe # sert pour un bref commentaire en fin de ligne
mais pas pour une longue description.
"""

#############################################################
# On peut cependant se servir du signe # pour des
# commentaires en bloc pour les mettre en valeur.
#############################################################
if DEBOGUER:
    print("Autre ligne")
    if JOUR_:
        print("On ne travaille pas aujourd'hui.") # Non plus
```

2. **Il faut d'abord sélectionner les lignes à indenter.** La méthode est la même que pour sélectionner des lignes à neutraliser, puisqu'il suffit de cliquer à la souris tout en glissant ou bien d'utiliser la touche Maj en se déplaçant avec les flèches du curseur. Cette seconde méthode est préférable, car plus précise.
3. **Choisis la commande** Format/Indent Region (Indenter région)**.**
4. **Relis le code une fois indenté pour vérifier qu'il constitue bien un bloc valide.**

   Si la valeur de l'indentation n'est pas logique par rapport aux lignes précédentes, tu obtiendras une erreur de syntaxe, ou une erreur de logique, ce qui est pire.

Certains programmeurs Python ne veulent pas se plier à la convention consistant à utiliser des multiples de quatre espaces, et préfèrent indenter de deux en deux, de six en six ou de huit en huit. C'est possible en théorie, mais il faut absolument que le choix reste cohérent dans la totalité du programme.

Pour désindenter, il suffit d'utiliser la commande Format/Dedent Region (Désindenter région).

Modifions les couleurs !

Je te conseille de modifier deux paramètres de coloration qui sont perturbants pour bien lire et comprendre le code dans l'éditeur.

> » Les commentaires sont affichés en rouge, comme les erreurs. Mais cela gêne la lecture du code source . je préfère les voir en gris pour qu'ils restent discrets.
>
> » Les définitions des noms de classe et des fonctions, juste après le mot réservé `def`, sont en bleu, comme les affichages de messages dans IDLE. Comme elles sont très importantes, elles méritent d'être affichées en rouge !

Personnaliser ces deux catégories est très facile :

1. **Ouvre le menu** Options **et choisis** Configure IDLE**.**
2. **Clique l'onglet** Highlighting **(ou** Coloration**), comme sur la** Figure 4.12**.**
3. **Dans le panneau qui représente un extrait de code source, clique la première ligne des commentaires, puis le bouton** Choose Colour for : **(ou** Choisir la couleur de : **) qui se trouve au-dessus.**
4. **Une palette de couleurs apparaît. Dans la dernière ligne de godets, choisis un gris clair ou moyen et valide en cliquant le bouton** OK**.**

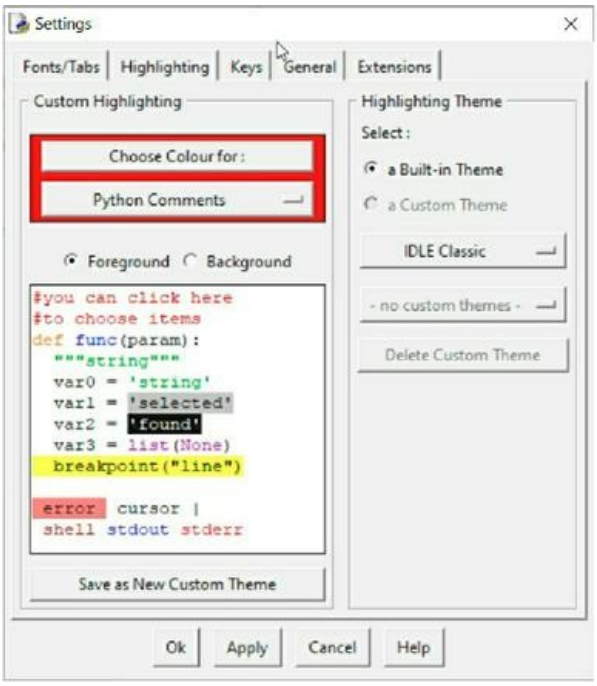


Figure 4.12 : La boîte de personnalisation de la coloration syntaxique.

5. **Si la boîte de dialogue** New Custom Theme **(ou** Nouveau thème perso**) s'affiche, saisis le nom à donner à ta configuration personnelle. Le choix de ce nom est totalement libre ! Clique le bouton** OK **pour valider.**
6. **Répète les Étapes 3 et 4 en changeant cette fois la couleur de la catégorie des définitions : clique pour cela dans le deuxième mot de la ligne 3,** `func`**, puis choisis un beau rouge.**
7. **Avec les deux boutons radio de droite sous** Select **(ou** Sélection**), tu peux comparer les deux thèmes.**
8. **Referme la boîte en cliquant le bouton** OK**.**

Récapitulons

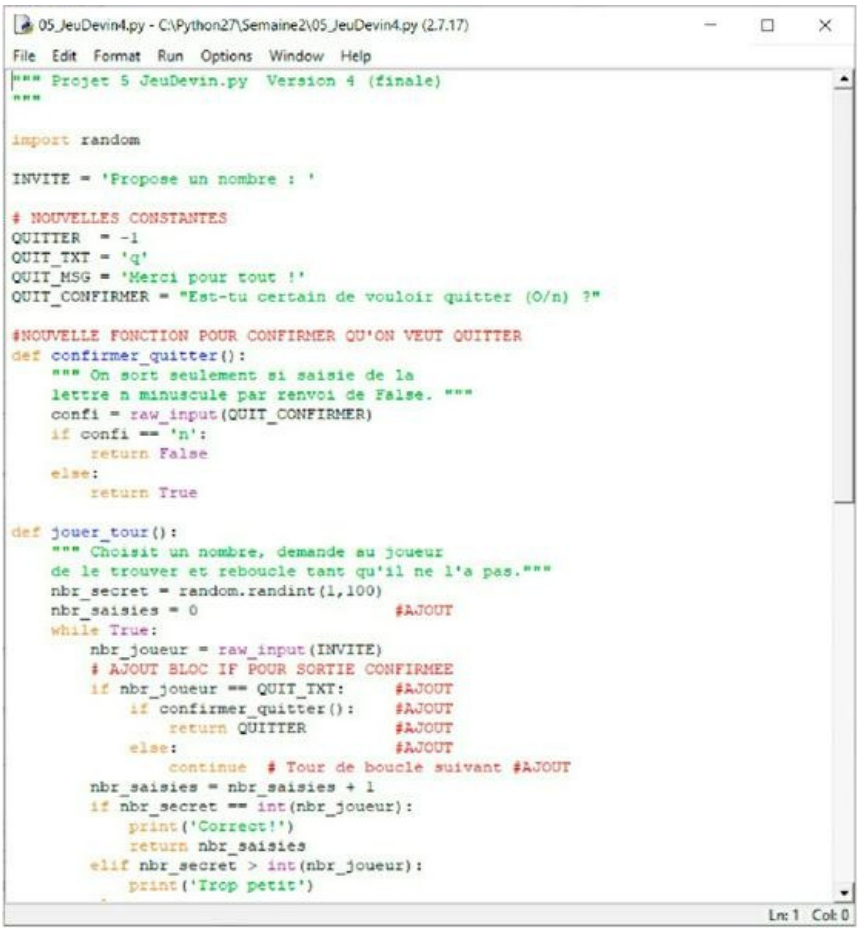
Au cours de ce projet, nous avons vu :

- comment démarrer et arrêter l'atelier IDLE ;
- les deux fenêtres de IDLE : celle de l'interpréteur Shell et celle de l'éditeur ;
- comment ajouter des commentaires dans le code source ;
- comment neutraliser des lignes de code en les transformant en commentaires ;
- comment indenter et désindenter rapidement un bloc de lignes.
- comment personnaliser la coloration syntaxique du code source dans l'éditeur.

Projet 5

Ta fonction : deviner()

Ce grand projet va te permettre de découvrir un des concepts les plus importants en programmation : les *fonctions*. Nous allons profiter de ces nouvelles connaissances pour revoir le jeu de devinette du Projet 3 en y ajoutant des fonctions. Cela va nous permettre de commencer à écrire nos programmes de façon structurée.



Au cours de ce chapitre, nous allons apprendre à communiquer avec une fonction et à lui faire renvoyer une valeur. En ce qui concerne les variables, nous verrons qu'il faut faire attention à leur portée, locale ou globale. Nous allons concevoir une fonction pour faire confirmer au joueur qu'il souhaite vraiment quitter le programme. Tu pourras t'en resservir dans tes autres projets. Recycler et réutiliser est une pratique habituelle en programmation.

La fonction : un sous-programme

Une *fonction* est le résultat d'une manière d'écrire le code source qui isole une séquence d'actions du reste du programme. Ces actions peuvent alors être déclenchées en indiquant simplement le nom de la fonction depuis une ligne extérieure à cette fonction (en invoquant la fonction). Cela évite des répétitions dans le code source et simplifie grandement la compréhension du programme et sa mise à jour.

Souviens-toi qu'au cours du Projet 3, nous avons été forcés de ressaisir les lignes de code source du jeu de devinette avant chaque exécution. Pour éviter ce travail, nous disposons maintenant de l'éditeur IDLE découvert dans le Projet 4. Il nous permet d'enregistrer les lignes dans un fichier et de les exécuter autant

de fois que nécessaire. Nous savons également regrouper une série de lignes sous forme d'un bloc dans une instruction de répétition en boucle `while`.

Ces techniques (sauvegarde/rechargement ou répétition) ne sont pas très pratiques lorsque l'on veut réutiliser du code source ; et la lecture devient de plus en plus complexe. Il faut subdiviser les traitements en plusieurs unités.

Prenons comme exemple les opérations élémentaires que tes parents te demandent de réaliser le matin pour être prêt à partir à l'école. Les voici dans l'ordre : « Te lever », « T'habiller », « Prendre ton petit déjeuner », « Mettre tes cahiers dans ton cartable », « Te brosser les dents », « Mettre tes chaussures », « Dire au revoir » et « Sortir pour aller à l'école ». Lorsque tes parents te disent de « Te préparer pour aller à l'école », toutes ces actions élémentaires sont supposées s'enchaîner, et l'on pourrait appeler la fonction globale `Te_preparer_pour_l'ecole`. Avec une seule phrase, on englobe toute une série d'instructions détaillées. Une fonction englobe donc un véritable petit programme à l'intérieur d'un autre.

Lorsque tu te libères ainsi des détails, tu peux penser à un niveau plus abstrait. Cela permet de réfléchir de façon plus générale, en subdivisant les différentes opérations par blocs. Dans les premières étapes de conception d'un programme, tu penses d'abord en termes de fonctions générales, puis tu remplis chaque fonction avec les détails techniques. Diviser pour mieux régner.

Avec les fonctions, tu peux mieux organiser ton code source et réutiliser chaque fonction.

Voici les deux obligations pour pouvoir utiliser une fonction :

- Il faut définir la fonction, avec son corps contenant les lignes d'instruction.
- Il faut ensuite invoquer la fonction en indiquant son nom ailleurs dans le programme.

Voici un exemple très simple qui reformule le programme de bienvenue aux Terriens avec une fonction. Ouvre l'atelier IDLE et saisis directement les trois lignes suivantes dans l'interpréteur Shell :

```
def afficher_salut():
    """Fonction pour afficher un message. """
    print("Salut les gens de France et de Navarre !")
```

Je rappelle qu'il faut frapper deux fois la touche Entrée pour revenir au niveau de l'invite après la dernière ligne.

Tu remarques qu'il ne se passe rien. Pour l'instant, ce que nous avons écrit n'a aucun effet. Mais nous avons défini la fonction grâce au nouveau mot réservé `def`. Il reste maintenant à appeler la fonction pour qu'elle exécute quelque chose :

```
afficher_salut()
```

Comme l'illustre la Figure 5.1, Python effectue un appel à la fonction dès qu'il rencontre son nom suivi d'une paire de parenthèses (vide ou pas, nous verrons cela). Lorsque l'exécution arrive à la ligne contenant l'appel, la prochaine action exécutée est la première ligne du bloc qui a été défini en tant que corps de la fonction, quelques lignes plus haut.

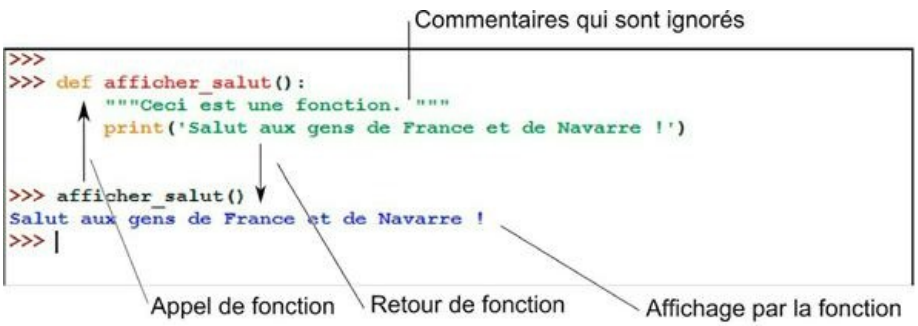


Figure 5.1 : Comment s'exécute une fonction qui a été appelée.

Le corps d'une fonction, c'est-à-dire son bloc de code, suit immédiatement la tête de définition (la ligne commençant par le mot réservé `def`) jusqu'à la dernière ligne ayant le même nombre de décalages par rapport à la marge (ou plus). La Figure 5.2 laisse entrevoir plusieurs corps de définition de fonction.

Le corps d'une fonction peut contenir des sous-blocs, donc plusieurs niveaux d'indentation.

Lorsque l'exécution arrive à la dernière ligne du corps d'une fonction, elle reprend dans la ligne qui suit immédiatement celle contenant l'appel à la fonction.

Les parenthèses qui suivent le nom de la fonction doivent toujours être présentes, même lorsqu'elles semblent inutiles. Elles servent à recevoir les valeurs que l'on veut transmettre à la fonction pour son démarrage, ce que nous verrons un peu plus loin.

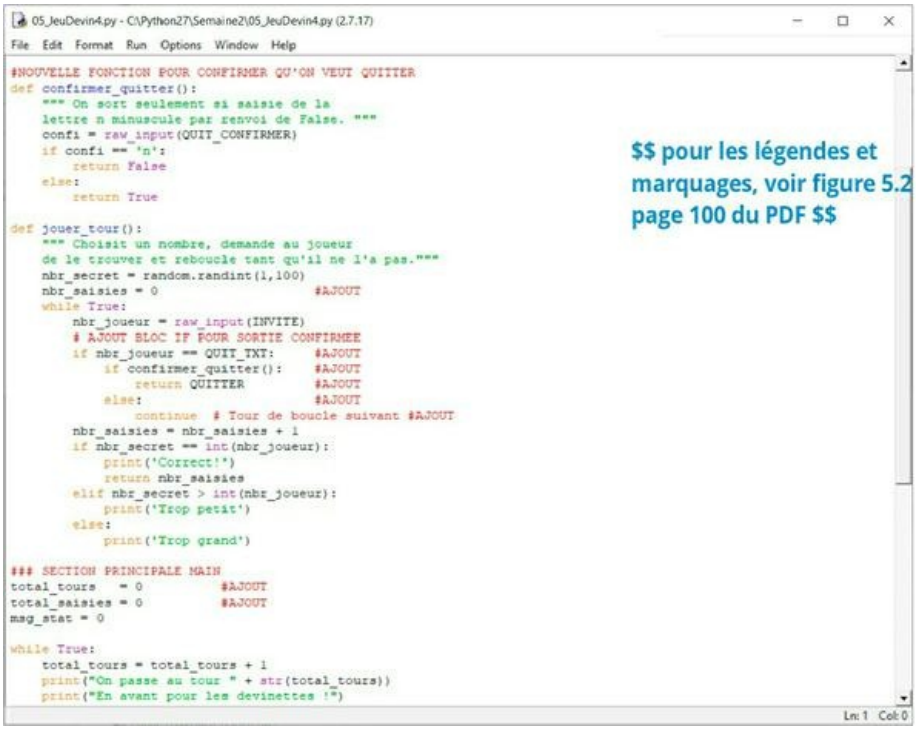


Figure 5.2 : Les lignes du corps de deux fonctions.

Si tu tentes d'appeler la fonction sans fournir les parenthèses, comme dans l'exemple ci-dessous, le programme n'affiche aucune erreur, mais il te confirme que le nom indiqué est bien celui d'une fonction et précise l'adresse en mémoire à laquelle commence la fonction (dans la mémoire RAM de l'ordinateur) :

```
>>> afficher_salut
<function afficher_salut at 0x026A1BB0>
```

Pour bien nommer tes fonctions

Les règles de choix des noms des fonctions sont très proches de celles concernant les variables :

- Le nom doit toujours commencer par une lettre ou par le caractère de soulignement (_).
- Par convention, les noms sont écrits en lettres minuscules.
- Après le premier caractère, tu peux utiliser des chiffres.
- La longueur n'est en théorie pas limitée, mais tu resteras raisonnable.
- Une fonction ne peut pas utiliser le nom d'un mot réservé du langage Python. Dans une utilisation plus avancée du langage, on peut réutiliser le nom d'une fonction prédéfinie de Python (built-in), mais évitons cela pour le moment.

Notre première fonction d'exemple est très simple puisqu'elle ne fait qu'une action. Elle ne peut pas faire varier son comportement selon les circonstances parce que nous ne lui fournissons pas de paramètres en

entrée pour permettre ces variations.

Ce mode de fonctionnement immuable convient lorsque l'on doit toujours faire faire la même chose à la fonction. Cela dit, les fonctions deviennent vraiment intéressantes lorsque l'on profite du fait qu'on peut leur transmettre des données qui font varier leur fonctionnement selon les valeurs d'entrée. Dans notre exemple, nous pourrions par exemple transmettre un message différent à chaque appel à la fonction.

Les fonctions constituent des outils essentiels pour tous les programmeurs, car elles permettent de structurer les programmes en blocs fonctionnels autonomes. D'ailleurs, toutes les fonctions prédéfinies de Python sont des fonctions comme celles que tu peux créer, et toutes les fonctions des librairies standard aussi.

Documentons nos fonctions avec les doc-chaînes

Nous avons vu dans le Projet 4 comment ajouter des commentaires pour décrire le fonctionnement global d'un programme et expliquer à quoi servent les instructions.

Dans le langage Python, il est possible de créer un type de commentaire spécial, mais il doit être placé tout au début du corps de fonction. Il s'agit des *doc-chaînes*, en anglais *docstrings*. La doc-chaîne sert à expliquer de façon générale à quoi sert la fonction. Du fait que la doc-chaîne doit toujours être placée juste après la ligne de tête de définition de la fonction, tous les programmeurs savent s'y référer aisément pour en savoir plus sur une fonction. Surtout, la doc-chaîne est exploitée par la fonction d'aide `help()` de Python, comme nous le verrons un peu plus loin.

Créer une doc-chaîne est très simple, puisqu'il suffit d'ajouter un commentaire entre triple guillemets juste après la ligne de définition de fonction :

```
def tester_docchaine():
"""Cette description concerne la fonction
tester_docchaine() et l'aide de Python
la trouve automatiquement."""
```

Le guide de style de Python concernant les doc-chaînes porte le numéro PEP 257. Il est assez long, ce qui laisse deviner qu'on peut dire bien des choses au sujet de ce mécanisme d'aide.

Sans aller jusqu'à se plonger dans cette littérature, voici les règles et conseils essentiels pour créer une doc-chaîne :

- La doc-chaîne doit être obligatoirement délimitée par des triples guillemets, comme dans `"""Je suis une doc-chaine"""`. Je rappelle qu'il ne faut pas répéter les triples guillemets à la fin de chaque ligne s'il y en a plusieurs.
- La doc-chaîne doit expliquer le fonctionnement général de la fonction. Avec l'exemple du départ à l'école le matin, nous pourrions envisager cette doc-chaîne `"""Prepare un enfant à aller à l'ecole"""`. Nous expliquons l'objectif de la fonction, pas les détails de la façon dont elle y arrive.
- La doc-chaîne doit est écrite en bon français (mais si possible sans accents, qui sont mal interprétés par Python) et non en langage abrégé ou SMS.
- Ne perds pas de temps à vouloir expliquer en détail ce qui se passe dans le corps de la fonction. Tu peux toujours revenir enrichir la doc-chaîne plus tard une fois que la fonction sera totalement écrite et testée.

En théorie, toute fonction devrait posséder une doc-chaîne. Dans la pratique, il est inutile d'en créer une pour des fonctions très simples. De plus, cela n'a d'intérêt que pour des programmes dont tu vas réutiliser certaines fonctions.

Bien qu'il s'agisse d'un commentaire, la doc-chaîne doit toujours être délimitée par des triples guillemets, car ce sont ces délimiteurs que les outils de Python recherchent et savent extraire du texte source. Tu peux ensuite appliquer la fonction `help()` standard de Python pour obtenir de l'aide sur une fonction que tu

viens de définir. Voici un exemple (tu remarques que nous n'indiquons exceptionnellement pas les parenthèses de fin de fonction) :

```
>>> help(tester_docchaine)
Help on function tester_docchaine in module __main__:

tester_docchaine()
Cette description concerne la fonction
tester_docchaine() et l'aide de Python
la trouve automatiquement.
>>>
```

Pour l'instant, tu pourrais croire que les doc-chaînes sont un peu superflues, mais c'est totalement faux. Il faut prendre dès le départ la bonne habitude de créer une doc-chaîne dès que l'on définit une nouvelle fonction. Si un jour Google décide de t'embaucher, les sélectionneurs vont s'attendre à ce que tu sois capable d'écrire en conformité avec les règles de style de Python, notamment en prévoyant une doc-chaîne pour chaque fonction.

L'AUTOMAGICIEN

Lorsque ton programme réussit à faire quelque chose automatiquement et que le résultat t'est utile, on dit que c'est de *l'automagie*.

D'autres outils sont capables d'extraire les doc-chaînes des programmes. En prévoyant des doc-chaînes pour toutes tes fonctions, tu peux facilement constituer un embryon de documentation du programme, ce qui va te faire gagner bien du temps plus tard.

Créons une amorce de fonction

Souvent, tu sauras qu'il va te falloir une fonction, mais sans pouvoir encore en écrire les détails. Tu peux dans ce cas profiter du mot réservé `pass` pour que la fonction possède au moins une ligne dans son corps afin qu'elle soit acceptée.

Dans l'exemple suivant, nous avons songé à une fonction pour résoudre la faim dans le monde, mais nous ne savons pas encore comment nous allons nous y prendre. Nous pouvons créer l'amorce de la fonction en nous servant de `pass`. Nous remplirons les détails plus tard :

```
def resoudre_faim_monde():
    pass
```

LA FONCTION PRINCIPALE MAIN

Toutes les lignes de code qui sont décalées du même nombre d'espaces sous la ligne de tête d'une fonction sont faciles à repérer. Elles seront exécutées l'une après l'autre lorsque la fonction sera appelée. Mais certaines instructions ne sont pas du tout décalées par rapport à la marge gauche. Elles n'appartiennent donc à aucune fonction. (Je ne tiens pas compte ici des décalages dans les blocs conditionnels.) À quoi correspond le niveau 1 dans un programme, celui en dehors de toutes les fonctions ? En fait, le niveau principal correspond à une fonction dont le nom n'est pas visible dans les programmes Python.

Dans d'autres langages, il y a une mention `main` (à prononcer « meïne ») pour indiquer où se trouve le programme principal, c'est-à-dire celui qui contient la première instruction qui va être exécutée lorsque le système va appeler le programme pour le faire démarrer. Le nom de cette fonction principale est `main`, parce que *main* signifie principal. Cela n'a rien à voir avec tes mains.

Étudions ce petit exemple :

```
""" Mini-programme pour mon
trer la position de main"""
def pour_tester():
   """ Corps de fonction
vide, mais valable."""

pour_tester()
```

Le code dans la fonction de test est clairement visible puisqu'il dépend de la ligne commençant par `def`. Dans la dernière ligne, nous appelons cette fonction. Cette ligne appartient à la fonction fantôme `main`. Dorénavant, tu sauras de quoi je parle quand je parle de la fonction `main` dans la suite du livre.

Lorsqu'une fonction est vide, on appelle cela une amorce (un *stub*). La fonction doit avoir au moins une ligne avec le mot réservé sans effet `pass` pour que cela ne provoque pas d'erreur lorsque l'interpréteur va la rencontrer.

Bonne nouvelle : si tu ajoutes immédiatement une doc-chaîne, tu n'as pas besoin de l'astuce consistant à insérer une ligne temporaire avec `pass`. Si tu ne sais pas encore ce qu'il faut écrire dans ta doc-chaîne, c'est peut-être que tu n'as en fait pas besoin de cette fonction. Vérifie qu'il n'y a pas une autre fonction dans ton programme qui pourrait répondre à peu près au besoin. Voici une amorce avec doc-chaîne :

```
def resoudre_faim_monde():
"""Cette fonction va trouver la vraie solution
pour que tous les humains et humaines mangent
assez chaque jour."""
```

Devinons()

En fin de Projet 3, nous étions arrivés à un petit programme complet. Je le réimprime ici, après avoir enlevé les trois signes des lignes d'invite :

```
import random
nbr_secret = random.randint(1,100)
invite = 'Propose un nombre : '
while True:
nbr_joueur = raw_input(invite)
if nbr_secret == int(nbr_joueur):
print('Correct !')
break
elif nbr_secret > int(nbr_joueur):
print('Trop petit')
else:
```

```
print('Trop grand')
```

Nous allons voir comment mettre en place une fonction pour le traitement de ce programme. Nous n'allons pas incorporer toutes les lignes, et notamment pas la directive d'importation au début. Voici comment procéder :

1. **Dans la fenêtre de IDLE, demande la création d'un nouveau fichier et donne à ce fichier le nom *JeuDevin.py*.**

   J'ai expliqué dans le Projet 4 comment faire pour démarrer un nouveau projet.

2. **Ajoute une chaîne doc-chaîne tout au début du fichier, ce qui permettra de savoir à quoi sert ce programme.**

   Je rappelle qu'il faut utiliser des triples guillemets pour délimiter cette doc-chaîne.

3. **Demande d'enregistrer le fichier pour plus de sécurité (File/Save ou Fichier/Sauver).**

4. **Normalement, il faut maintenant ressaisir le programme de la fin du Projet 3, sans les trois signes d'invite en marge gauche.**

   Pour plus de facilité, tu peux également charger dans une autre fenêtre de IDLE le fichier que je fournis dans l'archive des exemples sous le nom *03_Devinette.py* puis copier/coller depuis cette fenêtre vers la fenêtre du nouveau projet. Tu peux ensuite refermer la fenêtre d'origine.

   Nous pouvons maintenant retoucher le programme pour constituer une fonction.

5. **Rédige la ligne de tête de la définition d'une fonction en lui donnant le nom `jouer_tour()`.**

6. **Pour bien en prendre l'habitude, insère sur une ou plusieurs lignes une doc-chaîne délimitée par trois guillemets (même vide).**

7. **Sélectionne toute la boucle conditionnelle `while` qui provient du Projet 3 (neuf lignes, normalement) pour la coller juste sous la dernière ligne de la doc-chaîne de la nouvelle fonction.**

8. **Sélectionne les lignes que tu viens de copier puis demande leur indentation au moyen de la commande Format/Indent Region. Tu dois avoir indenté les lignes de quatre espaces.**

9. **Un peu plus bas que la fin de la fonction, et sur la marge gauche, ajoute un appel à cette fonction :**

   ```
   jouer_tour()
   ```

10. **Enregistre le fichier.**

    Lorsque tu procèdes à cet enregistrement, les erreurs de syntaxe éventuelles sont mises en valeur. Si cela se passe, vérifie ce que tu as saisi, en surveillant notamment les délimiteurs.

11. **Tu peux maintenant tester ton programme avec la touche F5 ou la commande Run/Run Module (Exécuter/Exécuter module).**

Normalement, le jeu de devinette doit fonctionner exactement de la même manière qu'à la fin du Projet 3. Voici l'aspect du code source de ce projet.

Listing 5.1 : Code source JeuDevin1.py (version 1)

```
""" Projet 5 JeuDevin.py  Version 1
"""
import random
nbr_secret = random.randint(1,100)
INVITE = 'Propose un nombre : '
def jouer_tour():
    """ Choix d'un nombre, demande au joueur
    de le trouver et reboucle tant qu'il ne l'a pas."""
    while True:
```

```
        nbr_joueur = raw_input(INVITE)
        if nbr_secret == int(nbr_joueur):
            print('Correct !')
            break
        elif nbr_secret > int(nbr_joueur):
            print('Trop petit')
        else:
            print('Trop grand')
# Section MAIN
jouer_tour()
```

Si tu as comparé avec soin le code source auquel tu as abouti avec celui présenté ci-dessus, tu dois avoir remarqué une différence : le nom de la variable `INVITE` contenant le message d'invite est écrit en lettres capitales. Je l'ai écrit ainsi afin de suivre la convention Python qui veut que les données qui ne vont pas changer au cours du programme (les constantes) portent des noms en lettres capitales. Il est donc logique que cette variable (qui ne variera pas) soit écrite en capitales.

Une *constante* est une donnée qui ne peut être utilisée qu'en lecture. Dans certains langages, on parle de *donnée statique*. Il est conseillé de regrouper toutes les constantes en début de fichier source. Ce sont des paramètres fixes dont il est bon de prendre connaissance au départ.

La définition d'une fonction doit toujours se trouver avant le premier appel à cette fonction dans un fichier source Python.

Notre premier problème de logique...

La première version de notre projet contient un problème qui va devenir évident lorsque nous aurons mis en place une autre boucle pour que le joueur puisse enchaîner avec une autre partie une fois qu'il a deviné le premier nombre.

Il suffit de mettre en place la boucle infinie tout à la fin du fichier, pour appeler sans cesse la fonction :

```
### SECTION PRINCIPALE MAIN
while True:
    jouer_tour()
```

Tu peux enregistrer le fichier puis lancer l'exécution par `F5` depuis la fenêtre de l'éditeur. Voici le genre de données que tu devrais voir s'afficher :

```
>>> ========================== RESTART ======================
>>>
Propose un nombre : 67
Trop petit
Propose un nombre : 77
Trop petit
Propose un nombre : 98
Trop petit
Propose un nombre : 99
Trop petit
Propose un nombre : 100
Correct !
Propose un nombre : 50
Trop petit
Propose un nombre : 90
Trop petit
Propose un nombre : 99
Trop petit
Propose un nombre : 100
Correct !
Propose un nombre : 90
Trop petit
```

```
Propose un nombre : 100
Correct !
Propose un nombre :
```

Notre joli programme fonctionne, mais il pose deux problèmes :

> » À chaque nouveau tour, le nombre à deviner reste le même (dans notre exemple, c'est la valeur 100). Une fois que le joueur a trouvé la réponse pour le premier tour, le jeu n'a plus aucun intérêt. Il faut que le programme génère une nouvelle valeur au hasard au début de chaque tour. Est-ce que tu peux trouver la solution ? Si le nombre à deviner ne change pas, c'est qu'il y a un problème au niveau de la manière dont il est généré ou stocké. Essaie de suivre mentalement le chemin d'exécution du programme et repère l'endroit où est produit le nombre à deviner. Je vais te donner la solution un peu plus loin.
>
> » L'autre problème, moins grave, est que tu ne parviens pas bien à distinguer le passage d'un tour au suivant.

… et sa solution

Dans notre programme, le nombre qu'il faut deviner ne change pas à cause de l'endroit où a été placé l'appel à la fonction de génération. Voici les étapes du déroulement d'une exécution :

1. **Le programme exécute la directive `import`.**
2. **Il génère le nombre au hasard.**
3. **Nous affectons la valeur littérale à la variable `INVITE`.**
4. **Nous définissons une fonction (mais cela ne l'exécute pas encore).**
5. **Dans le programme principal en bas, nous entrons dans une boucle `while`.**
6. **Nous répétons sans arrêt l'appel à la fonction dans cette boucle. Cela provoque l'exécution des instructions de la fonction (étape 4), sans jamais revenir à l'instruction de l'étape 2 pour générer un nouveau nombre.**

Pour obliger le programme à générer un nouveau nombre à deviner dans chaque tour de boucle, nous pouvons par exemple ajouter une copie de l'instruction qui appelle la fonction de génération dans le corps de l'autre fonction.

Pour résoudre le problème de distinction entre les tours, il suffit d'ajouter une instruction d'affichage `print` dans le programme principal en bas. Nous en profitons pour ajouter une ligne pour afficher la valeur à deviner. Bien sûr, dans la version finale, il faudra supprimer cet affichage. Dans la version actuelle, cela nous permet de vérifier que le programme fonctionne dorénavant comme prévu.

Je te propose d'étudier la version complète modifiée. J'ai ajouté d'abord un appel à la fonction `randint()` dans notre nouvelle fonction. Pour repérer la ligne, cherche le commentaire `#AJOUT` en fin de ligne.

J'ai ajouté également des instructions d'affichage dans le programme principal.

Listing 5.2 : Code source JeuDevin2.py

```
""" Projet 5 JeuDevin.py  Version 2
"""
import random
nbr_secret = random.randint(1,100)
INVITE = 'Propose un nombre : '
def jouer_tour():
""" Choix d'un nombre, demande au joueur de le
trouver et reboucle tant qu'il ne l'a pas."""
nbr_secret = random.randint(1,100)             # AJOUT
while True:
nbr_joueur = raw_input(INVITE)
if nbr_secret == int(nbr_joueur):
print('Correct !')
```

```
break
elif nbr_secret > int(nbr_joueur):
print('Trop petit')
else:
print('Trop grand')
### SECTION PRINCIPALE MAIN
while True:
# Quatre instructions d'affichage en plus :
print("C'est reparti pour un tour !")           # AJOUT
print("Chut! Le nombre secret : "+str(nbr_secret))  #AJ
print("En avant pour les devinettes !")         # AJOUT
jouer_tour()
print("")                                       # AJOUT
```

En ajoutant un appel à la fonction de génération dans notre fonction, nous avons résolu le problème de logique. Nous pourrions donc supprimer maintenant le premier appel à cette fonction puisqu'il est inutile. Pour l'instant, laisse cette ligne en place parce que j'ai besoin d'expliquer deux points très importants :

> » Tout d'abord, pour l'instant, le programme semble fonctionner comme s'il y avait deux variables homonymes `nbr_secret`. Si tu essaies le programme, tu vois que la valeur à deviner reste la même d'un tour à l'autre, même si la valeur qu'il faut deviner pour gagner change et que le programme fonctionne donc comme prévu.
>
> » Tu peux en déduire que, bien qu'une nouvelle valeur soit générée pour la variable `nbr_secret` dans la fonction, la nouvelle valeur n'est pas connue dans l'instruction d'affichage qui conserve la valeur générée tout au début.

Fais plusieurs essais pour comprendre ce qui se passe en provoquant la fin d'exécution par Ctrl+C. Dans l'extrait d'affichage qui suit, la fonction avait généré au départ la valeur 50. Pourtant, j'ai été forcé de deviner des réponses différentes à chaque tour (c'était d'abord 73, puis 17).

```
C'est reparti pour un tour !
Chut! Le nombre secret : 50
En avant pour les devinettes !
Propose un nombre : 50
Trop petit
Propose un nombre : 75
Trop grand
Propose un nombre : 63
Trop petit
Propose un nombre : 68
Trop petit
Propose un nombre : 72
Trop petit
Propose un nombre : 73
Correct !
C'est reparti pour un tour !
Chut! Le nombre secret : 50
En avant pour les devinettes !
Propose un nombre : 50
Trop grand
Propose un nombre : 25
Trop grand
Propose un nombre : 12
Trop petit
Propose un nombre : 18
Trop grand
Propose un nombre : 15
Trop petit
Propose un nombre : 16
Trop petit
Propose un nombre : 17
Correct !
C'est reparti pour un tour !
```

```
Chut! Le nombre secret : 50
En avant pour les devinettes !
Propose un nombre :
```

Une portée de variables

Notre projet est petit, mais il permet d'aborder des concepts très importants, et notamment celui de portée. La *portée* d'une variable délimite les niveaux du programme dans lesquels le nom de cette variable est reconnu pour la même donnée. Il n'est pas interdit dans Python d'utiliser le même nom de variable en plusieurs endroits du programme, mais cela peut rendre les choses délicates et merveilleuses à la fois. C'est grâce à la portée de chaque variable que Python choisit quelle valeur utiliser parmi les homonymes.

Dans notre exemple, nous utilisons à plusieurs endroits le même nom `nbr_secret` : dans la fonction `jouer_tour()`, mais également dans la fonction principale. Les deux homonymes n'entrent jamais en conflit, comme si les deux variables étaient dans des dimensions différentes. Lorsque la variable change de valeur dans une dimension, celle de l'autre dimension n'est pas modifiée. D'ailleurs, dans la fonction `jouer_tour()`, la variable est nouvelle lors de chaque appel. Lorsque la fonction se termine, la valeur de la variable est perdue.

Une variable qui est définie dans une fonction est appelée *variable locale*. La portée de la variable est limitée au bloc qui constitue le corps de la fonction. Cela permet de créer des homonymes sans entraîner de conflit, mais cela entraîne parfois des effets indésirables.

Mais ce n'est pas tout. Notre variable `nbr_secret` a été citée d'abord à l'extérieur de la fonction `jouer_tour()` et première citation signifie définition. Comment la fonction réussit-elle à trouver la valeur de la variable ?

C'est simple : lorsque Python rencontre le nom d'une variable, il cherche dans la fonction s'il y a une variable portant ce nom qui y reçoit une valeur. S'il ne trouve pas ce genre d'instruction (avec l'opérateur `=`), il remonte d'un niveau jusqu'à la fonction `main`. S'il trouve un nom de variable avec une valeur, il utilise celle de ce niveau supérieur.

En revanche, l'inverse n'est pas vrai. Si tu places une affectation de variable dans une fonction, la variable sera locale, mais elle ne pourra jamais être utilisée à l'extérieur de la fonction (à un niveau supérieur).

Prenons un autre exemple que notre jeu pour vérifier cela. Nous définissions une variable `A` dans le programme principal puis nous définissons trois fonctions nommées `test1()`, `test2()` et `test3()`. Chacune des trois essaie d'afficher la valeur que possède une variable nommée `A`. Peux-tu imaginer les résultats avant de regarder ce qui s'affiche lorsque tu exécutes le programme ?

Listing 5.3 : Code source Locales.py

```
""" 05_Locales.py
Etude des portées des variables locales"""

A = 'Je suis le contenu de la variable A de main.'

def test1():
print('Dans test1.')
print(A)
print('Sortie de test1.')

def test2():
print('Dans test2.')
A = "Contenu de la variable locale A de test2."
print(A)
print('Sortie de test2.')

def test3():
```

```
print('Dans test3.')
print(A)
A = "Contenu de la variable locale A de test3."
print('Sortie de test3.')

#Section principale
test1()
print('De retour dans main')
print(A)
test2()
print('De retour dans main')
print(A)
test3()
```

Les résultats sont affichés dans la Figure 5. 3.

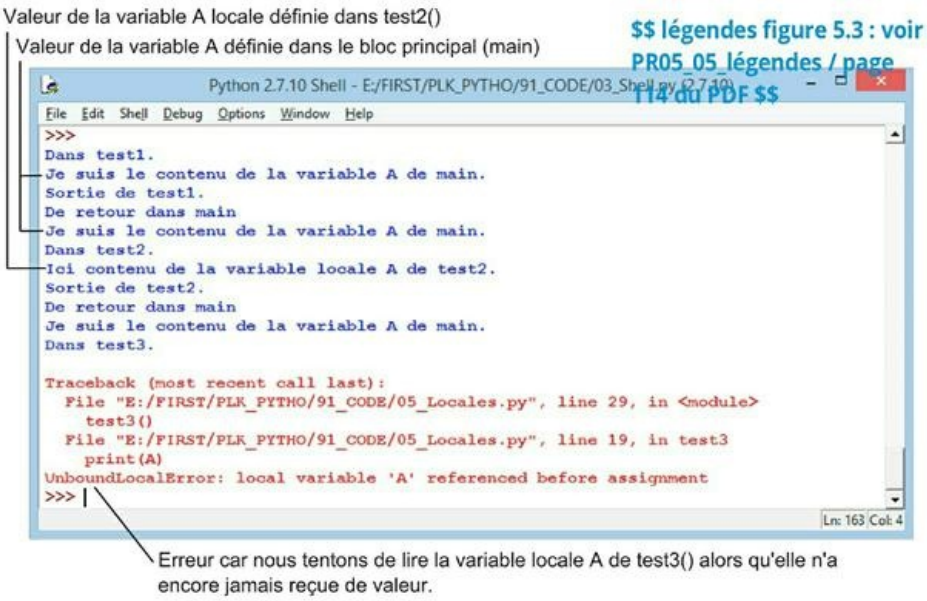


Figure 5.3 : Une variable peut en cacher une autre !

Voyons ce qui se passe. Lors de l'appel à la fonction `test1()`, Python ne trouve pas de variable locale nommée `A` ; il remonte donc d'un niveau et trouve à lire la variable globale `A`. Il ne change pas cette valeur. Lorsque nous appelons `test2()`, Python donne une nouvelle valeur à la variable `A`, qui n'est pas la même que celle du niveau supérieur ! Au retour de la fonction, nous constatons que la valeur de la variable globale homonyme n'a pas changé.

Mais pourquoi l'exécution de `test3()` échoue-t-elle ? La seule différence entre `test2()` et `test3()` est que nous tentons dans `test3()` d'afficher la valeur avant de créer la variable locale homonyme. Dans les deux cas, il y a bien une variable locale nommée `A` définie dans le corps de la fonction. Python le sait car il analyse toute la fonction avant d'exécuter sa première ligne. Il prévoit qu'il pourra utiliser la valeur d'une variable locale nommée `A`, qui est dans la portée locale. C'est pour cette raison que Python refuse d'aller chercher la variable homonyme du niveau supérieur.

Lorsque nous demandons d'afficher la valeur de la variable locale, nous le faisons trop tôt. Dans la ligne de la demande d'affichage, la variable locale `A` n'existe pas encore. Cette variable locale masque la variable globale homonyme.

Une fonction, cela communique

Pour l'instant, il n'y a pas d'échange de données entre la fonction et le niveau principal. Nous ne transmettons pas de données lors de l'appel à la fonction (les parenthèses de l'appel sont vides), et nous ne renvoyons pas de valeur depuis la fonction en fin de travail. Pourtant, les niveaux sont prévus pour fonctionner en équipe. Python dispose de mécanismes pour transmettre des données au moment de l'appel

aux fonctions et pour leur faire renvoyer une valeur en fin d'exécution. C'est ce que nous allons voir dans les deux prochaines sections.

Les arguments d'une fonction

Nous sommes déjà en mesure d'utiliser des informations extérieures à la fonction, en allant puiser la valeur d'une variable définie au niveau principal, mais à condition qu'il n'y ait pas de variable portant le même nom dans la fonction qui la masquerait. Dans ce cas, il faut faire attention à ne pas donner une nouvelle valeur à la variable dans la fonction si ce n'est pas voulu.

À l'exception des constantes pour lesquelles c'est tout à fait habituel, cette manière d'utiliser des informations extérieures à une fonction est tout à fait déconseillée. La suite explique une technique bien plus efficace.

Python permet de fournir des données en entrée d'une fonction, et c'est d'ailleurs le but même des fonctions que de recevoir des données lors de leur démarrage. Voici les conditions nécessaires pour pouvoir transmettre des données en début d'exécution d'une fonction :

1. **Tu dois d'abord faire l'inventaire des données à transmettre. Le nombre d'éléments n'est pas limité, et tu ne rencontreras jamais cette limite.**
2. **Pour chaque élément, il faut choisir un nom temporaire de variable qui ne servira que dans la fonction.**
3. **Il ne reste plus qu'à lister les différents noms des variables dans l'ordre le plus approprié entre les parenthèses de la ligne de définition de la fonction.**

Chacun des noms de variable indiqués dans les parenthèses de la définition de fonction est appelé un *argument*.

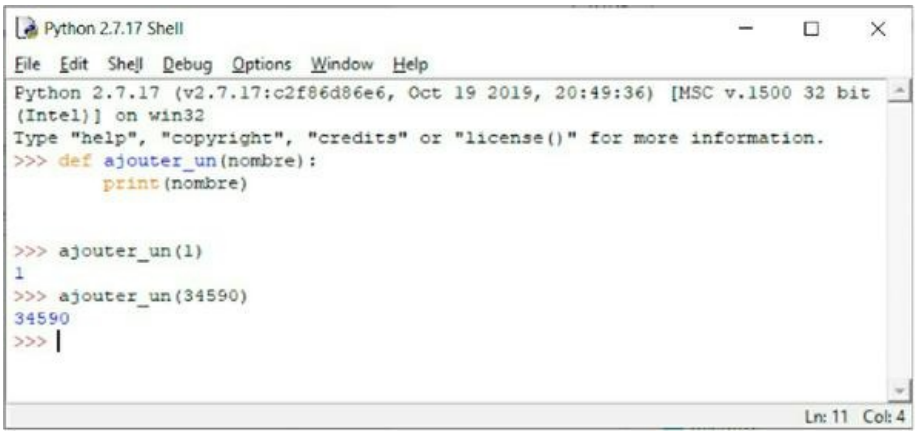


Figure 5.4 : Nous définissons les arguments en indiquant des noms de variables dans l'ordre.

Pour essayer le petit extrait de la Figure 5.4, passe dans la fenêtre de l'interpréteur Shell et saisis les lignes suivantes :

```
def ajouter_un(nombre) :
print(nombre)
```

Voici deux appels pour tester l'argument :

```
>>> ajouter_un(1)
1
>>> ajouter_un(54870)
54870
```

Le nom choisi pour la fonction (`ajouter_un()`) est volontairement étonnant. En effet, pour l'instant, la fonction n'ajoute pas un à la valeur qu'elle reçoit. Nous y reviendrons. Ce que nous voyons, c'est que la fonction récupère bien la valeur qui lui est transmise. Lorsque l'appel est exécuté, cela provoque la création d'une variable locale portant le nom `nombre`, puis la valeur 1 (ou 54870) y est stockée. Dans l'instruction de

la fonction, nous affichons cette valeur. Nous arrivons alors à la fin du bloc de la fonction et reprenons l'exécution juste après l'appel.

```
Les arguments ne sont pas facultatifs. Si tu définis une fonc-
tion pour qu'elle reçoive un argument, tu obtiens une erreur
si tu l'appelles sans lui fournir cet argument :
>>> ajouter_un()

Traceback (most recent call last):
File "<pyshell#8>", line 1, in <module>
ajouter_un()
TypeError: ajouter_un() takes exactly 1 argument (0 given)
```

Le message rappelle que la fonction a besoin de recevoir une valeur au démarrage (un argument), mais qu'elle n'en a pas reçu.

Une fonction peut recevoir plusieurs arguments, mais il faut fournir une valeur pour chacun des arguments lors des appels. L'exemple suivant définit deux arguments nommés A et B. Lors de l'appel à la fonction, nous transmettons les valeurs entières 1 et 2, qui sont stockées dans les deux variables locales A et B dans le même ordre que l'ordre de définition.

```
>>> def afficher_2nombres(a,b):
        print(a,b)

>>> afficher_2nombres(1,2)
(1, 2)
```

Les valeurs que reçoit la fonction au démarrage sont stockées dans les variables dans l'ordre dans lequel elles sont présentées : ce sont des *arguments positionnels*. Dans l'exemple de la Figure 5.5, les valeurs 1 et 2 sont stockées à l'envers, respectivement dans les variables B et A, puisque c'est dans cet ordre que les noms sont stipulés dans la définition.

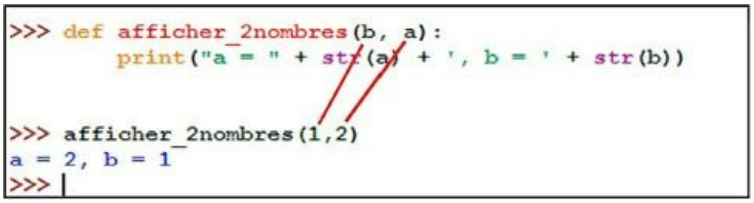


Figure 5.5 : Les valeurs sont stockées dans les variables locales dans l'ordre dans lequel elles sont définies.

Valeur d'argument implicite (argument nommé)

Dans certains cas, un argument peut être défini avec une valeur implicite, ce qui lui permet d'avoir une valeur dans tous les cas, même lorsque tu ne fournis pas de valeur lors de l'appel.

C'est une sorte de système d'appel de fonction à géométrie variable. Bien sûr, le ou les arguments nommés doivent être indiqués après tous les arguments normaux (sinon, Python est perdu).

Pour attribuer une valeur par défaut (implicite) à un argument, il faut affecter cette valeur figée dans la ligne de définition de la fonction.

Dans l'exemple suivant, une variable nommée `aff_oui` reçoit par défaut la valeur True grâce à la mention `aff_oui=True`.

Listing 5.4 : Code source AjouterUn1.py

```
def ajouter_un(a, b, aff_oui=True):
    if aff_oui:
        print("a = " + str(a))
```

```
        print("b = " + str(b))
```

Tu peux saisir ces lignes dans l'interpréteur. Voici le résultat de deux appels :

```
>>> ajouter_un(1, 2)
a = 1
b = 2
>>> ajouter_un(1, 2, False)
>>>
```

Une remarque préalable : dans un test avec `if`, on peut indiquer un nom de variable seul. Cela revient au même que d'écrire une expression de test comme ceci : `if aff_oui == True :` .

La variable locale `aff_oui` est un argument implicite avec la valeur `True`. Lorsque nous appelons la fonction avec deux arguments `(1, 2)`, les deux valeurs sont bien stockées dans les variables `a` et `b`, et la troisième variable reçoit la valeur `True`, à défaut de mieux. Cela provoque l'affichage des deux lignes qui dépendent du test sur `aff_oui`.

Tu peux aller à l'encontre de la valeur par défaut, comme dans le deuxième appel de l'exemple. En écrivant `ajouter_un (1,2, aff_oui=False)`, nous passons outre la valeur par défaut de l'argument. Les deux lignes d'affichage ne sont pas exécutées puisque le test échoue.

Il faut toujours indiquer les arguments nommés après tous les arguments positionnels obligatoires dans la définition, sinon Python s'en rend compte :

```
>>> def ajouter_un(aff_oui=True, a, b):
        pass

SyntaxError: non-default argument follows default argument
>>>
```

Bien que ce ne soit pas fréquent, tu peux définir plusieurs arguments nommés pour une fonction. Chacun obtient sa valeur dans l'ordre dans lequel ils sont indiqués. S'il y a plus d'un argument nommé, un problème peut survenir si tu acceptes d'utiliser la valeur par défaut pour un argument, mais pas pour le suivant. Si tu appelles avec trois arguments une fonction qui doit recevoir deux arguments obligatoires et deux arguments nommés, il manquera une valeur et celle fournie ne sera pas stockée dans la bonne variable.

Python résout le problème en offrant une syntaxe particulière dès qu'il y a plusieurs arguments nommés. Il suffit de préciser le nom de la variable avant la valeur lors de l'appel à la fonction. Dans l'exemple qui suit, nous écrivons `VautMieuxQue = False, aff_oui = False`. En donnant explicitement une valeur, on maintient le contrôle de l'ordre de stockage des valeurs dans les arguments.

Listing 5.5 : Code source AjouterUn2.py (injecté dans le Shell)

```
>>> def ajouter_un(a, b, VautMieuxQue=False, aff_oui=True):
    insert = " "
    if VautMieuxQue:
        insert = ' vaut mieux que '
    if aff_oui:
        print(str(a) + insert + str(b))

>>> ajouter_un(1, 2, True, True)
1 vaut mieux que 2
>>> ajouter_un(1, 2, aff_oui=True, VautMieuxQue=True)
1 vaut mieux que 2
>>> ajouter_un(1, 2, aff_oui=True, VautMieuxQue=False)
1 2
>>>
```

Lors des essais ci-dessus, le deuxième appel spécifie les arguments 2 et 4 dans le mauvais ordre, mais puisque les noms sont spécifiés, tout se passe bien. L'ordre n'a pas d'importance. Ce n'est bien sûr pas le cas pour les arguments positionnels. (Cette technique suppose de connaître le nom des arguments dans la définition de la fonction.) Dans le dernier appel, nous écrivons :

```
ajouter_un(1, 2, aff_oui=True, VautMieuxQue=False)
```

Le quatrième argument était ici inutile puisque c'est sa valeur par défaut.

La fonction rend sa copie

Tu sais maintenant que le niveau principal du programme peut transmettre des données lorsqu'il appelle une fonction. Il suffit de transmettre des arguments ou d'indiquer le nom d'une variable qui n'est pas redéfinie dans la fonction (mais cette dernière méthode n'est à utiliser que pour les constantes).

Ce que tu ne sais pas encore, c'est que la fonction peut à son tour transmettre des données vers le niveau depuis lequel elle a été appelée (soit le niveau principal, soit une fonction d'un niveau supérieur). Ce retour depuis une fonction est fondé sur le mot réservé `return`.

Étudie cet exemple :

```
>>> def ajouter_un(nombre)
return nombre+1
```

J'ai conservé le même nom de fonction `ajouter_un()`, mais la fonction est devenue plus puissante et mérite dorénavant son nom. Elle récupère en entrée la valeur de l'argument `nombre` et renvoie la valeur qui résulte de l'addition de 1 à `nombre` avec `return`.

Tu peux vérifier en faisant un appel à la fonction

```
>>> ajouter_un(4)
5
```

La seule instruction de cette fonction effectue trois actions :

1. **Elle commence par lire la valeur stockée dans `nombre`, c'est-à-dire 4.**
2. **Elle ajoute 1 à cette valeur pour obtenir 5.**
3. **Elle renvoie la valeur à l'endroit du programme où se trouve l'appel à la fonction.**

La fonction termine son travail et l'exécution se poursuit dans le niveau supérieur.

Tu peux récupérer la valeur que renvoie la fonction en la stockant dans une variable. Dans l'exemple suivant, nous recueillons le fruit de l'appel à la fonction dans la variable `valret` :

```
>>> valret = ajouter_un(4)
>>> valret
5
```

Par définition, une fonction renvoie toujours une valeur, même si tu n'utilises pas le mot réservé `return` dans le corps de la fonction. Dans la fonction d'affichage du message de bienvenue (début du chapitre), nous ne connaissions pas encore le mot réservé `return`. Pourtant, elle renvoie quelque chose. Mais quoi ?

```
>>> valret = afficher_salut()
Salut les gens de France et de Navarre !
>>> print(valret)
None
```

Cette fonction ne renvoie aucune valeur, mais la variable `retval` en contient bien une après l'exécution de la fonction : c'est une pseudo-valeur qui s'écrit `None` (rien). C'est un peu comme si tu demandais quelle

était la valeur renvoyée et que l'interpréteur Python te répondait qu'il n'y en avait pas. Répondre cela n'est pas la même chose que ne pas répondre. Tu n'utiliseras pas cette réponse technique dans tes projets, mais tu auras parfois besoin de la reconnaître, notamment face à un souci dans un programme avec une fonction qui devrait renvoyer une valeur mais qui n'en renvoie pas. Dans le Projet 6, je te montrerai un exemple qui utilise cette pseudo-valeur de retour.

Créons un compteur de score

Les pages précédentes nous ont permis d'apprendre deux concepts extrêmement importants :

- l'envoi d'informations en entrée d'une fonction au moyen des arguments pour faire varier son fonctionnement ;
- le renvoi d'information à la fin de l'exécution d'une fonction en direction de la ligne qui a appelé cette fonction, grâce à `return`.

Profitons de ces nouvelles connaissances pour enrichir notre projet de jeu de devinette. Nous allons ajouter de quoi tenir un score et prévoir une fonction pour quitter plus confortablement. En effet, tu ne peux pas demander aux utilisateurs de savoir taper la combinaison Ctrl+C pour quitter.

Nous allons améliorer le projet afin qu'il réalise les opérations suivantes :

- compter le nombre de tours de jeu ;
- compter le nombre d'essais dans chaque tour de jeu puis pour la totalité du jeu (nous pourrons ainsi en déduire une moyenne d'essais par tour).

Pour créer cette nouvelle version du projet, je te conseille de recharger la plus récente version, qui correspond au Listing 5.2, c'est-à-dire au fichier *JeuDevin2.py*.

Pour compter les tours :

1. **Nous ajoutons une variable dans la section principale sous le nom `total_tours`.**
2. **Nous ajoutons des instructions pour augmenter de un (incrémenter) cette variable dans la boucle infinie `while` du niveau principal.**

Pour compter le nombre de saisies dans chaque tour :

1. **Nous ajoutons une variable `total_saisies` dans la fonction `jouer_tour()`.**
2. **Nous faisons renvoyer la valeur stockée dans cette variable vers la boucle qui a appelé la fonction au moyen du mot réservé `return`.**

Voici le code complet de cette nouvelle version du projet. Les commentaires de fin de ligne indiquent les lignes ajoutées ou modifiées.

Listing 5.6 : Code source JeuDevin3.py

```
""" Projet 5 JeuDevin.py  Version 3
"""

import random

# nbr_secret = random.randint(1,100)   #SUPPR
INVITE = 'Propose un nombre : '

def jouer_tour():
""" Choisir un nombre, demander au joueur
de le trouver et reboucler tant qu'il ne l'a pas."""
nbr_secret = random.randint(1,100)
nbr_saisies = 0                    #AJOUT
while True:
```

```
nbr_joueur = raw_input(INVITE)
nbr_saisies = nbr_saisies + 1  #AJOUT
if nbr_secret == int(nbr_joueur):
print('Correct !')
return nbr_saisies         #MODIF
elif nbr_secret > int(nbr_joueur):
print('Trop petit')
else:
print('Trop grand')

### SECTION PRINCIPALE MAIN
total_tours   = 0          #AJOUT
total_saisies = 0          #AJOUT

while True:
total_tours = total_tours + 1                  #AJOUT
print("On passe au tour " + str(total_tours))   #MODIF
# print("Chut ! Le nombre secret ...           #SUPPR
print("En avant pour les devinettes !")

ce_tour = jouer_tour()                          #MODIF

total_saisies = total_saisies + ce_tour         #AJOUT
print("Tu as fait "+str(ce_tour) + " saisies.") #AJOUT
moy = str(total_saisies / float(total_tours))   #AJOUT
print("Ta moyenne de saisies/tour = " + moy)    #MODIF
print("")
```

Dans cette nouvelle version du projet, les lignes ajoutées comportent la mention #AJOUT en fin de ligne, et celles modifiées la mention #MODIF. Il y a même une ou deux lignes devenues inutiles ; elles ont été conservées mais neutralisées par un `#` au début.

Le programme compte dorénavant le nombre de tentatives de deviner la valeur dans la fonction. La partie principale du programme ne peut pas accéder aux variables locales de la fonction de devinette, car elles sont inaccessibles. Le seul moyen de renvoyer le nombre de saisies vers le niveau 1 consiste à utiliser `return` dans la fonction et à récupérer la valeur dans une autre variable portant le nom `ce_tour`.

Nous totalisons le nombre de saisies pour tous les tours en ajoutant la valeur que nous venons de recevoir dans `ce_tour` à la variable `total_saisies`. Nous pouvons ainsi calculer la moyenne des essais par tour. Il suffit de diviser le nombre total de saisies par le nombre total de tours. Tu remarques l'utilisation du convertisseur de type `float()` qui oblige Python à fonctionner avec des valeurs non entières (revois le Projet 3 si nécessaire).

Et si le joueur veut quitter ?

Pour quitter le jeu à tout moment, le joueur ne sera pas étonné d'y arriver en frappant la touche Q. Pour éviter qu'il sorte sans le vouloir, il est conseillé de lui demander confirmation. Nous allons donc ajouter une fonction dans laquelle nous demandons au joueur de refuser la sortie par la touche N. Par défaut, nous allons considérer que c'est ce qu'il veut faire.

Voici comment va être construite la fonction de confirmation :

1. **Nous définissons une constante qui contiendra le message de demande de confirmation.**
2. **Nous définissons une fonction portant le nom `confirmer_ quitter()` et nous créons immédiatement sa doc-chaîne.**
3. **Dans cette fonction de confirmation, nous demandons à l'utilisateur de confirmer par n'importe quelle touche sauf N, ou d'infirmer par N qu'il ne veut pas quitter. Nous nous servons de la fonction de saisie `raw_input()`.**

Si l'utilisateur frappe la touche N, nous renvoyons la valeur False. Dans tous les autres cas, nous renvoyons True. Nous considérons en effet que dans la majorité des cas, il n'a pas frappé la touche Q par erreur.

Voici déjà un aperçu du code source de la nouvelle fonction (ne saisis rien pour l'instant) :

```
QUIT_CONFIRMER = "Es-tu certain de vouloir quitter (O/n) ?"
#NOUVELLE FONCTION POUR CONFIRMER QU'ON VEUT QUITTER
def confirmer_quitter():
""" On sort seulement si saisie de la
lettre n minuscule par renvoi de False. """
confi = raw_input(QUIT_CONFIRMER)
if confi == 'n':
return False
else:
return True
```

Dans la demande de confirmation, la touche saisie par l'utilisateur est stockée dans la variable `confi`. Si elle contient `'n'`, la fonction renvoie la valeur `False`, ce qui doit provoquer la fin du programme. Si la variable contient quoi que ce soit d'autre (l'utilisateur peut même frapper directement la touche Entrée), la fonction renvoie la valeur `True`, ce qui confirme qu'il veut quitter. Par défaut, c'est ce que nous supposons.

Nous pourrions remplacer tout le bloc conditionnel `if... else` par une ligne unique, mais sophistiquée :

```
return confi != 'n'
```

C'est une écriture compacte, mais un peu trop complexe pour le moment. Tu peux la relire plusieurs fois. Elle a exactement le même effet que les quatre lignes du test.

Il reste maintenant à injecter ces nouvelles lignes de code dans le programme :

1. **Nous définissons une constante portant le nom QUIT_TXT qui va contenir `q` (c'est la lettre que l'utilisateur doit saisir pour s'avancer vers la sortie).**
2. **Nous définissons une autre constante portant le nom QUITTER qui correspond à la valeur que la fonction `jouer_tour()` doit renvoyer si l'utilisateur veut quitter. Nous donnons à cette constante la valeur spéciale `-1`.**
3. **Dans la fonction `jouer_tour()`, nous ajoutons un test pour détecter la saisie de la touche Q (donc la lettre `q` en minuscule) au lieu d'un nombre.**
4. **Si cette touche est détectée, nous appelons la fonction `confirmer_quitter()`.**
5. **Si cette fonction renvoie la valeur True, nous quittons réellement. En fait, nous faisons renvoyer la valeur spéciale -1 avec `return QUITTER`.**

S'il n'y a pas eu de volonté de quitter, nous utilisons le nouveau mot réservé `continue`. Il ne peut être utilisé que dans une boucle conditionnelle. Il provoque le retour immédiat au début du bloc pour démarrer un nouveau tour de boucle. Dans certaines boucles, cela peut provoquer l'incrémentation du compteur de tours, s'il y en a un.

Les nouvelles constantes sont placées tout en haut du programme, après la directive `import random` :

```
QUITTER  = -1
QUIT_TXT = 'q'
```

Le code conditionnel dans la fonction `jouer_tour()` doit être placé juste après l'instruction de saisie `raw_input()`. Voici son aspect :

```
# AJOUT BLOC IF POUR SORTIE CONFIRMEE
if nbr_joueur == QUIT_TXT:
if confirmer_quitter():
```

```
return QUITTER
else:
continue  # Tour de boucle suivant
```

Chaque fois que le joueur saisit un nombre, il est comparé à la valeur de la constante `QUIT_TXT`, c'est-à-dire la lettre `q`. Si c'est bien un q, nous appelons la fonction `confirmer_quitter()`. Si le joueur confirme qu'il veut quitter, nous sortons de la fonction `jouer_tour()` en renvoyant la valeur spéciale `- 1` (la constante `QUITTER`). S'il ne veut pas quitter, par exemple parce qu'il a tapé la touche `Q` par erreur, l'instruction `continue` abrège le parcours de la boucle et revient au début pour demander un autre nombre.

Il faut noter que dans le cas où le joueur tape q, mais change d'avis, la variable nbr_saisies ne contient pas un nombre mais la lettre q, ce qui risque d'entraîner une erreur. C'est pour cette raison qu'il faut abandonner la poursuite du tour de boucle en cours et revenir demander un nouveau nombre.

Il nous reste à apporter des aménagements dans la fonction principale. Nous devons tester si c'est la valeur QUITTER qui a été renvoyée dans la variable `ce_tour`. Si c'est bien cette valeur spéciale, il faut faire ceci :

1. **Générer un message de statistiques et l'afficher.**

   Lorsqu'un tour se termine, c'est normalement parce que le joueur a trouvé le nombre. Nous augmentons donc de un le compteur `total_ tours` à chaque retour de la fonction. Mais si le joueur est sorti du tour avant d'avoir trouvé (en demandant de quitter), il faut enlever un au nombre de tours joués. Problème : si le joueur abandonne dès le premier tour, cela rendrait la variable égale à zéro, et entraînerait une erreur de division dans l'instruction de calcul de la moyenne. C'est pourquoi cette instruction est placée dans la branche conditionnelle dans laquelle `total_tours` n'est jamais égal à zéro.

   Nous pourrons ainsi calculer la moyenne du nombre de saisies par tour, puisqu'il suffit de diviser le nombre total de nombres saisis pour tous les tours par le nombre de tours réussis.

   Il faut absolument utiliser une division avec des nombres à virgule flottante pour que la réponse soit correcte. Nous nous servons de `float()` pour convertir au moins un des nombres entiers en nombre décimal.

2. **Pour sortir de la boucle `while`, nous utilisons `break`.**

3. **Tout à la fin du programme, nous affichons des messages de statistiques.**

   Voici la nouvelle section de code source qui devra être insérée juste au retour de la fonction `jouer_tour()`, c'est-à-dire après la ligne `ce_tour = jouer_tour()`.

   ```
   # AJOUT BLOC IF  POUR TESTER SI QUITTER
   if ce_tour == QUITTER:
   total_tours = total_tours - 1
   if total_tours == 0:
   msg_stat = "1er tour pas fini ! " +\
   "Tu veux recommencer ?"
   else:
   moy = str(total_saisies / float(total_tours))
   msg_stat = "Tu as fait " + str(total_tours) +\
   " tours. Moyenne de " + str(moy)
   break
   ```

4. **Pour finir proprement l'exécution du programme, nous ajoutons deux lignes d'affichage de messages tout à la fin, donc en dehors de la boucle principale `while` :**

   ```
   print(QUIT_MSG)
   print(msg_stat)
   ```

Après toutes ces explications, tu as bien mérité de savoir pourquoi j'ai choisi l'étrange valeur `-1` pour la constante `QUITTER`. À ton avis, comment peut-on depuis le niveau du programme principal distinguer un

nombre qui représente le nombre de saisies du joueur au cours du dernier tour et un nombre spécial qui signifie que le joueur veut quitter ?

Si on renvoie une lettre pour dire que l'on veut quitter, cela va poser problème parce que le format des instructions de test n'est pas le même selon qu'il s'agisse d'une valeur numérique ou d'une valeur chaîne. Si tu décides par exemple de renvoyer la chaîne `"quitter"` pour dire que le joueur veut quitter, c'est cette chaîne qui va être stockée dans la variable de type numérique `ce_tour`. Lorsque le programme va tester la variable, cela va provoquer une erreur qui va entraîner l'arrêt du programme. La sanction serait la même pour tester un nombre là où on attend une chaîne.

Tu découvriras plus tard des techniques qui permettent de régler ce genre de soucis. Pour l'instant, il suffit de toujours faire renvoyer un nombre, mais un nombre qui ne peut jamais être celui du nombre de saisies. J'ai choisi la valeur `-1`, car le joueur ne peut jamais avoir fait moins de zéro saisie. Tu peux également choisir un nombre du style 100 000, parce que l'on imagine qu'un joueur n'aura jamais fait cent mille saisies (à moins de jouer pendant deux semaines sans manger et sans boire). Je trouve que la valeur `-1` est plus raisonnable.

Au niveau principal, à chaque retour de l'appel à la fonction `jouer_tour()`, nous testons si la valeur renvoyée est égale à QUITTER. Si c'est le cas, nous préparons quelques messages qui sont affichés juste avant la fin et nous sortons de la boucle principale `while` par `break`. Nous affichons enfin les messages et le programme est terminé. Quelle aventure !

SORTIR D'UN PROGRAMME PYTHON

Pour l'instant, nous sommes sortis de nos programmes Python en arrivant par défaut, à la dernière instruction.

D'une certaine manière, l'interpréteur tombe du programme par le bas. Mais souvent, tu as besoin de sortir volontairement. Python propose deux techniques :

L'instruction standard nommée `exit` qui sert à sortir du mode interactif de l'interpréteur. Tu ne peux pas t'en servir dans un programme enregistré dans un fichier parce que cette instruction n'assure pas un bon nettoyage mémoire.

La fonction standard nommée `exit()`. Elle fait partie du module nommé `sys` de la librairie standard. Pour pouvoir appeler cette fonction, il faut donc demander l'importation du module `sys` :

```
# Test de sys.exit()
import sys
sys.exit()
print('Cette ligne ne devrait
jamais se lire.')
```

Si tu lances ce programme depuis une fenêtre de terminal Python (command line), il doit démarrer puis se terminer sans afficher aucun message. L'interpréteur n'arrive jamais jusqu'à la ligne `print`. En revanche, cela ne fonctionne pas ainsi si tu lances le programme depuis la fenêtre de l'éditeur IDLE (à cause de la manière dont fonctionne IDLE).

Le projet final

Voici le code source complet du programme de devinette fondé sur des fonctions.

Listing 5.7 : Code source JeuDevin4.py (version finale)

```
""" Projet 5 JeuDevin.py  Version 4 (finale)
"""
import random

INVITE = 'Propose un nombre : '

# NOUVELLES CONSTANTES
```

```
QUITTER  = -1
QUIT_TXT = 'q'
QUIT_MSG = 'Merci pour tout !'
QUIT_CONFIRMER = "Es-tu certain de vouloir quitter (O/n) ?"

#NOUVELLE FONCTION POUR CONFIRMER QU'ON VEUT QUITTER
def confirmer_quitter():
    """ On sort seulement si saisie de la
    lettre n minuscule par renvoi de False. """
    confi = raw_input(QUIT_CONFIRMER)
    if confi == 'n':
        return False
    else:
        return True

def jouer_tour():
    """ Choisir un nombre, demander au joueur
    de le trouver et reboucler tant qu'il ne l'a pas."""
    nbr_secret = random.randint(1,100)
    nbr_saisies = 0                    #AJOUT
    while True:
        nbr_joueur = raw_input(INVITE)
        # AJOUT BLOC IF POUR SORTIE CONFIRMEE
        if nbr_joueur == QUIT_TXT:     #AJOUT
            if confirmer_quitter():     #AJOUT

                return QUITTER         #AJOUT
            else:                      #AJOUT
                continue  # Tour de boucle suivant #AJOUT
        nbr_saisies = nbr_saisies + 1
        if nbr_secret == int(nbr_joueur):
            print('Correct !')
            return nbr_saisies
        elif nbr_secret > int(nbr_joueur):
            print('Trop petit')

        else:
            print('Trop grand')

### SECTION PRINCIPALE MAIN
total_tours   = 0          #AJOUT
total_saisies = 0          #AJOUT
msg_stat = 0
while True:
    total_tours = total_tours + 1
    print("On passe au tour " + str(total_tours))
    print("En avant pour les devinettes !")

    ce_tour = jouer_tour()

    # AJOUT BLOC IF  POUR TESTER SI QUITTER
    if ce_tour == QUITTER:                              #AJOUT
        total_tours = total_tours - 1                   #AJOUT
        if total_tours == 0:                            #AJOUT
            msg_stat = "1er tour pas fini ! " +\

                    "Tu veux recommencer ?"             #AJOUT
        else:                                           #AJOUT
            moy = str(total_saisies / float(total_tours)) #A
            msg_stat = "Tu as fait " + str(total_tours) +\
                    " tours. Moyenne de " + str(moy)    #AJOUT

        break                                           #AJOUT
    total_saisies = total_saisies + ce_tour
```

```
    print("Tu as fait "+str(ce_tour) + " saisies.")
    moy = str(total_saisies / float(total_tours))
    print("Ta moyenne de saisies/tour = " + moy)
    print("")

#AJOUT MESSAGE DE SORTIE
print(QUIT_MSG)
print(msg_stat)
```

Récapitulons

Nous en avons vu, des choses, au cours de ce projet ! Rien qu'à lire la liste des principaux points abordés, je suis émerveillé et même exténué :

- Nous avons découvert les trois nouveaux mots réservés `def`, `return` et `continue` (il n'en reste que neuf à découvrir).
- Nous avons plongé dans le très important concept de fonctions, et vu comment les définir et les utiliser.
- Nous avons appris les règles à appliquer pour donner des noms aux fonctions.
- Nous avons ajouté une doc-chaîne au début d'une fonction et vu l'intérêt de ces commentaires spéciaux.
- Nous avons appris à appeler une fonction.
- Nous avons créé une amorce de fonction en vue de définir les détails plus tard.
- Nous avons vu l'importance de la portée des variables selon qu'elles sont définies à l'intérieur d'une fonction ou au niveau général de la fonction `main`.
- Nous avons appris à transmettre des données à une fonction au moyen de ses arguments.
- Nous avons vu comment donner une valeur par défaut, implicite, à un argument et nous avons appris à distinguer les arguments positionnels des arguments implicites.
- Nous avons appris à faire renvoyer une valeur depuis une fonction au moyen du mot réservé `return`.
- Nous savons quitter un programme.
- Nous avons créé une fonction réutilisable pour demander à l'utilisateur de confirmer qu'il veut vraiment quitter le programme.

Semaine 3

Jouons avec les mots

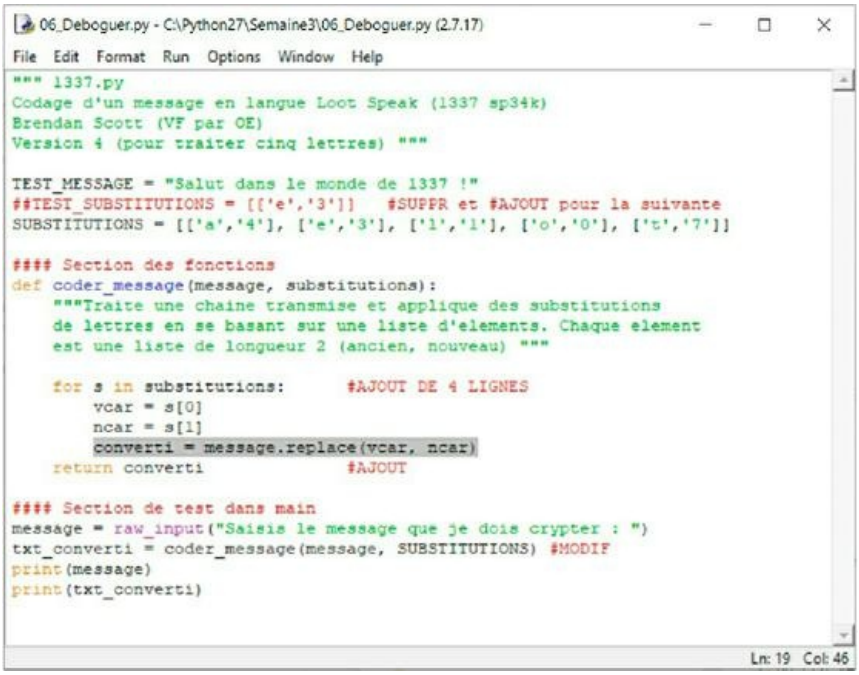
AU MENU DE CETTE SEMAINE

- Projet 6 : **Un message pour les h4ck3rs**
- Projet 7 : **Cryptopy !**
- Projet 8 : **Les phrases en folie**

Projet 6

Un message pour les h4ck3rs

Es-tu prêt à créer un programme qui transforme un message du français vers la langue des fondus de programmation que sont les hackers ? C'est l'écriture *leet* (ou *1337*). Tu as peut-être déjà rencontré ce genre d'écriture sur les sites des programmeurs qui vivent du côté obscur du Web. Le principe de ce codage est de remplacer certaines lettres de l'alphabet par des chiffres. La lettre `a` minuscule est par exemple remplacée par le chiffre `4` et la lettre `e` par le chiffre `3` : « cette lettre » s'écrira ainsi « c3773 1377r3 ».



Pour réaliser ce projet d'agrément, nous allons aller plus loin dans les chaînes de caractères, mais aussi découvrir les listes, les objets et la technique d'introspection de Python. Le code source du projet ne sera pas très complexe, mais les idées qu'il va illustrer sont d'une grande importance. Ce sera une autre occasion de voir à quel point Python est puissant. Il permet de faire des choses étonnantes très simplement. Tout ce que tu vas découvrir au cours de ce projet te sera utile au quotidien dans la suite de tes aventures pythonesques !

Garçon, il y a un objet dans ma chaîne !

Au départ, je voulais donner comme titre à cette section « Explosion cérébrale », parce qu'elle regorge d'informations importantes. (Mais, comme me l'a rappelé mon éditeur, on n'a jamais vu exploser un cerveau simplement du fait d'apprendre trop de choses en même temps. Et je pense donc que lire la suite n'aura aucun effet néfaste sur ta santé !) Les chaînes de caractères, tu connais déjà : tu as utilisé la première chaîne littérale dès le Projet 2. Tu te souviens peut-être que tu avais donné un nom à la chaîne « Salut les Terriens », et qu'en donnant ce nom, tu avais créé une variable.

Tu peux redémarrer l'atelier IDLE pour saisir cette définition de variable :

```
>>> messtest = 'Salut les Martiens !'
```

Si je ne tenais pas compte de l'avis de mon éditeur, je te conseillerais de bien t'accrocher à ta chaise avant de lire la suite. On ne sait jamais. Une fois que tu es prêt, saisis la commande suivante :

```
>>> dir(messtest)
```

Dans cette ligne, je te demande d'utiliser l'instruction standard de Python nommée `dir()`. C'est elle qui donne à Python ses superpouvoirs d'introspection. Elle permet au programme de te donner des informations au sujet de lui-même. Le fait qu'il y ait un jeu de parenthèses te laisse deviner que c'est bien une fonction et tu peux fournir un argument entre ces parenthèses. La réponse de la fonction est une liste d'objets techniques qui sont en relation avec l'élément indiqué. C'est clair comme du jus de chaussette, non ?

Voici le genre de liste que produit la fonction `dir()` :

```
>>> dir(messtest)
['__add__', '__class__', '__contains__', '__delattr__',
'__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__getnewargs__',
'__getslice__', '__gt__', '__hash__', '__init__', '__le__',
'__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '_formatter_field_name_split',
'_formatter_parser', 'capitalize', 'center', 'count',
'decode', 'encode', 'endswith', 'expandtabs', 'find',
'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
```

Ignore les premières lignes contenant des noms bizarres commençant par `__`. La liste contient ensuite des noms de fonctions qui sont d'office associés à la variable (ici, `messtest`). Autrement dit, lorsque tu définis la variable en choisissant son nom et en y stockant le contenu de la chaîne littérale `'Salut les Martiens ! '`, Python ne se contente pas de stocker les caractères quelque part en mémoire. Il fait bien plus de choses. Voici toutes les opérations qui sont réalisées en coulisses :

- Création d'une structure de donnée prototype.
- Mise en correspondance de cette structure avec le nom de la variable (`messtest`) et affectation de la chaîne littérale à ce nom.
- Au moment de cette affectation de valeur, Python a détecté qu'elle était de type chaîne (type `str`). Il a alors pu personnaliser la structure de données pour qu'elle soit adaptée à ce type chaîne.
- Python a ensuite raccordé techniquement toute une série de fonctions prédéfinies et de littéraux utilisables avec une variable de ce type chaîne.
- Toute cette suite de fonctions et de littéraux a été mise en place dans la structure de donnée prototype du départ.

Qui aurait pu croire que Python aurait tant de choses à faire juste avec une petite ligne de création de variable ? En fait, notre variable n'est pas seulement un symbole pour l'adresse mémoire à laquelle est stocké le début du texte. En réalité, `messtest` est un objet. Objet est le nom utilisé dans Python pour les structures de données prototypes dont je viens de parler. À vrai dire, tout dans Python est un objet.

LES TYPES D'OBJETS ET LES ID

J'ai parlé plus haut de chaînes et de listes, mais c'est une manière rapide de parler des objets du type chaîne (strings) et des objets du type liste, respectivement. Python dispose d'une instruction permettant de savoir quel est le type d'un objet. Sans surprise, son nom est `type()` :

```
>>>type("Je t'aime tant")
<type'str'>
>>>type([])   #U nelistevide
<type 'list'>
Tout objet est toujours d'un certain
type,ettousles objets possèdentun
```

identifiant ID qui est tout simplement l'adresse en mémoire où est stocké l'objet par Python. Pour connaître l'ID d'un objet (pas utile tous les jours), il suffit d'utiliser la fonction standard `id()` :

```
>>>id("Je t'aime aussi")
42088672
```

Le type d'un objet ne change jamais, mais son adresse mémoire varie d'une exécution à la suivante en fonction du prochain emplacement mémoire libre que trouve Python quand il l'implante.

Chaque objet est donc associé à toute une série de fonctions et de littéraux, qui sont les *attributs* de cet objet. Certains des attributs sont des fonctions portant dans ce cas le nom de *méthodes*. Pour simplifier, nous les appellerons malgré tout des fonctions, mais on parle aussi de méthodes de modules.

Si tu étais un objet (ce n'est pas le cas, nous sommes d'accord), tu posséderais des variables comme par exemple `moi.taille` et `moi.poids` et des méthodes comme `moi.laver_les_dents()` et `moi.aller_au_lit()`. Ce sont des fonctions que pourraient déclencher tes parents. Les deux attributs `taille` et `poids` mémorisent des informations à ton sujet alors que les deux méthodes lui font réaliser des actions.

Je rappelle que tout est un objet dans le langage Python.

Un point pour accéder aux attributs

Maintenant que tu sais qu'une simple variable comme `messtest` possède toute une panoplie d'attributs, il reste à savoir comment les utiliser.

Comment par exemple déclencher la méthode `upper()` qui est un attribut de `messtest` d'après la liste affichée par `dir()` ? En fait, tu connais déjà la réponse, puisque nous avons généré un nombre au hasard avec la fonction `randint()` du module `random`. Dans le Projet 3, nous avions écrit `random.randint()`. L'objet était le module `random` et sa fonction, pardon sa méthode, était `randint()`. Nous avions qualifié le nom de la fonction en indiquant le nom du module et un point séparateur.

Voyons maintenant ce qu'affiche le système d'aide pour la méthode `upper()` de notre variable. Même si tu ne lis pas l'anglais, tu peux deviner l'essentiel de ce qu'il faut en retenir (la fin est traduite ici) :

```
>>> help(messtest.upper) # Tu as vu le point ?
Help on built-in function upper:
upper(...)
S.upper() -> string
Renvoie une copie de la chaine S apres passage de toutes
les lettres en capitales.
```

Ce qui s'affiche correspond en fait à la doc-chaîne du début du code source de la fonction. Cette technique de documentation est vraiment pratique. Tu n'as rien fait d'autre que créer une variable et lui affecter une valeur. Automatiquement, tu disposes d'un objet doté de toute une série de méthodes, et tu n'as même pas besoin d'aller fouiller dans des livres de référence ou sur le Web pour savoir ce que fait chacune des fonctions. Il suffit d'interroger celle qui t'intéresse pour apprendre l'essentiel de ce qu'elle peut t'apporter.

Essayons ceci dans l'interpréteur :

```
>>> messtest.upper
<built-in method upper of str object at 0x7f3d0803e260>
```

Oui, oui, cela confirme qu'il s'agit d'une fonction/méthode, mais toi comme moi le savions déjà. Il faut ajouter le jeu de parenthèses ! En indiquant le nom sans les parenthèses, tu peux juste savoir s'il s'agit d'une fonction ou d'autre chose.

Lorsqu'un attribut est une méthode, le nom doit toujours se terminer par un jeu de parenthèses, même vide.

Tu peux remarquer que l'affichage a bien confirmé qu'il s'agit d'une méthode portant le nom `upper()` et appartenant à un objet du type `str`, l'objet se trouvant à l'adresse indiquée à la fin du message. Appelons cette méthode :

```
>>> messtest.upper()
'SALUT LES MARTIENS !'
```

La fonction a fait son travail comme prévu : elle a créé une nouvelle chaîne littérale écrite entièrement en lettres capitales à partir de celle que contient la variable `messtest`. On peut vérifier que le contenu de mon message n'a pas été modifié :

```
>>> messtest
'Salut les Martiens !'
```

Ne confonds pas capitales et majuscules. En typographie, la Majuscule désigne la première lettre d'un mot, dont elle seule est écrite en CAPITALE.

Des méthodes privées ?

Dans la liste affichée par l'instruction `dir()`, tu te souviens de la série d'attributs dont les noms commencent par deux caractères de soulignement (`__`). Les méthodes dont le nom commence ainsi sont des *méthodes privées*, les autres étant des méthodes publiques.

Quand tu seras devenu un maître en Python, tu pourras t'intéresser aux méthodes privées des objets, et tu pourras faire des choses formidables. Nous nous y intéresserons un peu dans le projet de carnet d'adresses vers la fin du livre. Pour le moment, à la trappe les méthodes privées ! Concentrons-nous sur les autres attributs de notre variable, et notamment ses méthodes publiques.

Petit rappel : on écrit `nom_objet.nom_attribut` et si c'est une méthode, c'est `nom_objet.nom_methode(argus)`. Les parenthèses sont destinées à recevoir les arguments d'entrée de la méthode.

LE DOUBLE-SOULIGNÉ (DOUBLESOUL)

Les deux caractères de soulignement (touche du 8) placés au début et à la fin des noms des méthodes privées, comme `__init__`, sont souvent appelés *doublesoul* (double soulignement). Les Anglais parlent de *dunder* (Double UNDERscore). C'est plus rapide que de dire « la méthode double caractère de soulignement init double caractère de soulignement ». Tu peux dire « doublesoul init », on te comprendra.

La liste entre en lice []

Tu as certainement remarqué que la longue réponse affichée par l'instruction `dir(messtest)` est encadrée par des crochets `[ ]`. Tu les as déjà rencontrés dans le premier projet lorsque nous avons parlé de l'instruction `range()` pour travailler sur une plage de valeurs. Je t'avais dit en ce temps-là que nous en reparlerions plus tard. Plus tard, c'est maintenant.

Les crochets indiquent que l'objet est du type liste. Une liste est une sorte de conteneur dans lequel on peut stocker d'autres objets dans un certain ordre. Chacun des objets que contient la liste est un élément.

Balayons une liste

Nous nous sommes servis dans le Projet 2 de l'instruction `range()` pour créer une liste puis circuler dans celle-ci en passant d'un élément au suivant, c'est-à-dire faire une itération. Rappelons le code source :

```
>>> range(3)
[0, 1, 2]
>>> for i in range(3):
print(i)
0
1
2
```

L'écriture `range(3)` produit la liste `[0,1,2]`. L'instruction `for i in range(3)` correspond donc en réalité à `for i in [0,1,2]`. Elle affecte successivement à la variable `i` la valeur de chaque élément de la liste, un élément par tour.

Les listes ne sont pas limitées aux valeurs numériques entières. Tu peux très bien écrire `for i in Y` si Y est un objet qui en contient d'autres. Par exemple, l'instruction `dir()` renvoie une liste de chaînes, ce que tu confirmes par le fait que chacune des réponses est délimitée par des apostrophes.

Tu peux donc tout à fait faire afficher la liste des attributs de la variable `messtest` de façon plus lisible, avec une chaîne par ligne :

```
>>> for i in dir(messtest):
print i
__add__
__class__
__contains__

# SUITE TRONQUEE
```

La dernière ligne n'est pas affichée ; c'est seulement la manière dont je montre que la liste est en réalité plus longue.

Essaie le programme pour voir apparaître la liste complète. Voilà encore une preuve de la puissance de Python. À partir du moment où tu disposes d'une liste, tu peux la balayer élément par élément avec une seule instruction : `for element in liste :` (ne pas oublier le signe deux-points final que `for` rend obligatoire).

Tu peux stocker une liste aussi aisément qu'une chaîne littérale :

```
>>> attrobj_str = dir(messtest)
>>> attrobj_str
['__add__', '__class__', '__contains__', '__delattr__',
'__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__getnewargs__',
'__getslice__', '__gt__', '__hash__', '__init__', '__le__',
'__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__',
```

```
'__rmod__', '__rmul__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '_formatter_field_name_split',
'_formatter_parser', 'capitalize', 'center', 'count',
'decode', 'encode', 'endswith', 'expandtabs', 'find',
'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
```

Puisque, dans Python, tout est un objet (je crois l'avoir déjà dit), tu peux même appliquer l'instruction `dir()` à cette nouvelle liste :

```
>>> dir(attrobj_str)
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__delslice__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__',
'__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__',
'__init__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__','' '__new__', '__reduce__', '__reduce_
ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__setslice__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
```

Cela dit, les deux listes ne sont pas identiques, même si le contenu est le même. La première est la liste des attributs de l'objet `messtest` qui est de type chaîne. La nouvelle liste est celle des attributs de la variable `attrobj_str`. Les types des objets ne sont pas les mêmes. Le type du premier objet est `str`, alors que celui du second objet est `list`. Nous verrons ce qui distingue ces types quand le besoin se fera sentir. Pour l'instant, tu peux te sentir rassasié après avoir ingurgité le mégaconcept fondamental d'objet et les types d'objets.

Créons notre propre liste

Lorsqu'il s'agit de créer une liste, tu peux la créer vide ou la peupler avec des éléments dès le départ. Parfois, il sera utile de disposer de la liste vide au départ, parfois non. Rien n'empêche d'ajouter et d'enlever des éléments plus tard. Voici la procédure générale pour créer une liste remplie d'éléments :

1. **Pour débuter la définition de la liste, indique son nom puis l'opérateur d'affectation `=` puis un crochet ouvrant `[`.**
2. **Indique la valeur des éléments les uns après les autres en les séparant par une virgule. Chaque élément peut être une valeur littérale, le nom d'une variable ou celui de n'importe quel autre objet Python.**
3. **Termine la définition de la liste avec un crochet fermant `]`.**

Si les nombres entiers étaient des bonbons, l'instruction suivante créerait une liste en contenant trois (on peut ajouter des espaces pour que ce soit plus lisible) :

```
>>> liste_bocal = [0, 1, 2]
```

Rien n'empêche de créer la liste sans contenu initial, ce qui équivaut à ignorer l'étape 2 ci-dessus :

```
>>> liste_bocal2 = []
```

Pour ajouter un élément, il suffit d'utiliser la méthode de liste nommée `append()`. Le nouvel élément est toujours ajouté en fin de liste :

```
>>> liste_bocal2.append('Mon truc en plumes')
```

```
>>> liste_bocal2
['Mon truc en plumes']
```

Rien n'empêche de mélanger des objets de types différents dans la même liste. Elle ne se plaindra pas. L'objet suivant contient des valeurs numériques entières, mais elle accepte sans broncher d'accueillir une valeur de type chaîne à la fin :

```
>>> liste_bocal = [12, 17]
>>> liste_bocal.append('Poussez-vous, les nombres')
>>> liste_bocal
[12, 17, 'Poussez-vous, les nombres']
```

La méthode `append()` ajoute toujours l'élément à la fin de la liste. La possibilité de mélanger des objets de types différents dans la même liste n'est pas possible dans tous les info-langages. En théorie, on peut même ajouter la liste à elle-même, mais ne t'aventure pas dans ces zones marécageuses !

ÉLÉMENT OU ARTICLE ?

Certains désignent les membres d'une liste sous le nom d'*élément*, et d'autres utilisent le mot *article*. Personnellement, je préfère conserver le mot élément pour le contenu d'une liste.

Le mot article, je le réserve pour des choses d'un autre type de structure de données, comme nous le verrons dans le prochain projet sur le cryptage.

Créons une liste active

Il est possible de faire générer les éléments d'une nouvelle liste à partir d'une liste existante en y appliquant une formule. Python dispose d'une technique rapide pour réaliser cela : la liste par compréhension. Cette technique est possible avec tous les *itérateurs* (pas de panique, nous y reviendrons plus loin), mais restons concentrés sur les listes.

Voici la procédure générale pour créer une liste par compréhension :

1. **Nous définissons d'abord une variable qui contiendra notre liste, puis nous commençons par le crochet ouvrant `[` de début de liste.**
2. **Nous choisissons une ou plusieurs variables temporaires.**
3. **Nous écrivons une formule qui manipule ces variables temporaires.**
4. **À la suite de la formule, nous ajoutons une instruction de boucle `for` qui va s'appliquer à chacune des variables temporaires.**
5. **Nous terminons par le crochet fermant `]` de fin de définition de liste.**

Voici par exemple comment créer une liste par compréhension pour faire générer les dix premiers nombres pairs à partir de zéro :

```
[2 * x for x in range(10)]
```

La Figure 6.1 montre cette création et son résultat, obtenu grâce à la fonction `list()` :

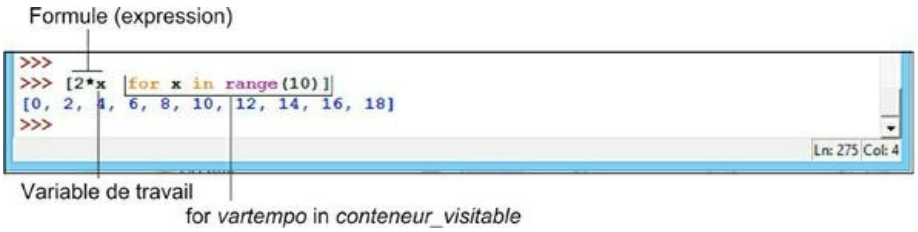


Figure 6.1 : Une liste par compréhension.

Voyons calmement les quatre étapes du traitement de Python pour créer cette liste par compréhension :

> » Python implante la liste vide.
>
> » Il traite ensuite chacun des éléments de la liste générée par `range(10)` et stocke la valeur dans une variable temporaire portant le nom `x`. L'instruction progresse donc dans la liste de nombres 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Je rappelle que par défaut, `range` commence à zéro.
>
> » Pour chaque élément, Python applique la formule qui est ici `2*x` à la valeur trouvée dans `x` puis ajoute le résultat à la nouvelle liste.
>
> » Le processus se termine lorsque tous les éléments générés ont été traités.

Cela constitue une technique très simple et très puissante pour créer une liste. Tu peux même ajouter une condition, pour par exemple filtrer la liste d'éléments générés. L'exemple suivant ajoute une condition pour ne garder que les valeurs paires. Le résultat est strictement le même que le précédent exemple :

```
>>> [x for x in range(10) if x%2 == 0]
[0, 2, 4, 6, 8]
```

La condition s'écrit `if x % 2 == 0`, ce qui oblige le nombre à être pair. Nous utilisons ici l'opérateur modulo qui permet de connaître le reste valeur située à gauche par la valeur située à droite de l'opérateur. Quand on divise des nombres par deux, seuls ceux qui sont pairs donnent un reste égal à zéro.

La condition est donc vraie pour les nombres pairs. Seulement dans ce cas, le nouvel élément contenu dans `x` est ajouté à la liste. Pour les nombres impairs, la condition est fausse et le nombre est ignoré.

Élément, es-tu là ?

Pour savoir si une liste contient un certain élément, il suffit d'utiliser le mot réservé `in` :

```
>>> 0 in [0,1]
True
>>> 72 in [0,1]
False
```

Cette instruction de test renvoie la valeur True (vrai) si l'élément est trouvé. Dans l'exemple, l'élément 0 fait bien partie de la liste [0,1], mais pas le nombre 72. On peut inverser la logique en utilisant le mot réservé `not` avant `in` :

```
>>> 0 not in [0,1]
False
>>> 72 not in [0,1]
True
```

Analysons notre traducteur de leet speak

Notre projet de conversion en écriture 1337 va réunir trois blocs fonctionnels.

1. **Nous devons demander à l'utilisateur de saisir le message qu'il veut faire traduire. Cela ne posera aucun problème en utilisant l'instruction `raw_input()`.**
2. **Nous devons ensuite scruter chacune des lettres du message. Si c'est une des lettres que nous voulons transformer dans son équivalent en langage leet, nous faisons la substitution.**
3. **Nous construisons au fur et à mesure le nouveau message puis nous l'affichons.**

Préparons le fichier de projet

Commençons par créer l'ossature du programme. Nous remplirons les détails plus tard.

ALERTE : MÉTHODE DE LISTE TROMPEUSE !

Il faut se méfier des méthodes de listes, car certaines, pas toutes, modifient la liste originale, au lieu de créer une liste après traitement. À la différence de la méthode `append()` que nous avons exploitée avec `messtest`, certaines méthodes ne renvoient pas une nouvelle chaîne, mais modifient la chaîne à laquelle on les applique.

Ce comportement a deux conséquences. Tout d'abord, la liste de départ est modifiée, ce qui peut étonner. Ensuite, la méthode ne renvoie pas une vraie valeur. Si tu tentes d'affecter la valeur renvoyée par ce genre de méthode, la variable réceptrice va contenir la pseudo-valeur `None` (rien). Si tu n'es pas prévenu, tu vas croire que la variable contient la liste après traitement. J'ai moi-même été piégé plus d'une fois. Voici par exemple une de ces méthodes vicieuses pour le type chaîne, la méthode `reverse()` qui inverse l'ordre des caractères :

```
>>> maliste = range(10)
>>>  maliste
[0, 1, 2, 3, 4, 5, 6, 7, 8,
9]
>>> liste_inverse = maliste.
reverse()
>>> # reverse() ne renvoie
pas la chaîne !
>>> print(liste_inverse)
None
>>> maliste     # La liste de
départ est changée !
[9, 8, 7, 6, 5, 4, 3, 2, 1,
0]
```

S'il s'agissait d'une fonction normale, elle renverrait la valeur, et tu pourrais t'attendre à ce que la nouvelle variable `liste_inverse` contienne le résultat du travail de la méthode. Ce n'est pas le cas ici, et la méthode `reverse()` renvoie cette satanée valeur `None`.

1. **Commence par créer un fichier dans lequel tu vas stocker le code source. Donne-lui le nom** *1337.py*.

   N'oublie pas de vérifier qu'il y a bien l'extension de nom `.py`. J'ai expliqué dans le Projet 4 comment créer un fichier de code source.

2. **Pour prendre une saine habitude, commence par écrire une doc-chaîne tout au début du fichier pour expliquer à quoi il sert.**

   Tu vas créer ainsi la doc-chaîne du module.

3. **Tu définis ensuite une constante portant le nom `TEST_MESSAGE` et tu lui affectes une chaîne comme message de test.**

4. **Tu vas créer une autre constante sous le nom `TEST_SUBSTITTIONS`. Tu lui fournis comme valeur une liste ne contenant pour le moment qu'un élément : `['e', '3']`.**

Méfie-toi, il y a une astuce ici !

Voici à quoi tu devrais avoir abouti :

```
""" 1337.py
Codage d'un message en langue Leet Speak (1337 sp34k)
Brendan Scott (VF par OE)
Version 1 (amorce de fonction) """

TEST_MESSAGE = "Bienvenue dans le monde de 1337 !"
TEST_SUBSTITUTIONS = [['e','3']]
```

Observe bien la dernière ligne ci-dessus. Il y a deux crochets au début et deux à la fin de la liste. Lorsque j'ai dit qu'il n'y avait qu'un seul élément, c'était un indice. Pour créer une liste avec un seul élément, il faut

rajouter une paire de crochets autour du seul élément, comme ceci : crochet ouvrant de la liste crochet ouvrant du ou des éléments. Dans son état actuel, le programme ne fait rien d'utile, mais nous allons bientôt l'enrichir.

Créons l'amorce de fonction

Nous allons maintenant créer l'amorce de la fonction qui va faire le traitement de conversion en langage 1337. Nous allons :

1. **Insérer une ligne de commentaires avec un dièse pour délimiter le début de la section des fonctions.**
2. **Choisir un nom approprié pour la nouvelle fonction et insérer l'amorce de fonction dans la nouvelle section.**
3. **Trouver des noms de variables qui vont servir dans la fonction.**

   La fonction a besoin de récupérer la chaîne du message qu'elle doit encoder et la liste des substitutions à appliquer. Donc deux variables.
4. **Sous la ligne de tête de fonction, ajouter la doc-chaîne qui va bien.**
5. **Insérer une ligne de commentaire par dièse pour marquer la section principale des tests du programme.**
6. **Dans cette section de tests, ajouter un appel à la fonction dont nous avons défini l'amorce en nous servant des variables de test définies. Récupérer la valeur que renvoie la fonction dans une autre variable.**
7. **Afficher le contenu de cette variable.**

Voici à quoi je suis arrivé :

Listing 6.1 : Extrait de 1337.py en version 1

```
#### Section des fonctions
def coder_message(message, substitutions):
"""Traite une chaine transmise et applique des
substitutions
de lettres d'apres une liste d'elements. Chaque element
est une liste de longueur 2 (ancien, nouveau)"""

#### Section de test dans main
txt_converti = coder_message(TEST_MESSAGE, TEST_SUBSTITUTIONS)
print(txt_converti)
```

Testons l'amorce

Il ne reste plus qu'à enregistrer le fichier et à en lancer l'exécution avec F5 (ou Run/Run module). Tu ne devrais pas voir apparaître d'erreur. Voici ce qui devrait s'afficher dans la fenêtre de l'interpréteur Shell :

```
>>> ============================ RESTART ======================
>>>
None
>>>
```

Il ne se passe pas grand-chose, n'est-ce pas ? C'est parce que la fonction `coder_message()` n'utilise pas pour l'instant le mot réservé `return`. La valeur que possède la variable dans le programme principal est `None`. Logique. Pour l'instant, nous avons mis en place l'ossature dynamique minimale qui va vers la fonction et qui en revient.

Nous aurions pu décider de faire renvoyer tel quel par la fonction le message reçu, mais cela n'aurait pas été une bonne idée. En effet, rien n'aurait permis de savoir que la fonction n'a rien codé. Nous aurions pu

aussi faire renvoyer un message de débogage temporaire du type `"Fonction en travaux"` ou renvoyer `None`. Pour l'instant, je préfère ne rien renvoyer. Je trouve que c'est plus simple, mais à chacun ses goûts. Et tout dépend de la situation.

Automatisons les substitutions de lettres

Le prochain objectif va être d'ajouter des instructions dans notre nouvelle fonction afin qu'elle remplace certaines lettres du message. Pour que le monde « leet » soit transformé en « 1337 », il suffit de remplacer les caractères un à un. Il se trouve que les objets du type chaîne (type `str`) disposent dès le départ d'une méthode portant le nom `replace()`. Pour en savoir plus à son sujet, il suffit d'appeler l'aide pour extraire sa doc-chaîne :

```
>>> msg = 'Une bouteille a la mer !'
>>> help(msg.replace)
Help on built-in function replace:

replace(...)
S.replace(old, new[, count]) -> string

Renvoie une copie de la chaîne S dans laquelle tous les
caractères old sont remplaces par new. Si count est fourni
(facultatif), seules les occurrences count sont
remplacees.
```

Restons dans la fenêtre de l'interpréteur pour faire quelques tests sur cette méthode. Il nous faut définir deux variables de type chaîne ne contenant chacune qu'un seul caractère. Nous pourrons appeler l'une des deux `vcar` (vieux caractère) et la nouvelle `ncar` (le nouveau caractère). Voici notre petit essai en mode interprété. Tous les caractères du message à traiter qui sont identiques à celui spécifié dans la chaîne `vcar` sont remplacés par celui spécifié dans la chaîne `ncar`.

```
>>> msg = 'Une bouteille a la mer !'
>>> vcar = 'e'
>>> ncar = '3'
>>> msg_leet = msg.replace(vcar,ncar)
>>> msg_leet
'Un3 bout3ill3 a la m3r !'
```

Tu remarques que nous récupérons le résultat dans une autre chaîne. Effectivement, cette méthode renvoie la chaîne après traitement. Dans notre exemple, les lettres e du message ont été remplacées par des chiffres 3. La chaîne du message initial n'est pas modifiée. Les deux arguments `vcar` et `ncar` doivent être du type chaîne, mais pour faire des essais rapides, il est inutile de les stocker d'abord dans des variables. On peut appeler la méthode de la façon suivante :

```
>>> msg_leet = msg.replace('e','3')
```

Tu n'es jamais obligé de créer tes variables avec les mêmes noms que dans l'exemple fourni par l'aide dans la doc-chaîne. Pour le texte source de notre projet, j'ai décidé de donner les noms `vcar` (vieux caractère) et `ncar` (nouveau caractère) aux arguments.

Remplaçons une seule lettre

Passons maintenant à l'insertion du code de remplacement dans notre fonction. Voyons ce que la fonction doit faire.

1. **Elle doit balayer caractère par caractère la chaîne du message à convertir.** De plus, elle doit chercher toutes les substitutions définies. Même si nous ne testons pour l'instant que pour une lettre(e), nous allons gagner du temps en anticipant le cas général, et nous écrivons ceci :

```
for s in substitutions:
```

2. **Pour chaque tour de boucle, la fonction doit extraire le couple ancien/nouveau.** Nous partons d'une liste de couple de chaînes. Il faut extraire de la liste l'ancien et le nouveau caractère pour les fournir en argument de la méthode `replace()`.

```
vcar = s[0]
ncar = s[1]
```

3. **Nous pouvons ensuite appeler la méthode `replace()` de la variable `message` à traiter pour réaliser les substitutions.**

```
converti = message.replace(vcar, ncar)
```

4. **Il ne reste plus qu'à renvoyer le message une fois converti :**

```
return converti
```

Voici la version complète du texte source de notre projet. Les nouveautés sont l'ajout de la boucle `for` dans la fonction `coder_message()` et l'instruction de renvoi `return`. De plus, dans la section principale en bas, j'ai ajouté une instruction d'affichage du message de test.

Listing 6.2 : Texte source de 1337.py (version 2)

```
""" 1337.py
Codage d'un message en langue Leet Speak (1337 sp34k)
Brendan Scott (VF par OE)
Version 2 (pour traiter 1 seule lettre) """

TEST_MESSAGE = "Bienvenue dans le monde de 1337 !"
TEST_SUBSTITUTIONS = [['e','3']]

#### Section des fonctions
def coder_message(message, substitutions):
    """Traite une chaine transmise et applique des
substitutions
    de lettres d'apres une liste d'elements. Chaque element
    est une liste de longueur 2 (ancien, nouveau)"""

    for s in substitutions:      #AJOUT DE 4 LIGNES
        vcar = s[0]
        ncar = s[1]
        converti = message.replace(vcar, ncar)
    return converti              #AJOUT

#### Section de test dans main
txt_converti = coder_message(TEST_MESSAGE, TEST_SUBSTITUTIONS)
print(TEST_MESSAGE)              #AJOUT
print(txt_converti)
```

Voici l'affichage qui résulte de l'exécution :

```
>>> ============================== RESTART ==================
>>>
Bienvenue dans le monde de 1337 !
Bi3nv3nu3 dans l3 mond3 d3 1337 !
```

Tout semble marcher à merveille, mais il y a un ver dans le fruit, ou plutôt une erreur de logique. En effet, le programme fonctionne avec la substitution telle qu'elle est définie, mais il ne marchera plus lorsque nous voudrons généraliser. Te voici prévenu. Nous résoudrons ce problème d'ici quelques pages.

Les erreurs de logique sont difficiles à localiser et à réparer. L'interpréteur Python ne peut pas t'aider parce qu'à ses yeux le code source est tout à fait correct, bien qu'il ne se comporte pas comme tu le désires. Les principales causes d'erreurs de logique sont une incompréhension du flux d'exécution réel ou une méprise sur la valeur que possèdent les variables.

Au fur et à mesure que l'on avance dans la rédaction d'un programme, on se familiarise avec et on finit par émettre des hypothèses sur ce que fait le code. C'est pourquoi je te l'avais déjà conseillé dans un projet antérieur : dès que tu as un souci, dis à voix haute comment fonctionne le programme. Si tu réalises les exercices de ce livre en groupe, tu auras naturellement l'occasion de discuter avec tes collègues des problèmes que vous rencontrez.

Faisons saisir le message

Nous savons déjà comment obtenir la saisie d'un texte, puisque nous l'avons fait dans le Projet 3.

1. **Nous demandons la saisie au moyen de la fonction standard `raw_input()`.**
2. **Nous récupérons dans une variable le texte qu'elle renvoie.**
3. **Nous transmettons directement la variable à la fonction `coder_ message()` pour qu'elle traite la chaîne.**
4. **Pour pouvoir comparer, nous affichons le message avant codage.**
5. **Nous modifions enfin l'instruction d'affichage du message encodé.**

Voici la troisième version du projet. Nous n'avons réalisé de modification que dans la section principale, tout en bas. Nous avons ajouté une ligne pour récupérer ce que saisit l'utilisateur et modifié l'argument de celle qui affiche le message original.

Listing 6.3 : Texte source de 1337.py (version 3)

```
""" 1337.py
Codage d'un message en langue Leet Speak (1337 sp34k)
Brendan Scott (VF par OE)
Version 3 (pour faire saisir le message pour 1 seule lettre)
"""

TEST_MESSAGE = "Bienvenue dans le monde de 1337 !"
TEST_SUBSTITUTIONS = [['e','3']]

#### Section des fonctions
def coder_message(message, substitutions):
"""Traite une chaine transmise et applique des
substitutions
    de lettres d'apres une liste d'elements. Chaque element
    est une liste de longueur 2 (ancien, nouveau) """

    for s in substitutions:      #AJOUT DE 4 LIGNES
        vcar = s[0]
        ncar = s[1]
        converti = message.replace(vcar, ncar)
    return converti              #AJOUT

#### Section de test dans main
message = raw_input("Message que je dois crypter : ")
#AJOUT
txt_converti = coder_message(message, TEST_SUBSTITUTIONS)
#MODIF
print(message)
#MODIF
print(txt_converti)
```

Réalise ces modifications puis lance l'exécution du programme. Saisis un message en faisant en sorte qu'il y ait au moins une lettre `e` pour que tu puisses voir l'effet du remplacement.

```
>>> ================================ RESTART ================
>>>
Message que je dois crypter : Elle est trop belle !
Elle est trop belle !
Ell3 3st trop b3ll3 !
```

Définissons toutes les substitutions

Tu ne peux prétendre comprendre le langage des hackers si tu ne fais que changer tous les `e` dans les messages. Il faut aller plus loin ! Voici les cinq substitutions que je te propose de mettre en place. Tu peux en ajouter d'autres plus tard, si le cœur t'en dit.

Nous allons pouvoir encoder les cinq substitutions en tant que liste. L'ordre dans la liste est significatif. Par exemple, le couple `['a', '4']` stocke la sous-liste qui permet de remplacer tous les exemplaires du premier élément (le caractère a, en position zéro) par le chiffre fourni en second élément (le chiffre 4, en position 1). On ne peut pas écrire `[a, 4]` parce que Python croirait que `a` est le nom d'une variable et que 4 est une valeur numérique entière (pas le chiffre 4).

Lettre	Chiffre de remplacement
a	4
e	3
l	1
o	0
t	7

Voici comment ces cinq substitutions peuvent être écrites :

```
['a','4'], ['e','3'], ['l','1'], ['o','0'], ['t','7']
```

Le résultat doit être une liste de cinq sous-listes, ce qui oblige à ajouter une paire de crochets autour du groupe de cinq paires de crochets. Faisons un essai dans l'interpréteur :

```
>>> substitutions = [['a','4'], ['e','3'], ['l','1'],
['o','0'],
['t','7']]
>>> for s in substitutions:
    print(s)

['a', '4']
['e', '3']
['l', '1']
['o', '0']
['t', '7']
```

J'utilise une boucle pour extraire les cinq sous-listes et les afficher, une par ligne. Récapitulons ce que nous avons fait :

- Nous avons créé une liste contenant cinq éléments.
- Chacun des cinq éléments est une liste. Chaque sous-liste ne contient que deux éléments.
- Chacun des deux éléments est une chaîne, d'un modèle un peu particulier, puisqu'elle ne contient qu'un caractère.

J'avais décidé dès le départ de donner dans la fonction d'encodage le nom `substitutions` (au pluriel) à l'argument pour la liste, alors que la liste ne contenait qu'un seul couple pour traiter la lettre e. Les instructions que nous allons écrire maintenant fonctionnent de la même façon qu'il y ait un élément ou plusieurs. La boucle `for` testée juste ci-dessus fonctionne avec une comme avec cinq substitutions.

Nous avons testé le code avec une seule substitution puis étoffé la liste. Nous utilisons une approche pas à pas, qui est celle conseillée.

C'est un sentiment grisant mais dangereux de voir qu'on avance vite dans la rédaction du code source. Mais à moins d'être parfait, et personne ne l'est, il y aura nécessairement des erreurs dans tes lignes. Et plus tu enchaînes de lignes sans faire de tests intermédiaires, plus ce sera difficile de trouver les causes des erreurs. Il faut subdiviser le code en portions de petite taille et tester chaque portion avant de poursuivre. Une fois que deux portions sont réunies, il faut aussi tester le résultat de cette association.

Appliquons les cinq substitutions

Une fois que ton code source semble fonctionner pour une seule substitution, tu peux généraliser pour appliquer les cinq substitutions que nous voulons réaliser.

1. **Nous définissons une nouvelle constante portant le nom `SUBSTITUTIONS` (oui, en capitales) et lui affecter la liste des cinq sous-listes préparée plus haut.**
2. **Dans l'appel à la fonction de codage, c'est cette liste que nous devons transmettre. La fonction de codage la reçoit dans un argument portant apparemment le même nom, mais en minuscules. C'est une autre variable ! Python distingue les CAPITALES des minuscules.**
3. **Nous pouvons neutraliser par mise en commentaire la constante `TEST_SUBSTITUTIONS`.**

Tu peux décider de supprimer les lignes des deux constantes de test ou de les conserver en commentaires. Tu auras peut-être envie de faire d'autres tests plus tard. Voici donc la quatrième version de notre projet.

Listing 6.4 : Texte source de 1337.py (version 4)

```
""" 1337.py
Codage d'un message en langue Leet Speak (1337 sp34k)
Brendan Scott (VF par OE)
Version 4 (pour traiter cinq lettres) """

TEST_MESSAGE = "Bienvenue dans le monde de 1337 !"
##TEST_SUBSTITUTIONS = [['e','3']]   #SUPPR et #AJOUT suivante
SUBSTITUTIONS = [['a','4'], ['e','3'], ['l','1'], ['o','0'],
['t','7']]

#### Section des fonctions
def coder_message(message, substitutions):
    """Traite une chaine transmise et applique des
substitutions
    de lettres d'apres une liste d'elements. Chaque element
    est une liste de longueur 2 (ancien, nouveau) """

    for s in substitutions:
        vcar = s[0]
        ncar = s[1]
        converti = message.replace(vcar, ncar)
    return converti

#### Section de test dans main
message = raw_input("Message que je dois crypter : ")
txt_converti = coder_message(message, SUBSTITUTIONS)
#MODIF
```

```
print(message)
print(txt_converti)
```

Je te conseille de faire une sauvegarde du projet dans l'état actuel (par exemple sous le nom de fichier *Po7_Debug*), car nous allons nous en resservir pour découvrir le débogueur de l'atelier IDLE en fin de chapitre. Tu peux ensuite lancer en toute confiance l'exécution de ton convertisseur pour hackers :

```
>>> ============================ RESTART ======================
>>> Message que je dois crypter : Que tu es belle, ma campagne
 !
Que tu es belle, ma campagne !
Que 7u es belle, ma campagne !
```

Tiens, tiens… le fonctionnement est bizarre. Les e auraient du devenir des 3 et les a des 4, non ? Seule la dernière substitution a fonctionné, celle qui remplace les t par des 7. Il y a vraiment une erreur de logique dans ce projet. Comment se fait-il qu'il n'y ait qu'une substitution qui soit appliquée ?

Un débogage minimal avec print

La légende dit que c'est une informaticienne américaine nommée Grace Hopper qui a inventé le terme de *bug* (que l'on traduit en français par « bogue »).

Vers 1950, à force de chercher la cause de la panne d'un de ses calculateurs (les tout premiers ordinateurs de l'histoire), elle finit par trouver un insecte (bug) bien au chaud parmi les lampes, qui dévorait l'isolant textile des fils électriques. Le débogage est devenu un art. Plus tu le pratiques, plus tu t'améliores dans la chasse aux bogues.

Voici quelques tactiques de débogage :

> » En insérant des instructions d'affichage `print()`, on peut visualiser les valeurs de certaines données. Ces valeurs varient en fonction du lieu d'affichage. En général, on commence par afficher les données sur lesquelles le programme travaillait à l'endroit où on pense que se trouve l'erreur puis on continue à progresser ainsi d'étape en étape.
>
> » En ajoutant des instructions d'affichage pour marquer la progression dans le flux d'exécution, du style `print("J'entre dans fonctruc()")` puis `print("Je sors de fonctruc()")`. Cela permet de savoir quand le flux est entré ou sorti d'une fonction.
>
> » Dans notre exemple, nous pourrions faire varier le contenu de la liste SUBSTITUTIONS pour voir quel effet cela aurait sur les remplacements.
>
> » Il faut toujours être méfiant parce que l'endroit où l'erreur semble située n'est pas toujours la vraie cause de l'erreur. Il faut remonter en arrière dans le flux d'exécution jusqu'au point apparent de survenue d'erreur et chercher ce qui peut en être la cause réelle.

Voyons comment appliquer ces règles au problème qui nous concerne ici.

1. **Nous allons ajouter plusieurs instructions d'affichage dans le code source.**
2. **Chaque instruction d'affichage soit affiche la valeur d'une variable, soit marque une étape dans le flux d'exécution.**
3. **Une fois ces témoins en place, nous pouvons lancer l'exécution du programme et analyser ce qui s'affiche.**

LA JOURNALISATION DE PYTHON (LOGGING)

Python est doté d'un module spécial pour journaliser l'exécution (c'est surtout utile pour les grands projets). Il recopie tous vos messages dans un fichier, ce qui permet de l'étudier calmement après l'exécution. Le fichier contient de nombreux détails, par exemple l'heure à laquelle chaque message a été écrit et le numéro de ligne source dans le programme. Ces détails sont extrêmement utiles, et ils n'apparaissent pas lorsque tu utilises simplement des instructions `print()` comme ici.

Voici donc la version de débogage du programme, qui est la version 5. Les commentaires de fin de ligne `#AJOUT` et `#MODIF` montrent où nous avons changé le texte source par rapport à la version précédente.

Listing 6.5 : Projet 1337.py (version 5)

```
""" 1337.py
Codage d'un message en langue Leet Speak (1337 sp34k)
Brendan Scott (VF par OE)
Version 5 (debogage avec des ajouts de print) """

TEST_MESSAGE = "Bienvenue dans le monde de 1337 !"
##TEST_SUBSTITUTIONS = [['e','3']]
SUBSTITUTIONS = [['a','4'], ['e','3'], ['l','1'], ['o','0'],
['t','7']]

#### Section des fonctions
def coder_message(message, substitutions):
"""Traite une chaine transmise et applique des
substitutions
de lettres d'apres une liste d'elements. Chaque
element est une liste de longueur 2 (ancien, nouveau) """

for s in substitutions:
vcar = s[0]
ncar = s[1]
converti = message.replace(vcar, ncar)
print("Texte converti = " + converti)    #AJOUT
print("Je sors de coder_message()")          #AJOUT
return converti

#### Section de test dans main
message = raw_input("Message que je dois crypter : ")
txt_converti = coder_message(message, SUBSTITUTIONS)
print("Tu avais saisi ceci  : " + message)         #MODIF
print("Le codage donne ceci : " + txt_converti)    #MODIF
```

J'ai ajouté deux instructions d'affichage dans la fonction : une dans la boucle et une juste avant de quitter la fonction. La première permet de bien voir le traitement d'un des cinq caractères. (Note bien l'indentation différente des lignes `print()`.) La deuxième instruction est en dehors du bloc de répétition puisqu'elle est plus proche de la marge gauche. Lorsqu'elle s'affiche, nous sommes certains que nous sortons de la fonction.

Voici le genre d'affichage que l'on obtient.

```
Message que je dois crypter : Elle est vraiment belle.
Texte converti = Elle est vr4iment belle.
Texte converti = Ell3 3st vraim3nt b3ll3.
Texte converti = E11e est vraiment be11e.
Texte converti = Elle est vraiment belle.
Texte converti = Elle es7 vraimen7 belle.
Je sors de coder_message()
Tu avais saisi ceci  : Elle est vraiment belle.
```

```
Le codage donne ceci : Elle es7 vraimen7 belle.
```

Dans la version précédente, j'avais remarqué que seul le dernier remplacement de caractère était fait, car on voyait le résultat dans les instructions d'affichage du bloc principal. En fait, je me trompais ! Chaque caractère est traité correctement, mais à chaque tour de boucle, on repart du message initial, et on perd les traitements précédents. Le traitement n'est pas cumulatif.

Il s'agit d'un exemple typique d'erreur de logique. La fonction ne travaille pas sur la chaîne en cours de conversion, mais toujours sur la chaîne initiale, ce qui fait perdre les étapes de traitement précédentes.

La solution est ridiculement simple, puisque nous allons pouvoir nous passer de la variable locale `converti`. Nous pouvons tout à fait renvoyer dans la variable `message` que nous avons obtenue en argument d'entrée le résultat du traitement sur cette même variable. Il y a donc en tout trois lignes à modifier pour qu'elles utilisent la variable `message` à la place de la variable `converti`. Je te laisse effectuer ces trois retouches sans mon aide, mais compare ensuite avec la version finale du programme que je te propose ci-après.

Listing 6.6 : Projet 1337.py (version 6, finale)

```
""" 1337.py
Codage d'un message en langue Leet Speak (1337 sp34k)
Brendan Scott (VF par OE)
Version 6 FINALE (on traite le message original) """

TEST_MESSAGE = "Bienvenue dans le monde de 1337 !"
##TEST_SUBSTITUTIONS = [['e','3']]
SUBSTITUTIONS = [['a','4'], ['e','3'], ['l','1'], ['o','0'],
['t','7']]

#### Section des fonctions
def coder_message(message, substitutions):
    """Traite une chaine transmise et applique des
substitutions
    de lettres d'apres une liste d'elements. Chaque
    element est une liste de longueur 2 (ancien, nouveau) """

    for s in substitutions:
        vcar = s[0]
        ncar = s[1]
        message = message.replace(vcar, ncar) #MODIF
        print("Texte converti = " + message)  #MODIF
    print("Je sors de coder_message()")
    return message                            #MODIF

#### Section de test dans main
message = raw_input("Message que je dois crypter : ")
txt_converti = coder_message(message, SUBSTITUTIONS)
print("Tu avais saisi ceci  : " + message)
print("Le codage donne ceci : " + txt_converti)
```

Voici l'affichage que tu devrais obtenir :

```
>>> ============================== RESTART ==================
>>>
Message que je dois crypter : Elle est vraiment trop belle.
Texte converti = Elle est vr4iment trop belle.
Texte converti = Ell3 3st vr4im3nt trop b3ll3.
Texte converti = E113 3st vr4im3nt trop b3113.
Texte converti = E113 3st vr4im3nt tr0p b3113.
Texte converti = E113 3s7 vr4im3n7 7r0p b3113.
Je sors de coder_message()
Tu avais saisi ceci  : Elle est vraiment trop belle.
```

```
Le codage donne ceci : E113 3s7 vr4im3n7 7r0p b3113.
```

Une fois que ton programme fonctionne comme prévu, tu peux neutraliser par mise en commentaires (ou même supprimer) les lignes des instructions d'affichage qui ne servaient que pour le débogage.

Je rappelle qu'il suffit de mettre un signe `#` au début d'une ligne pour la neutraliser.

Profitons du débogueur de IDLE

L'atelier IDLE est doté d'un outil de mise au point qui est le *débogueur* intégré. Il permet d'arrêter l'exécution sur une ligne choisie (ce qui s'appelle un *point d'arrêt*) puis de continuer à progresser à partir de là. Le débogueur permet d'afficher les valeurs de toutes les variables. Je n'ai pas la place dans ce livre pour décrire toutes les possibilités de cet outil, mais nous pouvons faire un petit tour rapide. À toi de le découvrir plus amplement par la suite !

Les utilisateurs de Mac OS remarqueront que le débogueur IDLE ne fonctionne pas très bien, voire pas du tout. Tout dépend de la version de Python et de la version du système, et peut-être même des phases de la lune. Au lieu d'utiliser le clic-droit, tu peux essayer la combinaison Commande+clic. Si cela ne marche pas non plus, tu ne peux que lire ce qui suit sans pratiquer.

1. **Charge dans l'éditeur (par File/Open ou Fichier/Ouvrir) le code source correspondant au fichier en version 4 du projet. Je t'avais invité à en faire une sauvegarde. Le moment est venu de t'en servir.**
2. **Tu peux aussi ouvrir un nouveau fichier dans l'atelier IDLE puis copier/coller le code de la version 4 entre deux fenêtres.**
3. **Enregistre le nouveau fichier sous le nom** *debogage.py* **(sans accent, c'est mieux).**
4. **Reviens à la fenêtre de l'interpréteur Shell et choisis la commande Debug/Debugger ou Débogage/Debugger)**

   Tu dois voir apparaître une nouvelle fenêtre, qui est celle de contrôle du débogueur (Figure 6.2).
5. **Reviens maintenant dans la fenêtre d'édition du code source. Clique dans la ligne qui correspond dans ton programme à l'appel à la fonction `coder_message()`. C'est celle-ci :**

```
txt_converti = coder_message(message, SUBSTITUTIONS)
```

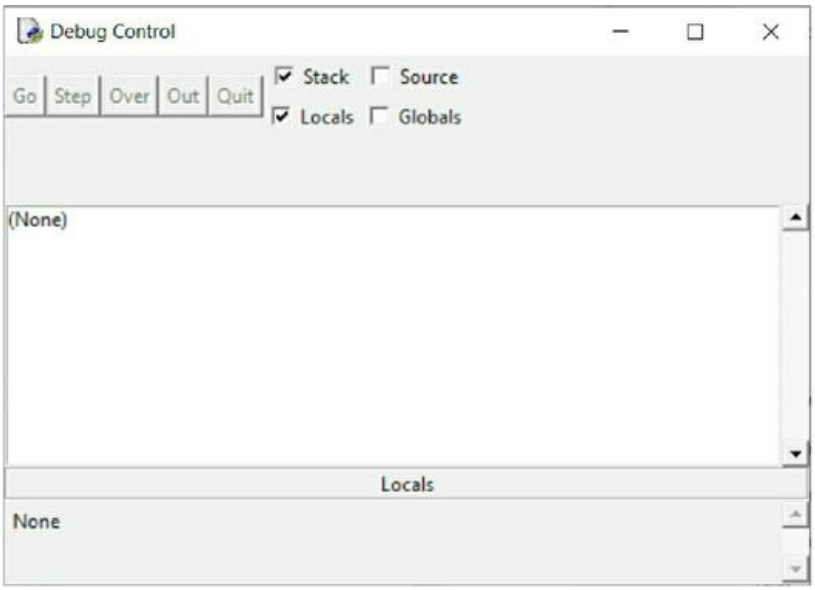


Figure 6.2 : Aspect général de la fenêtre de contrôle du débogueur.

6. **Active la ligne qui effectue la conversion du message, c'est-à-dire :**

```
converti = message.replace(vcar, ncar)
```

Chez moi, c'est la ligne 19. Clique-droit dans cette ligne pour ouvrir un menu local et, pour poser un point d'arrêt, choisis la commande Set Breakpoint (ou Poser point d'arrêt).

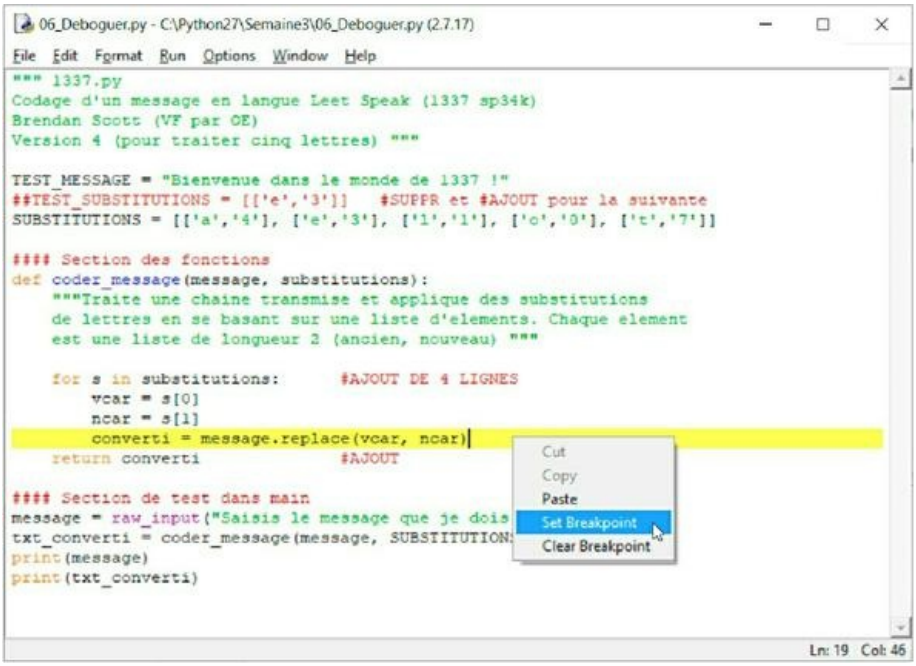


Figure 6.3 : Pose d'un point d'arrêt depuis le menu local.

Utilisateurs Mac OS : tentez le raccourci Commande+Clic, tout en sachant qu'il ne fonctionnera peut-être pas.

Le numéro de la ligne courante est indiqué dans le coin inférieur droit (Figure 6.3) Dès que tu as posé un point d'arrêt, la ligne concernée est surlignée en jaune.

7. **Une fois que ce point est posé, tu peux lancer l'exécution du code comme d'habitude.**

L'exécution commence puis l'atelier IDLE donne le contrôle à la fenêtre Debug Control. Alors que cette fenêtre était vide et ennuyeuse, la voici qui fourmille d'informations intéressantes (Figure 6.4).

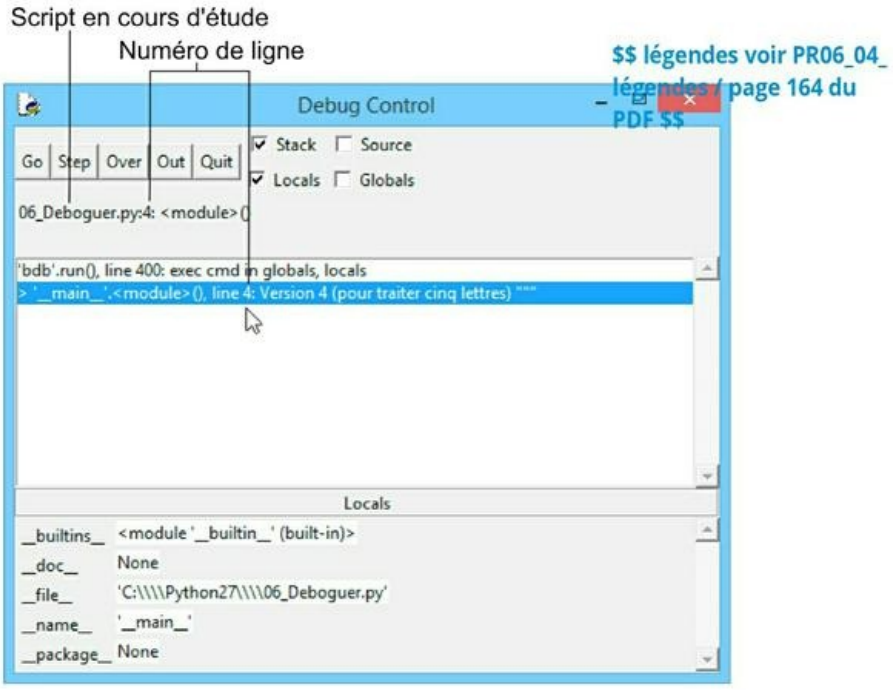


Figure 6.4 : La fenêtre Debug Control en pleine activité.

Voici ce qu'on peut apprendre en regardant la fenêtre de contrôle :

Nous sommes pour l'instant dans la ligne 4 du programme *debogage.py*.
La prochaine ligne de code qui sera exécutée se termine par un triple guillemet. C'est donc un commentaire. (Si la ligne est suffisamment courte, tu peux la voir en entier.)
Les valeurs que possèdent les variables locales sont indiquées dans le panneau Locals en bas.

8. **Clique le bouton Go en haut à gauche de la fenêtre.**

   Nous arrivons ainsi à l'étape de saisie du message à crypter, donc dans la fenêtre de l'interpréteur Shell. Nous avions demandé à Python d'exécuter jusqu'au prochain point d'arrêt, mais pour l'instant, nous n'y sommes pas encore. La fonction `raw_input()` provoque une pause. Il faut saisir du texte et valider pour que le programme puisse continuer.

9. **Bascule dans la fenêtre de l'interpréteur pour saisir ton message, par exemple `"Elle est trop belle"` (sans les guillemets) et valide par Entrée. La Figure 6.5 montre ce qui se passe alors.**

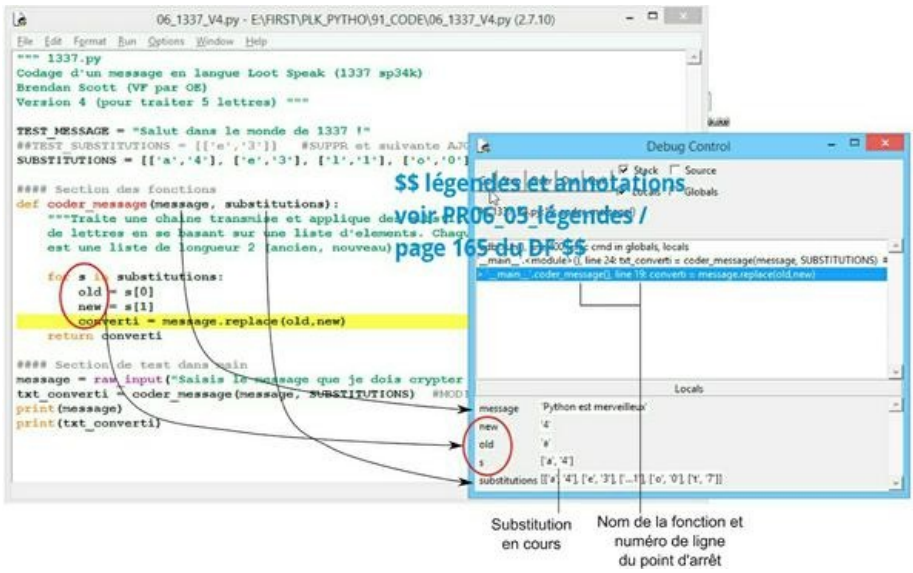


Figure 6.5 : Accès aux variables témoins sur un point d'arrêt.

   La fenêtre de contrôle a encore changé. Nous pouvons maintenant apprendre ceci :

   Nous sommes dorénavant en pause en ligne 19 dans une fonction qui porte le nom `coder_message()`.

   Les valeurs de toutes les variables qui sont connues dans cette fonction sont affichées dans le panneau Locals. La substitution en cours concerne `['a', '4']`, la valeur de `ncar` est 4 et la valeur de `vcar` est a.

10. **Clique une fois de plus le bouton Go.**

    L'exécution retombe sur le point d'arrêt. En effet, nous l'avons posé dans une boucle. L'exécution s'y arrête donc à chaque tour de boucle. Dans le panneau Locals, tu peux voir qu'une nouvelle variable a été ajoutée à la liste. Tu vois également que la valeur de `message` n'a pas changé, malgré le tour de traitement.

    Si tu cliques encore Go, tu continues à faire des tours de boucle. Le `message` local ne change pas. Pourtant, la variable `converti` change. Dans le tour précédent, les a ont été remplacés par des 4, mais maintenant ils sont redevenus des a. J'imagine que tu as compris où était le problème. La variable `message` n'est pas mise à jour suite aux étapes de traitement. Nous avions un doute d'ailleurs en voyant le dernier affichage dans la section principale.

Tu peux essayer les autres boutons du débogueur.

Pour ôter un point d'arrêt, il suffit de cliquer-droit (Commande+Clic sur Mac) dans la ligne concernée et de choisir la commande Clear Breakpoint (ou Enlever point d'arrêt). Tu peux ensuite refermer le débogueur depuis la fenêtre de l'interpréteur Shell par la commande Debug/Debugger.

Récapitulons

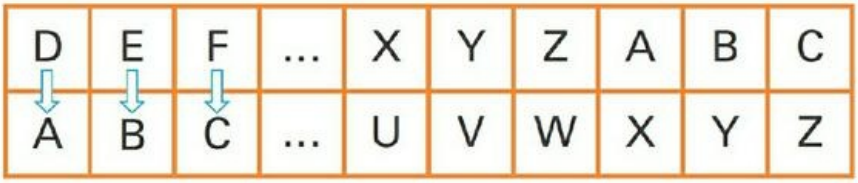
Passons en revue les sujets abordés pendant la réalisation de ce projet. Nous avons vu :

- que tout dans Python était un objet et qu'il existait plusieurs types d'objets.
- que chaque objet possédait un identifiant unique ID qui est l'adresse mémoire à laquelle l'objet est stocké.
- les attributs des objets. Certains attributs sont des variables ou des littéraux et d'autres sont des fonctions qui s'appellent dans ce cas des méthodes.
- comment désigner ici l'attribut d'un objet au moyen de la syntaxe fondée sur le point séparateur : `nomobjet.nomattribut`. Lorsque l'attribut est une méthode, nous ajoutons la paire de parenthèses avec les arguments éventuels.
- plusieurs méthodes des objets du type liste pour modifier l'original de la liste. Je t'ai mis en garde sur le fait que ces fonctions ne renvoient pas toutes la liste modifiée, mais la pseudo-valeur `None`.
- comment créer des objets du type liste avec des éléments.
- comment créer une liste par compréhension et en utilisant une condition.
- le mot réservé `in` qui permet de savoir si une liste contient un certain élément. Nous avons inversé le test avec le mot réservé `not` devant `in` pour vérifier qu'un élément n'est pas dans une liste.
- la méthode `replace()`, qui permet de modifier un objet de type chaîne.
- comment effectuer un débogage minimal en ajoutant des instructions d'affichage `print()`,
- les grands principes du débogueur intégré à l'atelier IDLE, comment poser un point d'arrêt pour suspendre l'exécution dans le débogueur et comment inspecter les valeurs des variables locales pendant les pauses dans l'exécution.

Projet 7

Cryptopy !

Nous allons poursuivre dans le même domaine que dans le projet précédent pour créer un programme pour crypter et décrypter des messages. Nous allons nous servir d'un chiffrement très ancien, le chiffrement César.



Je ne te ferai pas l'affront de te présenter Jules César, puisque l'histoire de Rome est au programme de l'école primaire. Les chiffrements sont indispensables aux espions pour transmettre des messages sans qu'ils puissent être lus, sauf par le destinataire. Le chiffrement est une procédure ou un appareil qui transforme un texte normal (le texte en clair) en un texte illisible (le texte secret). Dans le chiffre de César, on décale les lettres de l'alphabet de quelques positions. Pour écrire la lettre `d`, on écrit `a`. Pour `e`, on écrit `b`, etc.

Le signe antibarre est en fait un code d'échappement qui doit être suivi d'un autre caractère, l'ensemble des deux prenant une signification spéciale. Lorsque tu demandes l'affichage d'une chaîne contenant la séquence `\n`, tu ne verras pas apparaître à l'écran ni l'antibarre ni le n, mais cela va provoquer un saut de ligne. Vérifions cela :

```
>>> print("1\t2")
1	2
>>> print("1\n2")
1
2
```

LES CARACTÈRES IMPRIMABLES

Nous allons avoir besoin de la liste de tous les caractères imprimables. Il y a un module standard nommé *string* qui possède une fonction renvoyant tous les caractères imprimables, qui sont ses attributs. Voici comment faire générer cette sorte d'alphabet :

```
>>> import string
>>> string.printable
'0123456789abcdefghijklmno-
pqrstuvwxyzABCDEFGHIJKLMNO
PQRSTUVWXYZ!"#$%&\     '()*+,-
./:;<=>?@[\\]^_`{¦}~   \t\n\r\
x0b\x0c'
```

La fin de la liste est un peu étrange. À quoi servent toutes ces barres obliques inverses (je les appelle des antibarres) ? Elles correspondent à des caractères de contrôle qui ne s'affichent normalement pas :

\t un pas de tabulation

\n un saut de ligne

\r un retour chariot

\\pour afficher l'antibarre elle-même Voici les différentes étapes que nous allons traverser pour réaliser ce projet :

- Nous allons partir d'un message en texte clair.
- Nous cryptons ce message.
- Nous affichons le message crypté.
- Nous décryptons le message pour retrouver un texte en clair.
- Nous affichons le texte une fois décrypté pour contrôle.

Notre programme doit pouvoir gérer les lettres en majuscules et en minuscules, ainsi que les signes de ponctuation et les chiffres. Jules César s'était limité aux lettres en minuscules. (César considérait le latin comme monocaméral, c'est-à-dire constitué uniquement de lettres en minuscules.)

Du balai, les caractères de contrôle !

Les derniers caractères de la liste, à partir de `\t`, n'ont pas besoin d'être cryptés. D'ailleurs, s'il y a un saut de ligne dans le texte source, il faudra normalement le conserver, et ne pas le traduire en une lettre. De même, si le texte crypté contient un saut de ligne, mieux vaut le laisser tel quel dans le texte décrypté. Tous les autres caractères qui sont générés par `string.printable` peuvent faire l'objet d'un cryptage.

Il faut donc filtrer la liste des caractères. Python propose à cet effet un opérateur spécial qui s'écrit `[:]` et permet de récupérer un sous-ensemble d'une chaîne ou d'une liste.

Faisons des tests dans l'interpréteur :

```
>>> str_tst = '012345789'
>>> str_tst[0:1]
'0'
>>> str_tst[1:3]
'12'
>>> range(10)[0:1]
[0]
>>> range(10)[1:3]
[1, 2]
```

```
>>> str_tst[:3]
'012'
>>> str_tst[3:]
'345789'
```

L'opérateur d'extraction s'écrit sous la forme `nomchaine[a : b]`. Tu remplaces bien sûr `nomchaine` par le nom de la chaîne sur laquelle il faut travailler. La variable `a` doit être remplacée par l'index dans la chaîne, c'est-à-dire le nombre de caractères à partir duquel il faut commencer l'extraction. La variable `b` indique la position du dernier caractère dans la chaîne. La chaîne de chiffres qui s'écrit `'0123456789'` est un peu spéciale, puisque chaque caractère équivaut à son index. La chaîne `'0'` est à l'index 0, la chaîne `'3'` est à l'index 3, etc. Dans la chaîne `'Salut'`, la lettre S est à l'index 0, la lettre a à l'index 1, etc.

L'extraction permet de récupérer tous les caractères à partir du caractère indiqué par la position `a`, jusqu'au caractère qui précède celui désigné par la valeur `b`. C'est un peu déroutant, puisque les indices des positions commencent à zéro. Sans compter le fait qu'il est possible de donner des valeurs négatives pour `a` et `b`. Dans ce cas, la recherche de la sous-chaîne se fait à partir de la fin.

```
>>> # tout sauf le dernier caractère
>>> str_tst[:-1]
'01234578'
>>> # tout à partir du dernier caractère
>>> str_tst[-1:]
'9'
```

Dans notre projet, il ne sera pas difficile de récupérer la sous-chaîne nettoyée des caractères de contrôle de la fin. Il suffit de prendre tout sauf les cinq derniers. Le résultat est stocké dans une nouvelle variable que nous appelons `jeucar`. Elle va contenir dans notre projet le jeu de caractères que le cryptage doit traiter :

```
>>> jeucar = string.printable[:-5]
>>> jeucar
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU-
VWXYZ!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~ '
```

Nous disposons maintenant de la liste de toutes les lettres, chiffres et signes de ponctuation que nous devons crypter.

Pour découper une tranche dans une liste, le procédé est exactement le même que pour une chaîne, sauf que les éléments qui sont extraits ne sont pas des caractères, mais des éléments de liste.

Créons notre table de substitution

Tu as peut-être déjà envoyé un message codé à un ami ou cherché à écrire quelque chose de telle manière que tes parents ne puissent pas te relire ? Et je ne parle pas du nouveau jargon utilisé pour échanger des SMS.

Nous devons donc remplacer chaque caractère du message en clair par un autre caractère du message crypté. Dans le chiffre de César, le décalage choisi était de trois lettres, c'est-à-dire que la lettre a remplace la lettre d. Comment peut-on automatiser ce travail ?

Observe la Figure 7.1. Elle montre la convention de cryptage que nous choisissons. En travaillant à la main, il faut chercher chacune des lettres dans la ligne du haut et écrire la lettre située dans la même colonne avant de confier le message au messager. Lorsque le mot à crypter est BYE, le cryptage donne YVB.

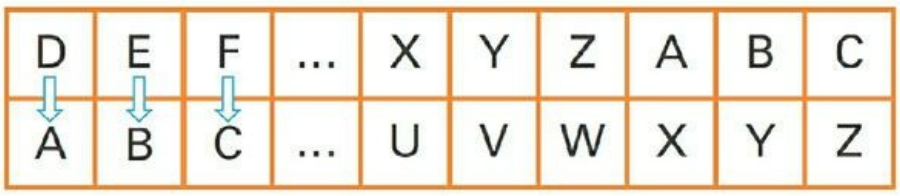


Figure 7.1 : Extrait de la table de cryptage de César.

Nous utilisons ici un décalage de trois positions. Nous allons donc supprimer les trois premiers caractères de la liste et les recoller à la fin. Pour une autre clé de cryptage, il suffit de choisir un décalage différent.

L'extrait suivant montre comment créer la liste de substitution à partir de la liste initiale. Nous affichons cette liste pour que tu puisses voir le résultat :

```
>>> subcar = jeu_car[-3:]+jeu_car[:-3]
>>> subcar
'}~ 0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU-
VWXYZ!"#$%&\'()*+,-./:;<=>?@[\\]^_`{¦'
```

Nous avons donc maintenant notre alphabet de départ et notre alphabet d'arrivée. Dans le prochain exemple, nous voyons la liste pour le texte en clair puis celle des caractères après décalage. Nous ne montrons que le début de la liste.

```
>>> print(jeu_car[:62]+'\n'+subcar[:62])
0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
}~ 0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVW
```

Pour afficher la liste complète, il suffit de saisir l'instruction suivante dans la fenêtre de l'interpréteur IDLE :

```
print(jeu_car+'\n'+subcar)
```

Démarrons notre projet

Commençons à mettre en place notre fichier de code source.

1. **Ouvre une fenêtre d'édition et commence par créer un fichier vide portant le nom *cryptopy.py*.**
2. **Insère en haut une doc-chaîne pour décrire globalement ce nouveau module.**
3. **Mets en place une section d'import et ajoute une directive pour importer le module `string`.**

```
#### Imports
import string
```

4. **Installe une section pour les constantes et récupère les instructions précédentes pour créer JEUCAR et CARSUBSTI. Modifie les noms pour qu'ils soient en capitales, puisque ce sont des données constantes.**

```
#### Constantes
JEUCAR = string.printable[:-5]
CARSUBSTI = JEUCAR[-3:]+JEUCAR[:-3]
```

5. **Définis une autre constante portant le nom MSG_TEST et stockes-y un message de test.**

```
MSG_TEST = "J'adore les Monty Python. Trop cool."
```

6. **Nous pouvons passer à la section des fonctions. Mets en place l'amorce d'une fonction portant le nom `encrypter()`, et n'attendant qu'un argument, qui sera le texte lisible (`texteclair`). Pense à mettre en place immédiatement une doc-chaîne de fonction. Pour l'instant, fais renvoyer le même message par `return`.**

J'ai expliqué au Projet 5 comment créer une amorce de fonction. Retourne à ce chapitre si nécessaire.

```
#### Fonctions
def encrypter(texteclair):
""" Crypte le message texteclair avec un chiffrement
de style Cesar (d-a) et renvoie le texte illisible.
"""
return texteclair #Pas de traitement pour le moment
```

7. **Nous pouvons passer à la section principale qui sera une section de test pour l'instant. Limitons-nous à mettre en place un appel à notre fonction en lui transmettant pour l'instant la constante MSG_TEST. Nous ajoutons deux instructions d'affichage.**

```
#### Main Section
textesecret = encrypter(MSG_TEST)
print(MSG_TEST)
print(textesecret)
```

Voici le programme complet dans son état initial.

Listing 7.1 : Texte source Cryptopy.py (version 1)

```
"""Cryptopy Version 1
Crypte et decrypte un texte en chiffrement Cesar.
Sait travailler avec le contenu d'un fichier texte.
Brendan Scott (VF par OE)
"""

#### Imports
import string

#### Constantes
JEUCAR = string.printable[:-5]
CARSUBSTI = JEUCAR[-3:]+JEUCAR[:-3]
MSG_TEST = "J'adore les Monty Python. Trop cool."

#### Fonctions
def encrypter(texteclair):
""" Crypte le message texteclair avec un chiffrement
Cesar (d-a) et renvoie le texte illisible. """

return texteclair #Pas de traitement pour le moment

#### Main Section
textesecret = encrypter(MSG_TEST)
print(MSG_TEST)
print(textesecret)
```

Lance l'exécution du programme pour vérifier qu'il n'y aucune erreur. Si tu en trouves, vérifie que le texte source correspond à celui du livre. Tu peux éventuellement recourir aux techniques de débogage que tu as découvertes à la fin du Projet 6.

Le type dictionnaire {}

Dans le projet précédent, nous avions utilisé la méthode standard `replace()` du type chaîne. Nous ne l'utiliserons pas dans ce projet pour deux raisons.

> » Ce n'est qu'une solution à court terme. Dans le projet pour parler en « leetien », nous n'avions que quelques caractères à substituer. Ici, nous devons remplacer tous les caractères de l'alphabet. Il faut toujours chercher à écrire du code source élégant. C'est ce que nous allons tenter de faire.

> » La solution du chapitre précédent n'est pas facile à généraliser. Si nous décidons de changer de nombre de décalages, il faut reprendre la fonction cryptage et la fonction de décryptage. Du travail en double !

Ce qu'il nous faut, c'est un type de données constitué de paires. En lui donnant par exemple la lettre `'d'`, nous récupérons la lettre `'a'`. Justement, il existe en Python un type de données permettant cela, il s'appelle le *dictionnaire*. Dans notre exemple, le `'d'` correspond à la *clé* et le `'a'` à la *valeur*. Pour créer une donnée du type dictionnaire, on utilise des {accolades}. Dans l'exemple suivant, nous créons un dictionnaire contenant un seul couple clé-valeur. Le nom de la variable est mentionné à gauche de l'opérateur d'affectation. L'unique article de ce dictionnaire est constitué d'une clé et d'une valeur, séparées par un signe deux-points.

```
>>> dico = {'d':'a'}
>>> dico['d']
'a'
```

Pour extraire une valeur du dictionnaire, il faut indiquer la clé qui lui correspond entre crochets (Attention : pas entre accolades !). Dans notre exemple, la clé est `'d'`. Pour récupérer la valeur correspondante, `'a'`, il faut écrire `dico['d']`.

Tu ne peux pas indiquer une valeur, sauf si le dictionnaire contient une clé identique. En général, cela provoque une erreur.

J'ai indiqué que j'utilisais le terme *élément* pour ce que contient une liste. Pour un dictionnaire, les contenus sont des *articles*. Un élément correspond à une seule chose qui est sa valeur alors qu'un article est un couple constitué d'une clé et d'une valeur.

Puisque le dictionnaire est un type de donnée Python, il est doté d'un certain nombre de méthodes. Souviens-toi que tout en Python est objet. Trois méthodes très utilisées sont `items()`, `keys()` et `values()`. Elles permettent d'obtenir des informations au sujet du contenu. Pour récupérer par exemple tous les articles d'un dictionnaire, il suffit d'écrire `dico.item()`. Tu peux essayer ces trois méthodes sur notre dictionnaire d'exemple, en vérifiant que tu comprends le résultat produit par chacune.

On peut créer un dictionnaire en le peuplant immédiatement avec plusieurs articles entre accolades, mais il faut que chaque article soit séparé du suivant par une virgule. Chacun des articles doit être écrit dans le format `cle:valeur`.

Pratiquement tous les types de données sont autorisés pour les valeurs. Pour les clés, on utilise en général soit une chaîne, soit un nombre entier. D'autres types sont possibles, mais nous nous cantonnerons à ces deux types pour l'instant.

Voici quelques essais dans l'interpréteur, avec des commentaires pour expliquer ce que fait chaque instruction :

```
>>> # Un dico vide
>>> dico_vide = {}
>>> # Dico déjà vu, un article
>>> dico = {'d':'a'}
>>> # Avec deux articles et la virgule
>>> dico = {'d':'a', 'e':'b'}
```

Méfiance : Python provoque une erreur d'exécution si tu tentes d'interroger une clé qui n'existe pas dans le dictionnaire :

```
>>> dico['f']
Traceback (most recent call last):
File "<pyshell#49>", line 1, in <module>
dico['f']
KeyError: 'f'
```

Le dictionnaire n'est pas un type de données immuable, c'est-à-dire que tu peux une fois qu'il existe ajouter des articles ou modifier ceux qui s'y trouvent déjà. Il suffit de fournir une valeur pour une clé existante ou pas. Dans la pratique, il arrivera rarement que tu fournisses directement une chaîne de caractères à insérer. Souvent, tu vas récupérer des données depuis une source externe, les stocker dans des variables temporaires puis fournir ces variables dans les opérations d'affectation au dictionnaire. Par convention, la variable temporaire pour la clé porte le nom `k` et la variable temporaire pour la valeur porte le nom `v`.

Illustrons cela :

```
>>> k = 'f'
>>> v = 'c'
>>> # Ajoutons un article
>>> dico[k] = v
>>> dico
{'e': 'b', 'd': 'a', 'f': 'c'}
```

Créons le dictionnaire de cryptage

Le type dictionnaire semble donc avoir d'excellents pouvoirs. Voyons comment en profiter. Nous allons créer un dictionnaire dans lequel chacun des caractères de JEU_CAR va devenir une clé associée à une valeur qui sera le caractère dans lequel il faudra le convertir.

Pour parvenir à nos fins, nous allons utiliser une nouvelle instruction standard portant le nom `enumerate()`. À partir d'une liste, `enumerate()` génère un nouvel objet numéroté.

Puisqu'une chaîne peut être traitée comme une liste, voici ce qu'il est possible de faire en demandant l'énumération d'une chaîne :

```
>>> [x for x in enumerate("Salut")]
[(0, 'S'), (1, 'a'), (2, 'l'), (3, 'u'), (4, 't')]
```

Cette instruction a généré une liste de couples constitués chacun d'un indice numérique correspondant à la position de chaque caractère dans la chaîne. Il est très pratique de combiner cette technique avec une boucle de répétition `for` pour balayer le contenu d'une liste et récupérer les indices des éléments.

```
>>> for i, c in enumerate("Salut"):
print(i, c)
(0, 'S')
(1, 'a')
(2, 'l')
(3, 'u')
(4, 't')
```

Dans cet exemple, chacun des éléments générés est décomposé en deux variables temporaires `i` et `c`. La première correspond à l'indice et l'autre au contenu qui est un caractère.

Nous allons nous servir de `enumerate()` pour créer l'association entre le jeu de caractères du texte clair et celui de la liste de substitution. Analysons les étapes de cette procédure :

1. **Nous choisissons un nom pour la variable dictionnaire.**
2. **Nous créons le dictionnaire vide.**

   Il reste ensuite à associer des valeurs à décliner.
3. **Nous profitons de la technique fondée sur la boucle suivante pour traiter chacun des caractères :**

   ```
   for i, k in enumerate(JEUCAR)
   ```
4. **Chaque caractère est stocké tour à tour dans la variable de travail**

k **(pour key, clé).**

Au début du projet, nous avions utilisé le nom c comme caractère. Dorénavant, nous utiliserons un k.

5. **La variable d'indice nommée i va permettre de récupérer le caractère correspondant dans CARSUBSTI et de le stocker dans la variable de travail v.**
6. **Il ne reste plus qu'à créer un article du dictionnaire en associant le contenu v à la clé k.**

Dans la pratique, j'ai ajouté ce traitement dans la section des constantes :

```
DICO_ENCRYP = {}
for i,k in enumerate(JEUCAR):
v = CARSUBSTI[i]
DICO_ENCRYP[k] = v
Les caractères de contrôle \t,\n,etc.,sontreportéstels quels:
for c in string.printable[-5:]: # Attention aux deux-points
DICO_ENCRYP[c] = c
```

Jointoyons avec join()

Pour crypter le message, nous devons changer chacun des caractères qui le constituent. Mais il y a un problème : les chaînes de caractères Python sont immuables, c'est-à-dire qu'on ne peut pas changer leur valeur. Il faut construire une nouvelle chaîne après conversion de l'ancienne, caractère par caractère.

Une technique simple pour créer une chaîne à partir d'une autre consiste à remplir la seconde avec la première, caractère par caractère :

```
>>> mastri = ""
>>> mastri
''
>>> for i in range(10):
mastri = mastri + str(i)
>>> mastri
'0123456789'
>>> mastri = ""
>>> mastri
''
>>> for i in range(10):
mastri = mastri + str(i)
>>> mastri
'0123456789'
```

Cette technique consistant à ajouter à la fin d'une chaîne les caractères d'une autre se nomme la *concaténation* (catena veut dire « chaîne » en latin). Je te conseille d'éviter cette technique en Python. Ce n'est que si tu n'as pas de solution ou lorsque tu as besoin de préparer une chaîne pour un affichage que tu t'en serviras. Pour les traitements sérieux, oublie cette approche. En effet, pour chaque tour de la boucle `for`, Python doit créer une nouvelle chaîne, y copier l'ancienne et ajouter le caractère suivant. Dans notre exemple, il y a création de 10 copies supplémentaires de la chaîne. Cela prend du temps et cela occupe l'espace en mémoire.

Dans Python, la bonne technique pour créer une nouvelle chaîne à partir d'une série de caractères isolés ou d'autres chaînes consiste à stocker tous les éléments dans une liste puis à joindre ces éléments pour obtenir une chaîne.

Je te propose le même exemple, en utilisant la méthode standard des chaînes nommée `join()`. Nous commençons par stocker tous les caractères dans une liste de travail qui porte le nom `accumule` puis nous raccordons tous les éléments au moyen de la méthode `join()` appliquée à une chaîne vide :

```
>>> accumule = []
```

```
>>> for i in range(10):
accumule.append(str(i))
>>> accumule
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
>>> ''.join(accumule)
'0123456789'
```

N'importe quelle chaîne convient pour recevoir le résultat de la jointure. Dans l'exemple, je propose une chaîne vide, mais on peut tout à fait fournir une chaîne peuplée. Son contenu sera inséré entre deux éléments de la liste.

Si nous prenons par exemple la chaîne littérale `'Jacadi'` pour la fusionner avec le contenu de la liste `accumule`, nous obtenons un résultat intéressant :

```
>>> ' Jacadi '.join(accumule)
'0 Jacadi 1 Jacadi 2 Jacadi 3 Jacadi 4 Jacadi 5 Jacadi 6
Jacadi 7 Jacadi 8 Jacadi 9'
```

Et maintenant à toi de tester : « Jacadi : fais des essais avec d'autres chaînes ! ». Note que la liste `accumule` aurait pu être une liste de chaînes.

Bref, dans le traitement par `join()`, la chaîne que tu fournis est insérée autant de fois que nécessaire entre deux éléments de la liste à laquelle elle est jointe.

Écrivons la fonction de cryptage

Nous avions créé l'amorce de la fonction de cryptage. Donnons-lui un peu de corps.

1. **Nous ne changeons pas le nom de fonction, mais nous pouvons éventuellement ajouter des informations à sa doc-chaîne.**
2. **Nous allons modifier sa ligne de tête (définition) pour qu'elle accepte un deuxième argument d'entrée, le dictionnaire de cryptage.**

```
def encrypter(texteclair, vardico_cryp):
```

3. **Le corps de la fonction doit être vide en dehors de la doc-chaîne. S'il y a des instructions, il faut les supprimer.**
4. **Nous implantons une liste vide pour stocker tous les caractères une fois chiffrés.**

```
textesecret = []
```

5. **Nous balayons ensuite le texte d'entrée pour récupérer chaque caractère après conversion en appliquant le dictionnaire reçu en argument :**

```
for k in texteclair:
v = vardico_cryp[k]
```

6. **Nous stockons successivement chacun des caractères chiffrés dans la liste de sortie.**

```
textesecret.append(v)
```

7. **Une fois que l'on sort de la boucle, nous utilisons `join()` pour obtenir la chaîne que nous renvoyons.**

```
return ''.join(textesecret)
```

8. **Dans le bloc principal, il faut retoucher l'appel à la fonction pour lui envoyer le second argument.**

```
textesecret = encrypter(MSG_TEST, DICO_ENCRYP)
```

9. **En revenant au début du texte source, dans la section des constantes, il n'y a plus qu'à ajouter la définition du dictionnaire de cryptage, vue un peu plus haut :**

```
DICO_ENCRYP = {}
for i,k in enumerate(JEUCAR):
v = CARSUBSTI[i]
DICO_ENCRYP[k] = v
# Les autres \t, \n etc. restent tels quels
for c in string.printable[-5:]: # Attention aux deux-points
DICO_ENCRYP[c] = c
```

Voici une nouvelle version du programme complet.

Listing 7.2 : Projet Cryptopy.py (version 2)

```
"""Cryptopy Version 2
Crypte et decrypte un texte en chiffrement Cesar.
Sait travailler avec le contenu d'un fichier texte.
Brendan Scott, (VF par OE)
"""

#### Imports
import string

#### Constantes
JEUCAR = string.printable[:-5]
CARSUBSTI = JEUCAR[-3:]+JEUCAR[:-3]
MSG_TEST = "J'adore les Monty Python. Trop cool."

# Generation du dictionnaire avec le jeu de caracteres
# (en clair).
DICO_ENCRYP = {}
for i,k in enumerate(JEUCAR):
    v = CARSUBSTI[i]
    DICO_ENCRYP[k] = v
# Les autres  \t, \n etc. restent tels quels
for c in string.printable[-5:]: # Attention aux deux-points
    DICO_ENCRYP[c] = c

#### Fonctions
def encrypter(texteclair, vardico_cryp):
    """ Crypte le message texteclair avec le dictionnaire
    fourni et renvoie le texte une fois rendu illisible. """
    textesecret = []
    for k in texteclair:
        v = vardico_cryp[k]
        textesecret.append(v)
    return ''.join(textesecret)

#### Main Section
textesecret = encrypter(MSG_TEST, DICO_ENCRYP)
print(MSG_TEST)
print(textesecret)
```

Voici ce qui devrait s'afficher si tu lances exécution :

```
>>> ============================== RESTART ====================
>>>
J'adore les Monty Python. Trop cool.
G$7alob¦ibp¦Jlkqv¦Mvqelk+¦Qolm¦9lli+
```

Une fois que tu as vérifié que cette version fonctionne bien, tu peux passer à la suite.

Notre fonction de décryptage

Il est inutile de chiffrer des messages si on ne peut pas les déchiffrer plus tard. Nous devons donc créer la fonction complémentaire pour restituer le message d'origine. Le traitement étant parallèle à celui du chiffrement, nous allons partir d'une copie de la première fonction.

1. **Sélectionne toutes les lignes de la fonction `encrypter()` et fait une copie, puis colle cette copie juste sous l'original.**
2. **Il faut changer le nom de la fonction en `decrypter`.**
3. **Il faut aussi changer le nom des deux arguments. Dorénavant, la fonction reçoit une chaîne illisible (`textesecret`) et travaille avec un dictionnaire de décryptage qui reste à définir (`vardico_decryp`) :**

```
def decrypter(textesecret, vardico_decryp):
```

4. **Tu peux modifier la doc-chaîne qui décrit ce que fait la fonction.**
5. **La fonction a besoin d'un dictionnaire de décryptage, qui sera l'inverse du premier.**

   Nous allons définir ce dictionnaire dans la section suivante.

6. **Pour tester la nouvelle fonction, nous ajouterons quelques lignes d'affichage un peu plus loin.**

Pour information, voici l'aspect de ma version de la fonction de décryptage :

```
def decrypter(textesecret, vardico_decryp):
""" Decrypte le message avec le dictionnaire
fourni et renvoie le texte une fois rendu lisible. """

texteclair = []
for k in textesecret:
    v = vardico_decryp[k]
    texteclair.append(v)
return ''.join(texteclair)
```

Créons le dictionnaire de décryptage

Pour effectuer le déchiffrage, nous travaillons en sens inverse : les `'a'` doivent redevenir des `'d'`. Puisque le point d'entrée du dictionnaire de cryptage s'écrit `DICO_ENCRYP['d']='a'`, celui du dictionnaire de décryptage doit s'écrire `DICO_DECRYP['a']='d'`.

Il suffit de travailler en sens inverse, de `v` vers `k` au lieu de `k` vers `v`. Puisque nous disposons déjà d'une boucle pour le premier dictionnaire, nous y ajoutons la création du dictionnaire de décryptage.

Voici le résultat de ce travail d'optimisation :

```
DICO_ENCRYP = {}
DICO_DECRYP = {}
for i,k in enumerate(JEUCAR):
    v = CARSUBSTI[i]
    DICO_ENCRYP[k] = v
    DICO_DECRYP[v] = k
# Les autres  \t, \n etc. restent tels quels
for c in string.printable[-5:]: # Attention aux deux-points
    DICO_ENCRYP[c] = c
    DICO_DECRYP[c] = c
```

Ces ajouts sont tous repris dans la prochaine version complète du projet, un peu plus loin.

Créons une section de test

Nous disposons maintenant des deux fonctions et des deux dictionnaires. Nous allons pouvoir lancer un test d'aller-retour. Nous allons crypter un message puis le décrypter pour vérifier que nous retrouvons bien ce que nous avions au départ. Nous allons même laisser Python vérifier que les deux messages sont strictement identiques grâce à l'opérateur de comparaison `==`.

Pour le test de la nouvelle fonction de décryptage, nous n'allons ajouter des instructions que dans la section principale en fin de programme.

1. **Il nous faut d'abord une phrase de test. Essayons d'abord avec la liste complète qui est dans JEUCAR.**

```
msg_testeur = JEUCAR
```

2. **Nous affichons le message tel qu'il est au départ.**

```
print(msg_testeur)
```

3. **Nous appelons la méthode `encrypter()` puis nous récupérons le résultat dans `textesecret`.**

```
textesecret = encrypter(msg_testeur, DICO_ENCRYP)
```

4. **Nous affichons le texte rendu illisible.**

```
print(textesecret)
```

5. **Nous appelons la fonction de décryptage et stockons le résultat dans `texteclair`.**

```
texteclair  = decrypter(textesecret, DICO_DECRYP)
```

6. **Nous affichons la nouvelle version du texte redevenu lisible.**

```
print(texteclair)
```

7. **Il ne reste plus qu'à afficher le résultat du test de comparaison logique entre les deux messages, la réponse pouvant être soit vraie, soit fausse :**

```
print(texteclair == msg_testeur)
```

Voici la nouvelle version de la section principale du projet. (Si tu avais d'autres lignes dans cette section, tu dois les neutraliser.)

```
#### Section principale et de test
msg_testeur = JEUCAR
textesecret = encrypter(msg_testeur, DICO_ENCRYP)
texteclair  = decrypter(textesecret, DICO_DECRYP)

print(msg_testeur)
print(textesecret)
print(texteclair)
print(texteclair == msg_testeur)
```

Lorsque tu lances l'exécution de cette version du programme, tu devrais obtenir quelque chose dans ce style :

```
>>> ============================== RESTART ====================
>>>
0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU-
VWXYZ!"#$%&'()*+,-./:;<=>?@[\]^_`{¦}~
}~ 0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU-
VWXYZ!"#$%&'()*+,-./:;<=>?@[\]^_`{¦
0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU-
VWXYZ!"#$%&'()*+,-./:;<=>?@[\]^_`{¦}~
True
```

Ne perds pas ton temps à comparer la première et la troisième ligne de ce qui s'affiche : tu as demandé au programme de faire la comparaison et d'indiquer par True ou False si le message de départ et le message redéchiffré étaient identiques.

Tu peux maintenant lancer des tests avec de vrais messages. Il suffit de changer la valeur de ces tests. En début de projet, j'avais proposé de créer la constante MSG_TEST. Il suffit de modifier sa valeur et de l'utiliser dans `msg_testeur` :

```
MSG_TEST = "J'adore les Monty Python. Trop cool."
```

Voici un exemple du résultat avec un message personnalisé :

```
>>> ============================== RESTART ====================
>>>
J'adore les Monty Python. Trop cool.
G$7alob¦ibp¦Jlkqv¦Mvqelk+¦Qolm¦9lli+
J'adore les Monty Python. Trop cool.
True
```

Pour conclure cette étape, voici le texte source complet du programme dans sa troisième version.

Listing 7.3 : Projet Cryptopy.py (Version 3)

```
"""Cryptopy Version 3
Crypte et decrypte un texte en chiffrement Cesar.
Sait travailler avec le contenu d'un fichier texte.
Brendan Scott (VF par OE)
"""

#### Imports
import string

#### Constantes
JEUCAR = string.printable[:-5]
CARSUBSTI = JEUCAR[-3:]+JEUCAR[:-3]
MSG_TEST = "J'adore les Monty Python. Trop cool."
# Generation du dictionnaire avec les deux jeux de caracteres
# (en clair et substitutions).
DICO_ENCRYP = {}
DICO_DECRYP = {}
for i,k in enumerate(JEUCAR):
    v = CARSUBSTI[i]
    DICO_ENCRYP[k] = v
    DICO_DECRYP[v] = k
# Les autres  \t, \n etc. restent tels quels
for c in string.printable[-5:]: # Attention aux deux-points
    DICO_ENCRYP[c] = c
    DICO_DECRYP[c] = c
#### Fonctions
def encrypter(texteclair, vardico_cryp):
    """ Crypte le message texteclair avec le dictionnaire
    fourni et renvoie le texte une fois rendu illisible. """
    textesecret = []
    for k in texteclair:
        v = vardico_cryp[k]
        textesecret.append(v)
    return ''.join(textesecret)
def decrypter(textesecret, vardico_decryp):
    """ Decrypte le message avec le dictionnaire
    fourni et renvoie le texte une fois rendu lisible. """
    texteclair = []
    for k in textesecret:
```

```
        v = vardico_decryp[k]
        texteclair.append(v)
    return ''.join(texteclair)

#### Section principale et de test
msg_testeur = JEUCAR               # ou MSG_TEST
textesecret = encrypter(msg_testeur, DICO_ENCRYP)
texteclair  = decrypter(textesecret, DICO_DECRYP)

print(msg_testeur)
print(textesecret)
print(texteclair)
print(texteclair == msg_testeur)
```

Rendons le programme utile

Nous pouvons maintenant rendre notre programme polyvalent, c'est-à-dire capable de crypter ou de décrypter un message que tu saisis au clavier. Pour l'instant, le message de test était incrusté dans le code source et donc non modifiable par l'utilisateur.

Une donnée qui est indiquée directement dans le texte source d'un programme (au lieu d'être le résultat d'une saisie par l'utilisateur ou d'une entrée de données) est une donnée « codée en dur » (hard coded).

Pour pouvoir tester avec différents messages, la solution la plus simple consiste à demander à l'utilisateur de saisir un message au moyen de la fonction `raw_input()`. Mais il y a un petit souci : le programme sait crypter et décrypter, mais il est incapable de dire si ce qui est saisi est du texte clair ou du texte secret. (Ceci dit, bonne chance pour saisir un texte crypté.)

La solution consiste à réaliser les deux traitements, cryptage et décryptage. L'utilisateur choisira la phrase qui est dans le format qu'il désire.

Analysons les retouches qui vont permettre d'atteindre cet objectif.

1. **Nous choisissons un message d'invite pour la saisie. Nous allons stocker dans une variable `message` la chaîne de caractères qui aura été saisie :**

```
message = raw_input("Message qui doit devenir secret :\n")
```

2. **Nous allons appeler les deux fonctions pour obtenir un texte en clair et un texte crypté :**

```
textesecret = encrypter(message, DICO_ENCRYP)
texteclair  = decrypter(message, DICO_DECRYP)
```

3. **Nous affichons une ligne de légende puis le message crypté.**
4. **De même, nous affichons une autre ligne de légende et le texte décrypté (mais il n'est pas lisible ; je te laisse chercher pourquoi).**
5. **Pour faire joli, j'ajoute au début une ligne de titre qui rappelle la version du projet.**

Tu peux supprimer les lignes de test de la version précédente, ou bien les neutraliser par mise en commentaires. Voici la seule section principale de la nouvelle version du projet.

Listing 7.4 : Extrait du projet Cryptopy.py (version 4)

```
#### Section principale et de test
# msg_testeur = JEUCAR
# textesecret = encrypter(msg_testeur, DICO_ENCRYP)
# texteclair  = decrypter(textesecret, DICO_DECRYP)
# print(msg_testeur)
# print(textesecret)
```

```
# print(texteclair)
# print(texteclair == msg_testeur)

print("*** Projet Cryptopy, version 4 ***\n")
message = raw_input("Message qui doit devenir secret :\n")
textesecret = encrypter(message, DICO_ENCRYP)
texteclair  = decrypter(message, DICO_DECRYP)
print("Message en sortie de encrypter() : ")
print(textesecret)
##
print("Message en sortie de decrypter() : ")
print(texteclair)
```

Tu as remarqué le caractère spécial `\n` à la fin du message d'invite. Il permet de faire saisir le message sur une nouvelle ligne. Voici un exemple de l'exécution :

```
>>> ============================== RESTART ==================
>>>
*** Projet Cryptopy, version 4 ***

Message qui doit devenir secret :
aaaaaaaeeeeeiiiiiioooooooooooottttttttttt
Message en sortie de encrypter() :
7777777bbbbbfffffflllllllllllqqqqqqqqqq
Message en sortie de decrypter() :
ddddddhhhhhlllllrrrrrrrrrrrrwwwwwwwwww
```

Pour confirmer que notre programme est devenu symétrique, relance l'exécution, mais cette fois-ci, ne saisis rien. Copie dans l'interpréteur la chaîne en sortie de `encrypter()` et colle-la devant l'invite puis valide. Tu dois retrouver ce que tu avais saisi :

```
>>> ============================== RESTART ==================
>>>
*** Projet Cryptopy, version 4 ***

Message qui doit devenir secret :
7777777bbbbbfffffflllllllllllqqqqqqqqqq
Message en sortie de encrypter() :
444444488888ccccccciiiiiiiiiiinnnnnnnnnn
Message en sortie de decrypter() :
aaaaaaaeeeeeiiiiiioooooooooooottttttttttt
```

Si tout se passe bien, tu as récupéré le message initial. Bien sûr, notre version affiche inutilement certaines versions du message dont tu n'as pas besoin. Si nous avions une interface graphique, nous aurions mis en place deux boutons : un pour crypter, l'autre pour décrypter. Mais cela suppose de découvrir la programmation avec une interface graphique, ce que je n'aborde pas dans ce livre d'initiation.

Exploitons un fichier texte

Pour éviter de devoir saisir des messages dans notre programme, ou même de faire d'incessants copier/coller, il y a une solution qui permet au passage de découvrir le très important domaine de la gestion des fichiers. Demandons à Python de lire le texte à crypter ou décrypter depuis un fichier. Les explications vont être un peu plus longues, parce que la gestion des fichiers est un sujet en soi.

Nous n'allons travailler qu'avec des fichiers au format texte pur (extension *.txt* sous Windows), c'est-à-dire des fichiers qui ne contiennent que des lettres et des chiffres, sans aucune indication de style ni mise en forme. Inutile d'essayer de faire lire par notre projet un fichier de traitement de texte Word ou autre.

N. d. T. : nous ne gérerons pas les lettres accentuées dans nos fichiers. La gestion des accents par Python est un autre sujet que je n'aborde pas ici, d'autant qu'elle est différente en Python 2 et en Python 3. Sache

seulement qu'il faut s'intéresser à Unicode pour commencer.

Ouvrir, écrire et fermer un fichier

Commençons en douceur par de petits essais dans l'interpréteur Shell de IDLE. Saisis bien l'instruction comme indiqué ci-après :

```
>>> ficow = open('tempo.py', 'w')
```

Cette instruction utilise la fonction standard d'ouverture `open()`. Elle demande de chercher et d'ouvrir le fichier mentionné comme premier argument et de l'ouvrir dans le mode écriture, comme stipulé par le second argument `'w'`, le W signifiant Write (écrire). Le résultat est stocké dans la variable située en membre gauche. Si le fichier désigné existe déjà, son contenu est détruit, pour être remplacé par ce que nous allons y écrire dans les instructions suivantes.

Attention : si tu demandes à Python de réaliser une action qui peut supprimer des données, il va l'effectuer sans te demander confirmation. Il n'y aura aucune sauvegarde de sécurité. Tu dois donc être très concentré lorsque tu écris des instructions qui manipulent des fichiers dans Python. Si tu as le moindre doute, il faut faire un test avec un fichier dont tu as fait une copie.

Pour en savoir plus, demande de l'aide dans l'interpréteur avec la commande `help(open)`. La réponse devrait être que cette fonction permet d'ouvrir un fichier du type `file`, et de renvoyer un *objet fichier*. Qu'est-ce que c'est que cet objet fichier ? Ce n'est pas le fichier lui-même, ni même son contenu. C'est une sorte d'identifiant qui permet au système de ton ordinateur d'accéder au fichier. C'est un peu comme une passerelle entre les instructions du programme et les données sur le disque. Tu ne dois pas être étonné qu'il s'agisse ici aussi d'un objet, et tu peux même utiliser `dir()` pour obtenir la liste de ses attributs, et notamment de ses méthodes.

Si tu saisis la commande `dir(ficow)`, donc avec l'objet qui a reçu l'identifiant lorsque l'ouverture a réussi, tu verras apparaître la liste de ses attributs, parmi lesquels les quatre méthodes `write()`, `read()`, `readlines()` et `close()`.

Lorsque tu as terminé d'utiliser un fichier, il faut penser à appeler sa méthode `close()` pour le refermer.

Voyons comment écrire quelques données dans le fichier que nous avons ouvert. Nous commençons par définir une variable chaîne. Attention à l'écriture de cette valeur chaîne. Il y a une chaîne dans la chaîne :

```
>>> pytxt = "print('Bonjour depuis le fichier.')\n"
```

Nous écrivons deux fois la variable dans le fichier et nous n'oublions pas de le refermer :

```
>>> ficow.write(pytxt)
>>> ficow.write(pytxt)
>>> ficow.close()
```

Notre chaîne se trouve au départ dans la variable `pytxt`, et se termine par le caractère de saut de ligne `\n`. La chaîne est stockée dans le fichier au moyen de la méthode `write()` qui appartient à notre objet fichier `ficow`. Tu remarques que nous écrivons deux fois la même ligne, car cela nous servira plus tard. Les deux lignes sont bien séparées grâce à la présence du saut de ligne. Nous terminons en refermant le fichier.

Ouvrir et lire un fichier

Pour pouvoir relire le fichier que nous venons de créer et remplir, nous devons ouvrir à nouveau le fichier, mais dans un autre mode, le mode lecture, « read ».

Je rappelle que lorsque tu ouvres un fichier en mode écriture `w`, tu détruis son contenu actuel. Il faut donc faire bien attention. Il faut changer la lettre `w` qui est le deuxième argument en lettre `r`. Pour montrer que

j'ai ouvert le fichier en mode lecture, je donne maintenant à l'objet fichier un nom qui se termine par cette lettre (`ficor`).

```
>>> ficor = open('tempo.py', 'r')
```

La seule différence entre cette instruction et la précédente est cette lettre `r` à la place du `w`. Tu as deviné que ce `'r'` signifier read, lire. Bien sûr, quand tu ouvres le fichier dans ce mode lecture, tu ne détruis pas son contenu (heureusement).

Une fois que le fichier est ouvert, nous pouvons utiliser la méthode `read()` de notre objet fichier `ficor` :

```
>>> print(ficor.read())
print('Bonjour depuis le fichier.')
print('Bonjour depuis le fichier.')
```

La méthode `read()` de notre objet de type `file` lit en une fois la totalité du contenu. Mais les objets fichier possèdent un mécanisme permettant de savoir où en est la lecture par rapport à la fin du fichier. Dans notre cas, puisque nous sommes déjà arrivés à la fin, il ne se passe rien si nous tentons une nouvelle opération de lecture :

```
>>> print(ficor.read())
```

Pour pouvoir relire le fichier, il faut le fermer puis le rouvrir :

```
>>> ficor.close()
>>> ficor = open('tempo.py', 'r')
>>> print(ficor.read())
print('Bonjour depuis le fichier.')
print('Bonjour depuis le fichier.')
>>> ficor.close()
```

On a plus souvent besoin de lire le contenu d'un fichier une ligne à la fois, plutôt que tout d'un coup. Il suffit d'utiliser pour cela la méthode `readlines()` qui ne lit qu'une ligne (malgré le pluriel dans son nom). Dans cet essai, je referme le fichier au cas où. Il n'y a pas d'erreur s'il l'est déjà.

```
>>> ficor.close()
>>> ficor2 = open('tempo.py', 'r')
>>> compteur = 0
>>> for ligne in ficor2.readlines():
        compteur = compteur + 1
        print("Ligne " + str(compteur) + " :" + ligne)

Ligne 1 :print('Bonjour depuis le fichier.')

Ligne 2 :print('Bonjour depuis le fichier.')
```

Tu remarques que dans l'affichage les lignes sont séparées par une seconde ligne vide, puisqu'il y a, en plus du saut de ligne de l'instruction d'affichage, celui provoqué par le caractère `\n` de la fin de la chaîne. Tu peux facilement vérifier que ce caractère de contrôle a été récupéré puisqu'il suffit de demander à voir le contenu de la variable ligne. Elle contient encore la dernière ligne lue :

```
>>> ligne
"print('Bonjour depuis le fichier.')\n"
```

Exécutons notre fichier texte !

J'ai fait d'une pierre deux coups en te demandant de stocker dans le fichier des lignes qui sont en fait des instructions Python tout à fait valables. Nous allons pouvoir vérifier que le fichier est un vrai fichier de script Python (le nom est d'ailleurs doté de son extension *.py*). Charge le fichier dans la fenêtre d'édition de

IDLE par File/Open. Dans l'exemple, j'avais choisi comme nom *tempo.py*. Tu peux lancer l'exécution du fichier sans rien y changer :

```
>>> ============================== RESTART ==================
>>>
Bonjour depuis le fichier.
Bonjour depuis le fichier.
```

Le résultat est exactement le même que si tu avais saisi les deux lignes de code dans l'éditeur et les avais enregistrées dans ce fichier.

Plus de confort avec with

Quand tu utiliseras beaucoup les fichiers, tu vas vite te rendre compte qu'il est stressant de ne pas oublier de refermer les fichiers quand tu n'en as plus besoin. Heureusement, Python propose un mot réservé qui permet d'automatiser cette fermeture (et de faire d'autres choses). C'est le mot `with` (avec). Il suffit de mentionner l'appel à la méthode d'ouverture `open()` après ce mot `with` puis d'ajouter l'autre mot réservé `as` et enfin le nom de la variable temporaire qui doit recevoir l'objet fichier avec lequel tu vas manipuler le fichier (par exemple `ficor`). Allez, un exemple dans l'interpréteur :

```
with open('tempo.py', 'r') as ficor:
print(ficor.read())

print('Bonjour depuis le fichier.')
print('Bonjour depuis le fichier.')
```

Étudions la première ligne. Je demande à Python d'ouvrir un fichier portant le nom *tempo.py* en mode lecture (r) puis de stocker l'objet qui résulte de cette ouverture dans la variable portant le nom `ficor`. Dans la ligne qui dépend de la première, je demande de lire d'un coup la totalité du contenu du fichier. Une fois qu'il est sorti du bloc dépendant de `with`, Python sait que tu n'as plus besoin du fichier et se charge de le refermer. Tout en bas de notre test, j'interroge l'objet fichier. La réponse me confirme que le fichier est bien refermé, alors que nous n'avons pas utilisé la méthode `close()` pour le faire.

Nous pouvons utiliser `with` pour ouvrir plusieurs fichiers à la fois. N'essaie pas de saisir cet exemple avec la ligne de commande car il ne fonctionnera pas :

```
with open(nomfic1, 'r') as objet_fic1,
open(nomfic2, 'r') as objet_fic2:
#actions avec objet_fic1
#actions avec objet_fic2
```

L'exemple précédent permet de comprendre la structure des lignes d'instruction réelles, mais il ne fonctionne pas. Il s'agit de *pseudo-code* ou de *code formel*.

Voici un bref exemple dans l'interpréteur. Pour ouvrir deux fichiers, nous en créons d'abord un deuxième mais détruisons son contenu antérieur. Ce fichier se nomme *tempo2.txt*. Méfie-toi.

```
with open('tempo2.txt', 'w') as ficow:
ficow.write('Truc Bidule')
```

Après cette instruction, le fichier est refermé. Nous rouvrons les deux en lecture :

```
>>> with open('tempo2.txt', 'r') as ficor2,
open('tempo.py', 'r') as ficor1:
print(ficor2.read())
print(ficor1.read())
Truc Bidule
print('Bonjour depuis le fichier.')
print('Bonjour depuis le fichier.')
```

C'est principalement avec les fichiers que tu utiliseras le mot réservé `with`. Dorénavant, utilise toujours cette technique, sauf si tu as une bonne raison de faire autrement.

Cryptons et décryptons le contenu d'un fichier

Au lieu de devoir saisir le message à crypter ou décrypter avec la ligne de commande, voyons comment produire une version utile du projet : nous allons lire les données depuis un fichier, les crypter et les restocker dans un autre fichier.

Voici d'abord les étapes de notre analyse fonctionnelle :

1. **Nous choisissons un nom pour le fichier qui va contenir le texte en clair qu'il faudra crypter. Je propose** *cryptopy_in.txt* **(*in* signifie entrée).**

   Pour l'instant, le nom du fichier ne pourra pas être changé. Il sera codé dans le code source.

2. **Nous créons fichier en y stockant quelques données.**
3. **Il faut ensuite choisir le nom de la constante dans laquelle nous allons stocker le nom du fichier d'entrée : NOMFIC_IN.**

   Pour traiter un autre fichier, il suffira de changer cette valeur.

4. **Nous choisissons aussi un nom pour le fichier dans lequel sera stocké le résultat après cryptage. Je propose** *cryptopy_out.txt* **(*out* signifie sortie).**
5. **De même, nous choisissons le nom de la constante qui va contenir le nom du fichier de sortie : NOMFIC_OUT.**
6. **Nous pouvons ensuite ouvrir le fichier d'entrée.**
7. **Nous lisons le texte qu'il contient, a priori non crypté.**
8. **Nous pouvons alors refermer le fichier d'entrée.**
9. **Nous procédons au cryptage du contenu.**
10. **Nous affichons pour contrôle le résultat crypté.** Cela nous sera utile pour la phase de test.
11. **Nous pouvons alors ouvrir le fichier de sortie.**
12. **Nous y écrivons le texte crypté.**
13. **Nous refermons le fichier de sortie.**

Les étapes 6 à 13 vont remplacer toute la section principale de la version actuelle du projet. Tu pourras donc neutraliser ou supprimer les lignes correspondantes.

Constantes de nom de fichier et données de test

Pour les deux constantes qui vont contenir le nom du fichier d'entrée et celui de sortie, j'ai choisi NOMFIC_IN et NOMFIC_OUT.

Procédons à un petit essai dans la fenêtre de l'interpréteur. Nous créons la constante pour le nom de fichier d'entrée puis nous ouvrons ce fichier pour y stocker une petite phrase :

```
>>> NOMFIC_IN = "cryptopy_in.txt"
>>> with open(NOMFIC_IN, 'w') as fic_lect:
fic_lect.write("Je ne suis qu'un test.")
```

Aucun message ne te confirme que tout s'est bien passé. N'hésite pas à utiliser l'explorateur ou le Finder pour vérifier que le fichier a bien été créé.

La librairie standard nommée `os.path` contient une fonction nommée `exists()` qui permet de savoir si un fichier existe ou non. Pour pouvoir t'en servir, il faut d'abord ajouter une directive d'importation `import os.path` puis appeler la fonction de la façon suivante :

```
os.path.exists(<Chemin_Fichier>)
```

Tu remplaces Chemin_Fichier par la chaîne permettant de désigner le fichier à partir de la racine du disque ou du système de fichiers. Si le fichier est dans le même dossier, le nom du fichier seul suffit.

Ouvrons le fichier et décryptons

Lorsqu'il s'agit de manipuler un fichier, le seul point délicat est de ne pas confondre le nom du fichier et le nom de l'objet fichier. Le nom du fichier est celui que tu donnes comme argument à la méthode `open()` pour l'ouvrir. Le nom de l'objet fichier est l'identifiant que la méthode te renvoie si elle réussit. Dans mon exemple, les deux noms des objets fichiers sont `fic_lect` et `fic_ecri`, donc en lecture et en écriture. (Pas d'accents dans les noms de variables.)

Les différentes étapes de la procédure qui suit se retrouvent dans le listing complet que je soumets ensuite. Nous commençons par créer les deux constantes dans la section correspondante pour définir les noms des fichiers en entrée et en sortie :

```
NOMFIC_IN  = "cryptopy_in.txt"
NOMFIC_OUT = "cryptopy_out.txt"
```

Voici la suite de la procédure :

1. **Si nécessaire, tu peux neutraliser par mise en commentaires l'ancienne section des entrées/sorties, dans la partie principale en bas.**
2. **Dans cette section principale, nous passons à l'ouverture du fichier d'entrée et à la lecture de son contenu. Nous stockons ce qui a été lu dans la variable nommée message.**

```
with open(NOMFIC_IN,'r') as fic_lect:
message = fic_lect.read()
```

3. **Nous pouvons alors appeler la fonction de cryptage et récupérer le résultat de son travail dans la variable `textesecret`.**

```
textesecret = encrypter(message, DICO_ENCRYP)
```

4. **Il ne reste plus qu'à ouvrir le fichier de sortie pour y stocker le texte crypté.**

```
with open(NOMFIC_OUT,'w') as fic_ecri:
fic_ecri.write(textesecret)
```

Voici la nouvelle version du code source complet du projet.

Listing 7.5 : Projet Cryptopy.py (version 5)

```
"""Cryptopy Version 5 (Fichiers)
Crypte et decrypte un texte en chiffrement Cesar.
Sait travailler avec le contenu d'un fichier texte.
Brendan Scott (VF par OE)
"""

#### Imports
import string

#### Constantes
JEUCAR = string.printable[:-5]
```

```
CARSUBSTI = JEUCAR[-3:]+JEUCAR[:-3]

# Generation du dictionnaire avec les deux jeux de caracteres
# (en clair et substitutions).
DICO_ENCRYP = {}
DICO_DECRYP = {}
for i,k in enumerate(JEUCAR):
    v = CARSUBSTI[i]
    DICO_ENCRYP[k] = v
    DICO_DECRYP[v] = k
# Les autres  \t, \n etc. restent tels quels
for c in string.printable[-5:]: # Attention aux deux-points
    DICO_ENCRYP[c] = c
    DICO_DECRYP[c] = c

MSG_TEST = "J'adore les Monty Python. Trop lol."
NOMFIC_IN  = "cryptopy_in.txt"
NOMFIC_OUT = "cryptopy_out.txt"

#### Fonctions
def encrypter(texteclair, vardico_cryp):
    """ Crypte le message texteclair avec le dictionnaire
    fourni et renvoie le texte une fois rendu illisible. """
    textesecret = []
    for k in texteclair:
        v = vardico_cryp[k]
        textesecret.append(v)
    return ''.join(textesecret)

def decrypter(textesecret, vardico_decryp):
    """ Decrypte le message textesecret avec le dictionnaire
    fourni et renvoie texteclair. """
    texteclair = []
    for k in textesecret:
        v = vardico_decryp[k]
        texteclair.append(v)
    return ''.join(texteclair)

#### Section principale main
## message = raw_input("Message qui doit devenir secret:\n")
## textesecret = encrypter(message, DICO_ENCRYP)
## texteclair  = decrypter(message, DICO_DECRYP)
## print("Message apres cryptage :")
## print(textesecret)
##
## print("Message redevenu lisible :")
## print(texteclair)
print("*** Projet Cryptopy, version 5 ***\n")
with open(NOMFIC_IN,'r') as fic_lect:
message = fic_lect.read()

textesecret = encrypter(message, DICO_ENCRYP)
print(textesecret)           # Pour tester

with open(NOMFIC_OUT,'w') as fic_ecri:
fic_ecri.write(textesecret)
```

Tu vois que les étapes 6 à 13 sont réalisées avec les six lignes ci-dessus. Incroyable non ? Voici ce que l'on voit à l'exécution avec le message de test que contenait le fichier d'entrée :

```
>>> ============================== RESTART ====================
>>>
*** Projet Cryptopy, version 5 ***
```

```
Gb¦kb¦prfp¦nr$rk¦qbpq+
```

Cette chaîne cryptée apparaît dans la fenêtre de l'interpréteur à cause de l'instruction d'affichage `print()`. Pour vérifier qu'elle a également été écrite dans le fichier de sortie, nous pouvons demander dans l'interpréteur d'ouvrir le fichier et d'en afficher le contenu :

```
>>> NOMFIC_OUT
'cryptopy_out.txt'
>>> with open(NOMFIC_OUT, 'r') as fic_sortie_r:
print(fic_sortie_r.read())

Gb¦kb¦prfp¦nr$rk¦qbpq+
```

J'ai d'abord vérifié que la constante de nom de fichier était toujours définie.

Notre programme devient un module !

Ne serait-il pas pratique de pouvoir faire des cryptages et décryptages directement depuis l'interpréteur Shell, sans avoir à exécuter le programme et modifier le contenu d'un fichier source ? Pour y arriver, il faudrait en quelque sorte pouvoir rendre utilisables nos fonctions sur la ligne de commande, en les exportant depuis le fichier de script *Cryptopy.py*. Sache que cela est tout à fait possible. Tu peux appliquer la directive `import` à ton propre programme.

Importe ton module

Magie, magie : va dans la fenêtre de l'interpréteur Shell et saisis la directive suivante :

```
>>> import Cryptopy
Gb¦kb¦prfp¦nr$rk¦qbpq+
```

La syntaxe est exactement la même que lorsque tu as besoin d'importer un module standard de la librairie Python. Peu importe quel programme Python, la seule limite est que Python doit trouver le fichier correspondant (et le nom de fichier ne doit pas commencer par un chiffre). Dans notre cas, cela suppose que si tu as créé toutes les versions des projets dans un autre dossier que celui dans lequel est installé Python, il faut en faire une copie dans ce dossier de base, par exemple, *C : \Python27*. Une fois que le code source est trouvé à cet endroit, tu peux accéder à toutes les fonctions et à toutes les constantes qui sont définies dans ton fichier source.

Définissons donc un message de test :

```
>>> txi = """Je me demande si on peut utiliser les fonctions
de Cryptopy.py depuis la ligne de commande Shell ?"""
```

Nous pouvons maintenant essayer d'utiliser la fonction `encrypter()` qui est définie dans notre fichier de projet. L'astuce est de donner comme préfixe de nom de module le nom du fichier source sans l'extension *.py*, comme ce serait le cas avec un module de la librairie standard :

```
>>> txs = Cryptopy.encrypter(txi, Cryptopy.DICO_ENCRYP)
```

Si tu demandes l'affichage du résultat de cet appel, tu vois que le travail a été fait :

```
>>> txs
'Gb¦jb¦abj7kab¦pf¦lk¦mbrq¦rqfifpbo¦ibp¦clk9qflkp¦ab¦zovmqlmv+mv¦
abmrfp¦i7¦ifdkb¦ab¦9ljj7kab¦Pebii¦<
```

Nous pouvons même utiliser la fonction `decrypter()` pour récupérer le message initial :

```
>>> print(Cryptopy.decrypter(txs, Cryptopy.DICO_DECRYP))
```

```
Je me demande si on peut utiliser les fonctions de Cryptopy.py
depuis la ligne de commande Shell ?
```

Tu peux donc utiliser toutes les fonctions que contient ton module depuis l'interpréteur Python. Tu peux même profiter du mécanisme d'aide à la saisie par `help()` pour insérer les noms des constantes des fonctions. Et puisque tu peux importer le module dans l'interpréteur, tu peux également l'importer dans un autre programme Python.

Mais pour que la directive `import` fonctionne, il faut que le fichier du module désiré se trouve dans l'un des deux endroits suivants :

> » le répertoire de travail ou répertoire courant du script depuis lequel tu veux importer l'autre script.
>
> » un des répertoires/dossiers mentionnés dans la liste de la variable d'environnement nommée PYTHONPATH, mais cette variable est vide au départ. Nous n'en parlerons pas ici.

Pour simplifier, tu peux installer le code source de ton projet et de tous les modules dont il a besoin dans le dossier standard de Python.

Tu as peut-être remarqué un petit effet secondaire gênant. Lorsque tu importes le module, il y a un ou deux messages qui apparaissent. Si on décrypte le dernier qui est crypté, on reconnaît le contenu du fichier d'entrée (*cryptopy_in.txt*). Dans la version actuelle du programme, nous demandons l'affichage de ce texte dans la section principale. En important le programme tel quel, Python a d'abord exécuté la totalité du contenu du module, et donc aussi les instructions d'affichage. C'est ce que nous voulions en utilisant le projet indépendamment, mais il ne faut pas que ce genre d'affichage vienne parasiter le fonctionnement d'un programme qui utilise certaines fonctions du module. Les autres programmes doivent exploiter les fonctions, pas voir leur affichage envahi par celui du module.

Python permet de détecter que le code source est utilisé en tant que module importé ou en tant que programme autonome. Le nom du programme qui s'apprête à démarrer est stocké par Python dans une variable système qui porte le nom de `__name__` (oui, c'est une variable doublesoul). Lorsque le programme est démarré de façon autonome, cette variable contient la valeur `__main__`. C'est Python qui se charge de procéder à cette copie.

Tu peux donc utiliser la variable `__name__` dans un test pour savoir si le code est exécuté en autonomie ou en tant que module. Il suffit d'ajouter une structure conditionnelle avec `if`.

En langage Python, il arrive fréquemment que l'on veuille isoler une série d'instructions en les faisant dépendre d'une instruction `if` pour ne pas les exécuter si le programme est utilisé comme module avec une directive `import` :

```
>>> if __name__ == "__main__":
        print("Je ne suis pas un module !")
Je ne suis pas un module !
```

Nous allons donc aménager notre projet pour qu'il puisse être importé par un autre script en devenant muet.

1. **Dans la section principale, nous ajoutons au début une instruction `if` pour comparer `_ _name_ _` et `_ _main_ _`.**
2. **Il ne reste plus qu'à rendre toutes les lignes du bloc principal réalisant les entrées et sorties fichier dépendantes de cette instruction conditionnelle en les décalant de quatre espaces.**

Voici la nouvelle section principale de notre projet une fois l'instruction conditionnelle mise en place pour qu'aucune instruction ne soit exécutée si le projet est utilisé en tant que module par un autre.

Listing 7.6 : Extrait du projet Cryptopy.py (version 6)

```
if __name__ == "__main__":
    # Lignes neutres omises ici.
```

```
    print("*** Projet Cryptopy, version 6 ***\n")
    with open(NOMFIC_IN,'r') as fic_lect:
        message = fic_lect.read()

    txt_resultat = encrypter(message, DICO_ENCRYP)

    with open(NOMFIC_OUT,'w') as fic_ecri:
       fic_ecri.write(txt_resultat)
```

Dans toute la suite, je fais toujours référence à cette partie du programme en tant que section principale (c'est la section de la fonction invisible `main`). Je te conseille comme moi d'ajouter une ligne de commentaires servant de titre pour voir le début de cette section.

Si tu importes ce programme, Python charge toutes les fonctions et constantes sans rien exécuter de la section principale :

```
>>> import Cryptopy        # Aucun message ne doit s'imprimer
>>>
```

Le texte comparant `__name__` avec `__main__` est une technique très pratique pour utiliser un script sans avoir à faire un copier/coller ou une copie du fichier source. Il devient facilement réutilisable.

L'avantage, c'est que cette pratique permet de centraliser les fonctions. Un seul fichier sert à plusieurs projets. Si une erreur est détectée, il n'y a qu'un endroit à corriger. Tous les programmes qui importent ce module profiteront automatiquement de la correction ou des améliorations que tu auras apportées.

Crypter, mais aussi décrypter

Pour l'instant, le projet ouvre un fichier et crypte son contenu. Il serait intéressant de permettre aussi le décryptage. Mais comment choisir entre crypter et décrypter lorsque je lance l'exécution ?

Nous allons y parvenir très facilement en mettant en place un aiguillage, sous forme d'une constante. Nous pourrons exécuter une branche conditionnelle ou une autre en fonction de la valeur de cette constante.

Voici les différentes étapes de cette dernière procédure du projet. Nous n'allons travailler que dans la section principale.

1. **Nous devons d'abord définir une constante qui sera de type logique, c'est-à-dire qu'elle pourra posséder soit la valeur True, soit la valeur False.**

   Lorsqu'elle contiendra la valeur True, nous crypterons les données lues depuis le fichier. Lorsqu'elle contiendra la valeur False, nous décrypterons le contenu. Mets en place cette définition constante tout au début de la section principale. Nous pourrions aussi l'ajouter dans la section des constantes au début du fichier.

```
ENCRYPT = True # Si True crypte vers OUT, si False crypte
vers IN
```

2. **Nous changeons ensuite toutes les références au nom de variable `textesecret` en `txt_resultat`, parce que le nom `textesecret` n'est plus approprié lorsque nous décryptons.**
3. **Entre l'instruction de lecture de fichier et celle d'écriture du fichier de sortie, nous mettons en place un bloc conditionnel avec `if`.**

L'instruction va tester la constante et choisir de crypter ou de décrypter le contenu du fichier d'entrée selon sa valeur True ou False.

```
if ENCRYPT:
    txt_resultat = encrypter(message, DICO_ENCRYP)
else:
    txt_resultat = decrypter(message, DICO_DECRYP)
```

Voici la section principale du programme dans la nouvelle version une fois cette instruction conditionnelle mise en place.

Listing 7.7 : Extrait du projet Cryptopy.py (version 7)

```
if __name__ == "__main__":
## message = raw_input("Message qui doit devenir secret:\n")
## textesecret = encrypter(message, DICO_ENCRYP)
## texteclair  = decrypter(message, DICO_DECRYP)
## print("Message apres cryptage :")
## print(textesecret)
##
## print("Message redevenu lisible :")
## print(texteclair)
    print("*** Projet Cryptopy, version 7 ***\n")
    ENCRYPT = True # True crypte vers OUT, False crypte vers
IN
    with open(NOMFIC_IN,'r') as fic_lect:
            message = fic_lect.read()

    if ENCRYPT:
        txt_resultat = encrypter(message, DICO_ENCRYP)
    else:
        txt_resultat = decrypter(message, DICO_DECRYP)

    print(txt_resultat) # Pour tester. Enlever ensuite.

    with open(NOMFIC_OUT,'w') as fic_ecri:
       fic_ecri.write(txt_resultat)
```

Dans la ligne qui donne sa valeur à la constante ENCRYPT, il faut donner la valeur False si le fichier d'entrée est déjà crypté. Si tu choisis la valeur False, tu peux faire un essai en plaçant d'abord un peu de texte crypté dans le fichier d'entrée.

Tu peux insérer les données depuis l'interpréteur comme je l'ai expliqué plus haut, ou bien ouvrir le fichier dans un éditeur, ou dans IDLE, écrire et copier/coller une phrase cryptée. Quand tu vas enregistrer le fichier, vérifie bien que tu choisis le type de fichier **.txt* ou Tous les fichiers dans la liste des types de la boîte d'enregistrement. De même, il faut choisir ce type dans la boîte d'ouverture pour que le fichier apparaisse dans le volet de droite.

Revoici le texte crypté que j'avais inséré dans le fichier d'entrée pour tester :

```
Gb¦kb¦prfp¦nr$rk¦qbpq+
```

Et voici le résultat de l'exécution avec ce texte :

```
>>> ============================== RESTART ==================

>>>
*** Projet Cryptopy, version 7 ***

Je ne suis qu'un test.
```

Ce genre de technique d'aiguillage à partir d'une constante est très souvent utilisé pour la mise au point des programmes avec une constante qui porte le nom traditionnel DEBUG. Cela permet d'activer ou non des instructions qui ne doivent servir que pour la mise au point.

Code source du projet complet

Voici pour finir en beauté le code source de la version ultime de notre projet.

Listing 7.8 : Projet Cryptopy.py (version 7 finale)

```
"""Cryptopy Version 7 (finale)
Crypte et decrypte un texte en chiffrement Cesar.
Sait travailler avec le contenu d'un fichier texte.
Brendan Scott (VF par OE)
"""

#### Imports
import string

#### Constantes
JEUCAR = string.printable[:-5]
CARSUBSTI = JEUCAR[-3:]+JEUCAR[:-3]

# Generation du dictionnaire avec les deux jeux de caracteres
# (en clair et substitutions).
DICO_ENCRYP = {}
DICO_DECRYP = {}
for i,k in enumerate(JEUCAR):
    v = CARSUBSTI[i]
    DICO_ENCRYP[k] = v
    DICO_DECRYP[v] = k
# Les autres  \t, \n etc. restent tels quels
for c in string.printable[-5:]: # Attention aux deux-points
    DICO_ENCRYP[c] = c
    DICO_DECRYP[c] = c

MSG_TEST = "J'adore les Monty Python. Trop lol."
NOMFIC_IN  = "cryptopy_in.txt"
NOMFIC_OUT = "cryptopy_out.txt"

#### Fonctions
def encrypter(texteclair, vardico_cryp):
    """ Crypte le message texteclair avec le dictionnaire
    fourni et renvoie le texte une fois rendu illisible. """
    textesecret = []
    for k in texteclair:
        v = vardico_cryp[k]
        textesecret.append(v)
    return ''.join(textesecret)

def decrypter(textesecret, vardico_decryp):
    """ Decrypte le message textesecret avec le dictionnaire
    fourni et renvoie texteclair. """
    texteclair = []
    for k in textesecret:
        v = vardico_decryp[k]
        texteclair.append(v)
    return ''.join(texteclair)

#### Section principale
if __name__ == "__main__":
## message = raw_input("Message qui doit devenir secret:\n")
## textesecret = encrypter(message, DICO_ENCRYP)
## texteclair  = decrypter(message, DICO_DECRYP)
## print("Message apres cryptage :")
## print(textesecret)
##
## print("Message redevenu lisible :")
## print(texteclair)
    print("*** Projet Cryptopy, version 7 ***\n")
ENCRYPT = False # True crypte vers OUT, False crypte vers
```

```
IN
with open(NOMFIC_IN,'r') as fic_lect:
message = fic_lect.read()

if ENCRYPT:
txt_resultat = encrypter(message, DICO_ENCRYP)
else:
txt_resultat = decrypter(message, DICO_DECRYP)

print(txt_resultat) # Pour tester. Enlever ensuite.

with open(NOMFIC_OUT,'w') as fic_ecri:
fic_ecri.write(txt_resultat)

# EOF (FIN DU PROGRAMME)
```

Récapitulons

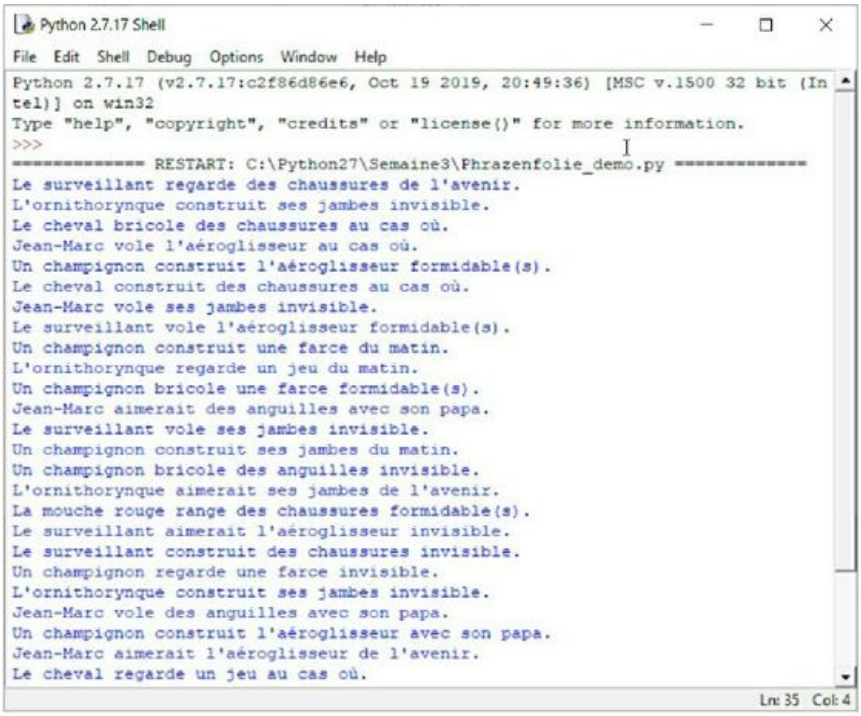
Voici les sujets abordés au cours de ce chapitre :

- Nous avons découvert le nouveau type de donnée dictionnaire. Un dictionnaire contient des articles, chaque article étant constitué d'une clé et d'une valeur.
- Un dictionnaire permet d'accéder directement à une valeur en fournissant la clé. Si `cle:valeur` est un article du dictionnaire `dico`, on accède à la valeur en écrivant `dico[cle]`.
- Nous savons créer un dictionnaire vide avec une paire d'accolades `{}`.
- Nous avons ajouté des articles dans un dictionnaire en spécifiant des couples clés : valeurs et en donnant des valeurs aux clés.
- Nous avons découvert le chiffrement César.
- Nous savons comment créer une nouvelle chaîne de caractères en faisant une jointure pour réunir tous les éléments d'une liste avec la méthode `join()`.
- Nous savons ouvrir un fichier en lecture et en écriture avec `open()` et le refermer avec `close()`.
- Nous savons lire et écrire dans un fichier avec les méthodes `read()` et `write()`.
- Nous avons vu les deux mots réservés `with` et `as` pour laisser Python faire le nettoyage et fermer les fichiers de lui-même.
- Nous avons vu comment réutiliser un module qui n'est pas un de ceux des librairies standard, par exemple ses propres programmes ou ceux de quelqu'un d'autre.
- Nous avons découvert l'écriture conditionnelle `if __name_ == __man__` pour isoler les fonctions de ton programme que tu veux pouvoir utiliser sans faire exécuter la partie principale du programme.
- Nous avons utilisé une constante pour contrôler le flux d'exécution dans une instruction conditionnelle.

Projet 8

Les phrases en folie

Ce petit projet va servir à reprendre notre souffle après les deux précédents et avant de plonger dans les deux suivants. Nous allons voir comment fabriquer un programme qui va générer des phrases amusantes, un peu dans l'esprit du jeu du cadavre exquis des poètes surréalistes. Ce sera l'occasion de découvrir les opérateurs pour contrôler le format des chaînes de caractères et ainsi améliorer l'affichage.



Au cours du projet, nous allons créer un patron dans lequel pourront être insérés des mots choisis plus ou moins au hasard dans une liste. C'est la combinaison aléatoire des mots pour former la phrase qui va donner des résultats absurdes, et souvent assez amusants.

Voici les étapes principales du projet :

1. **Nous allons d'abord découvrir les opérateurs de format pour les chaînes Python.**
2. **Nous allons créer un patron de formatage de chaîne.**
3. **Nous allons définir une liste de mots qui seront ensuite intégrés dans le patron de format.**
4. **Nous utiliserons une fonction standard pour faire choisir les mots au hasard.**
5. **Nous n'aurons plus qu'à afficher la phrase construite à partir des mots au moyen du patron de format.**

Des chaînes à contenu variable

Une chaîne de format permet de faire varier une partie du contenu d'un message en se servant d'un patron. Cela augmente les possibilités d'affichage des messages, et rend le code source plus lisible.

Commence par démarrer l'application Python IDLE. Dans la fenêtre de l'interpréteur, saisis l'instruction qui suit en faisant bien attention au fait qu'il y a deux chaînes séparées par un caractère que nous ne connaissons pas encore, le %.

```
>>> print("Bonjour %s ! Tu es un excellent auteur."%"Brendan")
Bonjour Brendan ! Tu es un excellent auteur.
```

Le résultat est très gentil pour l'auteur. Je t'en remercie.

Est-ce que tu comprends ce qui s'est passé ? La seconde chaîne, qui correspond à mon prénom « Brendan », a été injectée dans la première chaîne à l'endroit où se trouvait l'opérateur spécial `%s`.

Dans l'atelier IDLE, les chaînes de caractères sont par défaut affichées en vert et les opérateurs en noir.

- Le signe `%` s'appelle dans Python l'*opérateur de format*.
- La chaîne située à droite de l'opérateur est la *valeur de format*.
- La chaîne qui se trouve à gauche de l'opérateur est la *chaîne de format* ou le *patron*.
- Le % qui se trouve dans la première chaîne est un *spécificateur de conversion* (je jure que je n'invente pas ces mots).

Le spécificateur `%s` demande de convertir la chaîne servant de valeur de format vers le type chaîne (le `s` est l'initiale de string, chaîne). C'est ce spécificateur que tu utiliseras le plus souvent. Il en existe bien d'autres, mais les principaux sont (en plus de `%s`) :`%i` pour afficher des numériques entiers, `%f` pour des numériques à virgule flottante et `%%` (donc doublé) pour afficher le signe `%` lui-même.

En utilisant un patron pour créer des messages, tu vas grandement te simplifier la vie. Cela permet par exemple d'afficher facilement un message contenant une partie variable, du style `Il reste 15 tours avant la fin de la partie`. Le nombre de tours change à chaque affichage, mais le reste du message est fixe.

Ni trop, ni trop peu de valeurs de format

Pour générer nos phrases bizarres, nous allons combiner plusieurs mots en les injectant à la bonne position dans le patron. Voyons comment fonctionnent les spécificateurs de format dans la chaîne du patron. Nous allons adopter la procédure suivante :

1. **Nous insérons les spécificateurs dans le patron.**
2. **Nous indiquons l'une après l'autre les valeurs à injecter et à convertir, comme s'il s'agissait d'une liste, mais en ajoutant des parenthèses et non des crochets comme cela aurait été le cas pour une liste.**
3. **Il faut bien vérifier que le nombre de spécificateurs coïncide avec le nombre de valeurs de format qui doivent leur être associées.**

Prenons un exemple très simple :

```
>>> "%s %s"%(1,2)
'1 2'
```

Si le nombre de spécificateurs ne correspond pas au nombre de valeurs, tu obtiens une erreur. Voici deux instructions fausses :

```
"%s %s "%(1) " (erreur car pas de valeur pour le second spécificateur)
"%s %s"%(1,2,3)"(erreur car une valeur en trop, sans spécificateur)
```

Prends rapidement l'habitude de bien reconnaître les différentes erreurs, car tu en auras bientôt besoin pour la mise au point du programme. Il est possible de stocker les valeurs dans des variables qui seront fournies à la chaîne de format :

```
>>> vals = (1,2)
>>> "%s %s"%vals
'1 2'
>>>
```

```
>>> type(vals)
<type 'tuple'>
```

L'opérateur de format n'accepte qu'un seul argument. Par ailleurs, si tu ajoutes des parenthèses autour d'une série d'éléments séparés par des virgules, tu obtiens automatiquement un objet d'un nouveau type : un tuple. C'est cet unique objet qui est transmis à l'opérateur de format.

C'EST QUOI, UN TUPLE ?

Que signifie ce mot tuple que tu n'as sans doute jamais entendu ? Il est de la même famille que quintuple, sextuples, etc. C'est une collection de valeurs dans laquelle l'ordre est important, comme 1,2,3,4,5.

Le type de données tuple

Pour générer nos phrases, nous allons mettre en place des sortes de conteneurs à mots. Pour chaque essai, nous prendrons un mot dans chaque conteneur et nous l'insérerons avec les autres dans la chaîne de format. Puisque l'opérateur de format n'accepte qu'un seul argument, nous sommes forcés de regrouper les mots sélectionnés dans une variable unique. C'est cette variable qui pourra être transmise en argument à l'opérateur de format. Voilà pourquoi il nous faut ce nouveau type tuple.

Le tuple est le type de donnée Python le moins bien compris. Il ressemble à une liste, et certaines mauvaises langues disent que les tuples sont de mauvaises listes. En fait, le tuple est un peu différent de la liste. Tout d'abord, tu peux lire les éléments du tuple, mais tu ne peux jamais les modifier. C'est un type de données immuable : tu ne peux pas changer son contenu une fois qu'il est créé.

Voici les domaines d'utilisation principaux du tuple :

- lorsque tu gères des valeurs qui doivent être placées dans un ordre bien précis et ne plus en changer. C'est exactement ce dont nous avons besoin pour travailler avec une chaîne de format, c'est-à-dire un patron ;
- quand tu voudras utiliser une série de valeurs qui ne doivent absolument pas pouvoir être modifiées, ni volontairement, ni par erreur.

L'ordre des éléments dans le tuple est significatif, alors que dans une liste, l'ordre n'a pas d'importance.

On peut traverser (balayer) le contenu d'un tuple comme pour une liste afin de lire chacun des éléments qu'il contient :

```
>>> mon_tuple = ('e','3')
>>> for e in mon_tuple:
        print(e)

e
3
>>> mon_tuple[0]
'e'
>>> mon_tuple[1]
'3'
>>> mon_tuple(0)

Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    mon_tuple(0)
TypeError: 'tuple' object is not callable
```

Dans cette série d'essais, l'instruction `mon_tuple[0]` restitue la valeur du premier élément (les éléments sont numérotés à partir de zéro). La notation est la même que pour les listes, c'est-à-dire avec des

crochets. Les indices des éléments commencent à zéro, comme nous l'avons vu pour les substitutions du projet Projet 6.

Tu peux vérifier qu'il n'est pas possible de changer la valeur d'un élément en tentant d'affecter une valeur à l'élément zéro : `mon_tuple[0] = 17`. Tu vas obtenir une erreur. Prends l'habitude de reconnaître ce message d'erreur en anglais, pour pouvoir détecter une tentative de modification de tuple plus tard.

Rien n'empêche d'insérer un seul élément dans un tuple. Le code qui va utiliser les valeurs pour les injecter dans le patron n'accepte qu'un argument qui doit donc être un tuple. Que ce tuple contienne un ou plusieurs éléments ne change rien. Mais comment faire pour qu'un élément unique devienne un tuple ? Il suffit d'ajouter les parenthèses autour de l'élément et surtout de faire suivre l'élément d'une virgule.

```
>>> monotuple = (1,)
>>> monotuple
(1,)
```

C'est cette simple virgule qui prévient Python qu'il s'agit d'un tuple, pas les parenthèses. En général, on ajoute aussi les parenthèses pour que l'écriture soit plus lisible.

Grâce à un tuple, tu peux faire renvoyer plus d'une seule valeur par une fonction, ce qui est parfois indispensable. L'instruction qui a appelé la fonction peut récupérer ensuite le résultat sous forme d'une valeur unique de type tuple, ou de plusieurs éléments individuels. Dans ce cas, on parle de « déballage de tuple ».

Pour que ce déballage réussisse, il faut fournir exactement le bon nombre d'éléments. La fonction de test qui suit renvoie un tuple contenant les trois valeurs `(1,2,3)`. Le code qui a appelé la fonction peut récupérer un objet unique en écrivant `a = fonctest()` ou trois objets séparés en écrivant `a,b,c = fonctest()`. Toute autre écriture provoquera une erreur.

Cette technique basée sur le tuple offre donc beaucoup de souplesse à la mécanique de renvoi de valeur par les fonctions :

```
>>> def fonctest():
return(1,2,3)  # Renvoie un tuple de 3 éléments

>>> a = fonctest()
>>> a
(1, 2, 3)

>>> a,b,c = fonctest()
>>> a
1
>>> b
2
>>> c
3
>>> a,b = fonctest()

Traceback (most recent call last):
File "<pyshell#32>", line 1, in <module>
a,b = fonctest()
ValueError: too many values to unpack
```

On peut aussi distribuer le contenu d'un tuple directement dans des variables :

```
>>> a,b,c = (1,2,3)
>>> print("a: %s, b: %s, c: %s"%(a,b,c))
a: 1, b: 2, c: 3
>>> a,b = (1,2,3)

Traceback (most recent call last):
File "<pyshell#35>", line 1, in <module>
```

```
a,b = (1,2,3)
ValueError: too many values to unpack
```

Démarrons enfin le projet

Pour créer nos phrases bizarres, il nous faut définir un patron de format dans lequel nous pourrons injecter les mots. Il va organiser l'ordre suivant pour les mots :

```
sujet verbe objet qualificatif
```

Le patron de format correspondant pourra se présenter comme celui qui suit. Chaque opérateur `%s` marque un endroit où sera insérée une des valeurs pour construire la chaîne :

```
patron = "%s %s %s %s"
```

Commençons donc la réalisation de notre projet.

1. **Dans la fenêtre de l'interpréteur IDLE, demande l'ouverture d'un fichier vide avec** File/New (Fichier/Nouveau) **et enregistre immédiatement le fichier vide avec** File/Save (Fichier/Sauver) **en lui donnant le nom** *phrazenfolie.py*.

2. **Comme tu sais le faire, commence par mettre en place le commentaire de début de module qui servira de doc-chaîne.**

```
""" phrazenfolie.py Version 1
Programme qui produit des phrases bizarres en
appliquant un patron de chaînes.
Brendan Scott (VF par OE) """
```

3. **Ajoute ensuite le patron sur le modèle de celui décrit juste au-dessus :**

```
patron = "%s %s %s %s"
```

4. **Définis une constante pour stocker une première phrase figée qui permettra de faire des tests.**

```
PHRASE_BASE = "Mon professeur programme un jeu formi-
dable."
```

5. **Ajoute enfin les quatre définitions de mots suivantes :**

```
sujets = ["Mon professeur"]
verbes = ["programme"]
objets = ["un jeu"]
kalifs = ["formidable"]
```

Lorsque tu commences à travailler sur un nouveau projet, n'en fais pas trop. Il faut que la complexité augmente graduellement. Ne tente pas d'arriver au bout du projet le plus vite possible. Cela t'évitera de devoir chercher des nuits entières où se trouvent tes erreurs.

Pour vérifier que les valeurs de la liste sont bien injectées dans le patron et produisent une chaîne lisible, il nous faut insérer quelques instructions dans la partie principale.

1. **Insère un commentaire pour marquer le début de la section principale.**

2. **Saisis ensuite les lignes de la section principale telles que tu peux les lire dans le listing complet qui suit.**

Ces instructions commencent par créer un tuple en prenant le premier élément de chacune des quatre listes. Ce tuple est ensuite transmis au patron de format. La dernière instruction d'affichage affiche le résultat de la comparaison logique entre la chaîne synthétisée et la chaîne figée qui sert de témoin de contrôle.

Listing 8.1 : phrasenfolie.py (version 1)

```
""" phrazenfolie.py Version 1
Programme qui produit des phrases bizarres en
appliquant un patron de chaînes.
Brendan Scott (VF par OE) """

patron = "%s %s %s %s."
PHRASE_BASE = "Mon professeur programme un jeu formidable."

sujets = ["Mon professeur"]
verbes = ["programme"]
objets = ["un jeu"]
kalifs = ["formidable"]

# Section principale main
if __name__ == "__main__":
    sujet = sujets[0]
    verbe = verbes[0]
    objet = objets[0]
    kalif = kalifs[0]

    valformat = (sujet, verbe, objet, kalif)

    print(PHRASE_BASE)
    print(patron%valformat)
    print(PHRASE_BASE == patron%valformat)
```

En lançant l'exécution, tu dois voir apparaître ceci :

```
>>> ============================== RESTART ====================

>>>
Mon professeur programme un jeu formidable.
Mon professeur programme un jeu formidable.
True
```

D'OÙ VIENT LE SPAM ?

Le mot spam est en réalité une marque de viande en conserve. C'est également le sujet d'un sketch incroyable des Monty Python. On y voit un homme et une femme qui commandent à manger dans une taverne. La femme n'aime pas la marque Spam, mais tous les plats au menu contiennent le mot « Spam ».

À la table voisine, une bande de Vikings commence à entonner une chanson vantant les mérites de la viande Spam. C'est à cause de ce sketch que certaines variables temporaires dans Python sont souvent baptisées `spam` et c'est surtout la raison pour laquelle les courriels non désirés sont appelés du spam (en France, on dit aussi « pourriel »).

Des chaînes de synthèse !

Pour remplir le patron de format, nous devons sélectionner un élément au hasard dans chacune des quatre listes. Pour le moment, il n'y a qu'un élément dans chacune ; il n'y a donc aucun hasard. Ce n'est que pour les tests que nous nous limitons à un élément. Tu verras que nous n'aurons pas besoin de retoucher les instructions après avoir ajouté d'autres mots dans les listes qui servent de réservoirs de mots.

La sélection au hasard dans une liste est une opération assez facile. Nous disposons dans le module `random` de la fonction prédéfinie `choice()`. Je te propose de construire une liste par compréhension avec les

nombres de zéro à neuf puis d'appeler la fonction `random.choice()` pour qu'elle choisisse un des éléments au hasard :

```
>>> import random
>>> listerie = [x for x in range(10)]
>>> random.choice(listerie)
```

Nous pouvons maintenant passer à la version deux du code source :

1. **Tu peux supprimer ou neutraliser les instructions d'affichage et les autres instructions qui ne servent qu'à tester.**
2. **Chacune des quatre affectations du style `sujet = sujets[0]` est à remplacer par `sujet = random.choice(sujet)`.**
3. **Il faut ensuite construire un tuple à partir des valeurs de format.**
4. **Il ne reste plus qu'à afficher la phrase figée qui sert de repère puis le résultat de la synthèse de la chaîne à partir du patron et des valeurs de format.**

Je te laisse étudier la nouvelle version du projet et trouver quelles lignes il faut ajouter, modifier et supprimer par rapport à la version précédente.

Listing 8.2 : phrazenfolie.py (version 2)

```
""" phrazenfolie.py Version 2
Programme qui produit des phrases bizarres en
appliquant un patron de chaînes.
Brendan Scott (VF par OE) """

import random

# Section des contantes
patron = "%s %s %s %s."
PHRASE_BASE = "Mon professeur programme un jeu formidable."

sujets = ["Mon professeur"]
verbes = ["programme"]
objets = ["un jeu"]
kalifs = ["formidable"]

# Section principale main
if __name__ == "__main__":
    sujet = random.choice(sujets)
    verbe = random.choice(verbes)
    objet = random.choice(objets)
    kalif = random.choice(kalifs)

    valformat = (sujet, verbe, objet, kalif)

    print(PHRASE_BASE)
    print(patron%valformat)
    # print(PHRASE_BASE == patron%valformat)
```

Tu es sans doute impatient de voir ce que cela donne ? Allez, exécution !

L'affichage devra te confirmer que tout se passe bien : les deux phrases sont identiques, et pourtant la deuxième a été entièrement synthétisée.

```
>>> ============================== RESTART ==================

>>>
Mon professeur programme un jeu formidable.
```

```
Mon professeur programme un jeu formidable.
```

Enrichissons notre vocabulaire

Nous avons bien mérité une récréation. L'heure est venue de se montrer créatif. Pense à des mots qui pourraient bien se combiner les uns aux autres. Il faut trouver des sujets, des verbes, des objets et des qualificatifs. Bien sûr, l'accord du pluriel et du féminin ne sera pas peut-être pas toujours respecté.

Voici quelques règles pour les mots à chercher :

> » Chaque nouveau mot doit être de la catégorie prévue. Bien sûr, si tu mélanges les catégories, les phrases seront encore plus bizarres.
>
> » N'utilise pas de lettres accentuées.
>
> » Les mots ne doivent pas être composés. Mais il peut y avoir plusieurs mots dans l'élément, comme dans « Mademoiselle Poivre ».
>
> » N'utilise pas de pronoms comme je, tu, il. Il faut que le résultat soit à peu près correct.

Voici une suggestion de mots avec lesquels tu peux commencer :

```
sujets = [“Mon professeur”, “Un piranha”, “Thomas”, “Mlle
Poivre”, “Mon papa”, “Le lapin”]
verbes = [“mange”, “dort sur”, “observe”, “programme”,
“fabrique”, “vole”]
objets = [“un jeu”, “des anguilles”, “l’aeroglisseur”, “son
nez”, “des chaussures”, “une farce”]
kalifs = [“du futur”, “pour rire”, “de monstre”, “du matin”,
“en promotion”, “formidable(s)”]
“””[Ces quatre listes peuvent être beaucoup plus longues,
puisque tu peux distribuer les mots sur plusieurs lignes,
comme ici.]
L’essentiel est qu’il y ait une virgule avant chaque saut,
 de ligne.”””
```

Une fois que tu as créé ton réservoir de mots, n'oublie pas d'ajouter une paire de crochets autour de chaque liste. Ensuite, lance l'exécution pour vérifier que le résultat est aussi marrant que prévu. Dans l'extrait qui suit, j'ai supprimé la fameuse ligne de redémarrage de l'interpréteur.

```
Un piranha programme un jeu en promotion.
Mlle Poivre programme son nez en promotion.
Un piranha fabriquera une farce du futur.
Un piranha observe son nez du matin.
Mlle Poivre observe un jeu en promotion.
Mon professeur dort sur un jeu formidable(s).
Thomas programme un jeu formidable(s).
Mon papa fabriquera son nez du futur.
Mon papa a mangé l’aéroglisseur du matin.
Un piranha observe un jeu pour rire.
Mon professeur dort sur son nez du futur.
Le lapin dort sur un jeu de monstre.
Mlle Poivre fabriquera une farce de monstre.
Le lapin programme son nez en promotion.
Mon papa observe des chaussures du matin.
Mon professeur fabriquera une farce du futur.
Un piranha fabriquera un jeu du matin.
Mon papa fabriquera des chaussures pour rire.
Thomas dort sur une farce formidable(s).
Mon professeur vole des anguilles du matin.
```

Tout devrait fonctionner correctement. Cette fonction est un peu moins facile à tester parce que le résultat est aléatoire, à la différence d'une addition dans laquelle tu sais souvent toi-même quelle est la bonne réponse.

Je te laisse ajouter d'autres mots à ce vocabulaire.

Plus de phrases ?

Je te propose dans le fichier d'archive une troisième version dans laquelle j'ai mis en place une boucle `for` afin d'imprimer toute une série de phrases pendant la même exécution.

Le code source du projet complet

Voici le code source du projet dans sa version trois, donc avec quelques mots de vocabulaire (et nos exemples contiennent aussi une quatrième version qui affiche encore plus de phrases…).

Listing 8.3 : phrazenfolie.py en version finale

```
""" phrazenfolie.py Version 3 finale
Programme qui produit des phrases bizarres en
appliquant un patron de chaînes.
Brendan Scott (VF par OE) """

import random

# Section des contantes
patron = "%s %s %s %s."
PHRASE_BASE = "Mon professeur programme un jeu formidable."
sujets = ["Mon professeur", "Un piranha", "Thomas", "Mlle
Poivre", "Mon papa", "Le lapin"]
verbes = ["mange", "dort sur", "observe", "programme",
"fabrique", "vole"]
objets = ["un jeu", "des anguilles", "l'aeroglisseur", "son
nez", "des chaussures", "une farce"]
kalifs = ["du futur", "pour rire", "de monstre", "du matin",
"en promotion", "formidable(s)"]

# Section principale main
if __name__ == "__main__":
sujet = random.choice(sujets)
verbe = random.choice(verbes)
objet = random.choice(objets)
kalif = random.choice(kalifs)

valformat = (sujet, verbe, objet, kalif)

print(PHRASE_BASE)
print(patron%valformat)
# print(PHRASE_BASE == patron%valformat)
```

Récapitulons

Ce projet nous a permis de découvrir le puissant opérateur de format `%`. Nous l'avons utilisé pour créer des phrases bizarres. Voici les autres sujets abordés :

- Nous avons vu les chaînes de format, un des opérateurs de format, les valeurs de format et les spécificateurs.
- Nous avons utilisé le spécificateur de conversion de chaînes `%s`.

» Nous avons découvert un nouveau type de données non modifiable, le tuple.

» Le tuple sert à stocker des éléments dans un ordre significatif. Il permet de renvoyer plusieurs valeurs depuis une fonction et peut être « déballé ».

» Nous avons utilisé la fonction standard `random.choice()` pour sélectionner au hasard un mot parmi plusieurs dans une liste.

Semaine 4

Un peu de programmation orientée objet

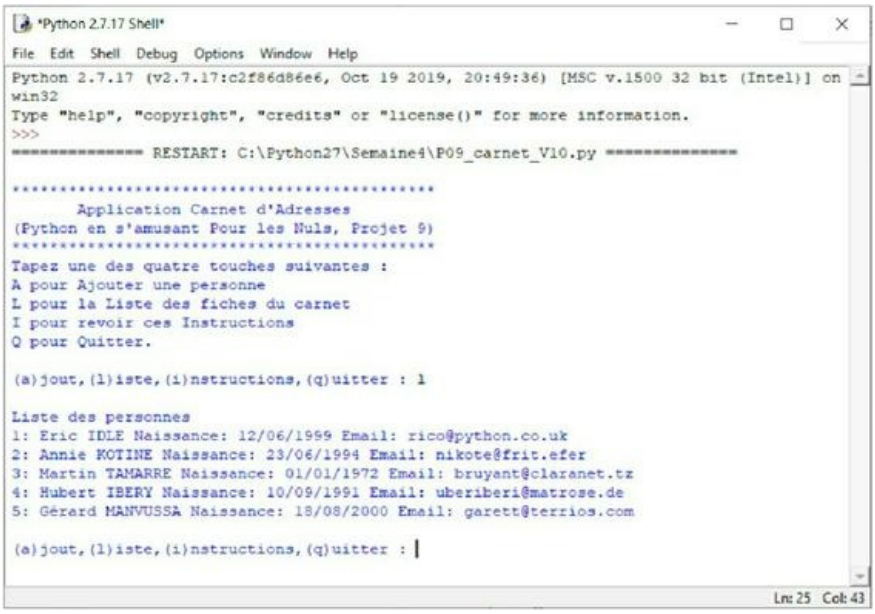
AU MENU DE CETTE SEMAINE

- Projet 9 : **Un carnet d'adresses**
- Projet 10 : **Multiplions !**

Projet 9

Un carnet d'adresses

Voici un projet qui va te guider dans le monde des applications réelles, puisque nous allons créer un programme de gestion de carnet d'adresses pour y stocker les noms, prénoms, dates de naissance et adresses de courriel de tes amis. Et qui n'a pas d'amis ? Tu pourras ensuite enrichir l'application pour y ajouter d'autres informations pour chacun de tes contacts. Tu pourras par exemple noter le nom de celui qui t'a emprunté un livre ou un billet de 10 €.



L'air de rien, ce projet va être l'occasion pour toi de découvrir un concept hyper-maxi fondamental de la programmation moderne, celui des classes et des objets. Tu vas définir des classes et créer des objets à partir de ces classes. Nous verrons comment stocker dans un fichier sur disque des objets que l'on peut recharger ensuite.

Des objets qui ont de la classe

Par principe, un carnet d'adresses est une répétition de fiches individuelles, ayant toutes le même format, mais avec des contenus variables. Nous allons créer une classe pour la fiche vide, ce qui nous permettra ensuite de créer autant d'objets que nous aurons de fiches de contact. Cela sera l'occasion de découvrir le nouveau mot réservé de Python nommé `class`.

Le concept de *classe* est au centre de la programmation orientée objet que l'on abrège en POO (en anglais on parle de OOP, *Object Oriented Programming*). Une classe regroupe des données et des fonctions.

Tu te souviens que quand on définit une variable du type chaîne, on hérite automatiquement de toute une série de méthodes utilisables avec cette chaîne ? Lorsque nous avons écrit une fonction, Python nous a immédiatement permis d'extraire la doc-chaîne du début du corps de la fonction avec sa fonction standard `help()`. Maintenant, l'heure est venue pour toi de mettre les mains dans le cambouis : tu vas définir tes propres classes, et plonger dans leurs entrailles pour définir toute une série de fonctions internes. Ne t'inquiète pas, ce n'est pas salissant.

Il y a deux niveaux dans la programmation orientée objet Python :

» La déclaration de classe. Elle sert à définir des modèles d'objets. Une *classe* ne fait rien, mais elle sert de modèle à partir duquel tu vas créer un ou plusieurs objets, qui sont des instances. Tu peux comparer la classe à

une recette de gâteau et l'objet à un gâteau. Tu ne peux pas créer un objet à partir d'un autre objet, mais toujours à partir d'une classe. Les objets ne sont pas des clones.

» Les instances de classes. Une *instance* est un objet réel stocké dans la mémoire de l'ordinateur, et construit grâce à son constructeur. C'est une fonction spéciale définie dans la déclaration de la classe. L'objet est l'instance d'une classe.

Définissons deux classes

Dans notre projet de carnet d'adresses, nous allons définir deux classes :

» Une classe pour le carnet d'adresses ; elle n'aura qu'une instance.

» Une classe pour les fiches individuelles des contacts. Il y aura une instance pour chaque personne dont tu veux stocker les informations.

La définition d'une classe ressemble beaucoup à celle d'une fonction :

1. **Il faut commencer par donner un nom à la classe.**

   Pour les noms des classes (suivant la règle PEP8 de Python), il faut commencer par une lettre capitale. Si le nom comporte plusieurs mots, il faut les accoler et écrire dans le format `DroMaDaire`, donc avec une capitale par mot dans le nom. Voici un exemple : `ClasseDeGrandeClasse`.

2. **Voici la syntaxe générique pour commencer la définition d'une classe :**

   ```
   class <NomDeLaClasse>(object):
   ```

   L'argument unique `(object)` signifie que la classe est d'un certain type dont elle hérite. Les classes peuvent hériter les unes des autres, mais nous ne détaillerons pas cela pour l'instant. Au minimum, elles doivent hériter de la classe `object`. Sauf si je dis le contraire, tu peux toujours ajouter la mention `(object)` dans toutes tes définitions de classes.

   N.d.T. : le choix de `object` comme nom pour la classe mère est mal choisi, puisque ce n'est pas un objet, mais une classe. La suite va te le prouver.

3. **Nous gardons la bonne habitude de commencer par une doc-chaîne qui décrit à quoi sert la classe.**

   Cette doc-chaîne fait partie du bloc de la classe, elle doit donc être indentée de quatre espaces.

4. **Après la doc-chaîne, il n'y a plus qu'à écrire ce qui va constituer le corps de la classe.**

   Pour l'instant, nous allons utiliser le mot réservé d'attente `pass` (même s'il n'est pas nécessaire, puisque tu as mis une doc-chaîne).

   Cette manière consistant à débuter la définition d'une classe va nous servir pour créer l'amorce de la classe nommée `FicheAdr`. Voyons d'abord dans l'interpréteur Shell comment interagir directement avec une définition de classe.

```
>>> class FicheAdr(object):
        """ Doc-chaine """
        pass

>>> FicheAdr      # Sans parenthèses
<class '__main__.FicheAdr'>
```

Dès que tu valides deux fois après avoir saisi l'instruction `pass`, cela provoque la création de la classe. Tu peux ensuite indiquer le nom de la classe nu, sans ajouter de parenthèses. Python va répondre que c'est bien une classe et qu'elle fait partie du programme principal `main`.

Créons un objet (une instance)

Pour créer un vrai objet, donc une instance, à partir d'une classe, il suffit d'indiquer le nom de la classe en ajoutant le jeu de parenthèses vide, comme si le nom de la classe était celui d'une fonction. Cela produit une instance de la classe correspondante, mais elle n'est stockée nulle part pour l'instant, et personne ne peut y accéder !

```
>>> FicheAdr()
<__main__.FicheAdr object at 0x026EF3F0>
```

Bien sûr, tu vas recueillir la référence à l'objet dans une variable dans laquelle l'objet venant d'être créé va être stocké :

```
>>> fiche_adr = FicheAdr()
```

Sois très attentif au fait que le nom de l'objet ressemble, mais n'est pas identique, à celui de la classe qui lui sert de modèle :

> » Le membre gauche (le nom de l'objet) utilise des mots en minuscules séparés par des caractères de soulignement (`nom_objet`).
>
> » Le nom de la classe du côté droit est écrit en format DroMaDaire, sans aucune espace ni sous caractère de soulignement (`NomClasse`).

En respectant cette manière d'écrire, tu pourras plus facilement distinguer les classes et les instances de classes que sont les objets.

Si tu interroges tour à tour les deux éléments dans l'interpréteur, tu constates que Python distingue bien les deux éléments :

```
>>> FicheAdr
<class '__main__.FicheAdr'>
>>> fiche_adr
<__main__.FicheAdr object at 0x026EFB70>
```

N'hésite pas à utiliser l'instruction `dir()` sur chacun des éléments pour voir tous les attributs de base dont tu peux directement profiter.

Attributs de classe et attributs d'objet

Nous avons vu dans les projets précédents que les types de données possèdent des attributs (des variables et des fonctions prédéfinies). Lorsque tu définis toi-même une classe, c'est à toi de choisir et de définir ses attributs.

Pour créer un attribut dans la définition d'une classe ou même dans une instance, on utilise la syntaxe basée sur le signe point, comme pour accéder aux méthodes des objets. Dans le Projet 6, nous avions utilisé la méthode de mise en capitales `upper()`, ce qui nous permettait de l'appliquer à notre message en écrivant `message.upper()`. C'est la même syntaxe que nous allons utiliser pour donner une valeur à un attribut qui deviendra une variable de la classe.

Dans l'exemple suivant, nous créons un attribut pour la classe et un autre pour l'objet (l'instance) en affectant une chaîne de caractères à chacun :

```
>>> FicheAdr.clattri = "Je suis un attribut de classe"
>>> fiche_adr.instattri = "Je suis un attribut d'instance"
>>> FicheAdr.clattri
'Je suis un attribut de classe'
>>> fiche_adr.instattri
"Je suis un attribut d'instance"
```

Et voici une petite surprise pour toi :

```
>>> fiche_adr.clattri
'Je suis un attribut de classe'
```

Nous avons bien indiqué le nom de l'objet au lieu de celui de la classe pour accéder à un attribut de la classe, et pas d'erreur ! Nous voyons que cet objet a hérité d'un nouvel attribut de la classe, alors que l'objet avait été créé avant (nous l'avons créé dans la session précédente). En revanche, et c'est tout à fait normal, la classe n'a pas hérité de l'attribut de son instance, puisqu'on n'hérite pas de ses enfants (en général). Tu peux utiliser l'instruction `dir()` pour le vérifier.

Lorsque tu demandes à Python de trouver un attribut, il va chercher dans un certain ordre. Il fait cette recherche au moment où il rencontre la ligne de la demande. C'est pour cela que l'instance avait entre-temps hérité des nouveaux attributs de la classe, même si elle avait été créée avant le changement dans la définition. En revanche, la classe ne profite pas des nouveaux attributs de l'instance, ce qui n'a d'ailleurs aucun sens. J'expliquerai pourquoi plus tard.

Souvent, tu auras besoin de créer une définition de classe en prévoyant des valeurs par défaut pour les attributs dont auront besoin les objets qui vont être créés à partir d'elle. Et, souvent aussi, il faudra changer les valeurs de ces éléments, principalement pour ceux qui sont des données. Les méthodes de la classe sont beaucoup moins souvent redéfinies, mais nous verrons néanmoins un exemple.

Nous pouvons par exemple créer un deuxième objet à partir de la même classe puis modifier au niveau de cet objet un attribut qu'il a hérité de la classe. Cette technique consistant à modifier une valeur héritée est appelée redéfinition d'attribut (override).

```
>>> fiche_adr2 = FicheAdr()
>>> fiche_adr2.clattri = "Attribut de classe redefini"
>>> fiche_adr2.clattri
'Attribut de classe redefini'
>>> FicheAdr.clattri
'Je suis un attribut de classe'
```

Cette redéfinition ne concerne que l'objet en question. L'attribut de la classe et celui des autres objets issus de la même classe n'est pas modifié.

Définissons notre projet

Le projet de carnet d'adresses va se contenter de stocker quatre données pour chaque contact :

- le prénom
- le nom de famille
- la date de naissance
- l'adresse de courriel (de messagerie ou d'e-mail)

Une fois le projet terminé, tu pourras ajouter d'autres champs, comme des petites notes sur ce que tu penses de chacun de tes contacts, l'adresse Skype, quels sont les amis de chacun, etc.

Démarrons le projet et créons une classe

Voici la première étape de nos exercices pratiques :

1. **Nous commençons par créer un nouveau fichier de projet vide. Je pense que je n'ai plus besoin de t'expliquer comment faire. Pourquoi ne pas l'appeler** *carnet.py* **?**

2. **Nous insérons une doc-chaîne avec les triples guillemets pour expliquer à tout le monde qu'il s'agit d'une application de gestion de carnet d'adresses.**
3. **Nous ajoutons une ligne de commentaires qui va servir de titre pour la section des classes, `#### Section des classes`.**
4. **Dans cette section, nous mettons en place l'amorce d'une définition de classe, que nous allons nommer `CarnetAdr`.**

   C'est cette classe qui va définir le carnet d'adresses. Je te rappelle qu'il faut commencer par ajouter une doc-chaîne, même minimale, dans le bloc.

   L'amorce d'une classe est très proche de celle d'une fonction.
5. **Pour l'instant, on ne prévoit qu'une seule instruction dans l'amorce, l'instruction `pass`. Copie ensuite toute l'amorce de la classe puis colle la copie en dessous pour créer l'amorce de la seconde classe. Change son nom en `FicheAdr`.**
6. **Insère ensuite un titre de section `#### Section principale`.**
7. **Dans cette section, nous prévoyons dès le départ de pouvoir utiliser ce projet sous forme d'un module au bénéfice d'un autre en créant un bloc conditionnel basé sur `if`. Dans ce bloc, nous créons un objet à partir de la première classe et un autre à partir de la seconde. Vérifie ta rédaction avec le listing complet qui suit.**

Listing 9.1 : carnet.py (version 1)

```
"""
carnet.py  Version 1
Gestion de fiches de contacts avec
prenom, nom, naissance, courriel (email)
Brendan Scott (VF par OE)
"""

### Section Classes
class CarnetAdr(object):
    """ Conteneur de fiches """
    pass

class FicheAdr(object):
    """ Fiche d'un contact """
    pass

### Section principale main
if __name__ == "__main__":
    carnet_adr = CarnetAdr()
    contact1 = FicheAdr()
```

Tu peux lancer l'exécution du projet dans cet état initial. Normalement, rien ne s'affiche (pas de nouvelles, bonnes nouvelles). Pourtant, les deux classes et les deux objets sont bien créés. Nous allons le vérifier.

Remplissons une première fiche

Pour associer des données à un objet, c'est-à-dire les stocker dans ses attributs, il suffit de fournir des noms pour les attributs et des valeurs, comme pour n'importe quel autre type de données.

Voici par exemple comment créer une fiche pour un premier contact. Je suppose que tu n'as pas redémarré l'atelier IDLE depuis les précédents essais dans l'interpréteur :

```
>>> contact1 = FicheAdr() #cree la fiche
>>> contact1.prenom  = "Eric"
```

```
>>> contact1.nomfami = “IDLE”
>>> contact1.datenaiss = “12/06/1999”
>>> contact1.courriel  = None
```

Python propose une technique pour simplifier la mise en place des valeurs initiales d'un objet au moment de la création de celui-ci. Il suffit de définir dans la classe une méthode portant un nom spécial : `__init__`. Lorsqu'une instance est créée à partir de la classe, Python tente d'appeler la méthode portant ce nom. Cette méthode contient des instructions pour donner des valeurs initiales aux attributs de l'objet.

En programmation objet, la méthode `__init_ _` est le *constructeur*. Il construit l'objet en y insérant des données, parmi d'autres opérations de démarrage.

Tu sais déjà qu'une méthode est une fonction appartenant à un objet. Pour définir la méthode `__init__` pour une classe déjà définie, il suffit d'ajouter la définition de la méthode dans le bloc de code de la classe.

Voici quelques points à prendre en compte :

- Inutile de chercher un joli nom. Tu dois utiliser le nom `__init__`, un point, c'est tout.
- À la différence des fonctions, les méthodes doivent toujours avoir au moins un argument, et cet argument doit nécessairement porter le nom prédéfini `self` (soi-même). Lorsque tu veux définir des valeurs initiales au niveau de la classe, il suffit de les transmettre en tant qu'argument d'entrée du constructeur. Lui va se charger de copier les valeurs dans les attributs.

Je te propose de découvrir le code source du constructeur de la classe `FicheAdr`. Dans notre exemple, nous choisissons d'initialiser le prénom, le nom, la date de naissance et l'adresse de courriel. Nous prévoyons la pseudo-valeur None, ce qui permet de créer l'objet avec de un à cinq arguments dans l'appel au constructeur :

```
class FicheAdr(object):
    """ Fiche d'un contact """
    def __init__(self, prenom=None, nomfami=None,
                 datenaiss=None, courriel=None):
        """Initialise les 4 attributs. datenaiss doit
        se formater JJ/MM/AAAA
        """
        self.prenom    = prenom
        self.nomfami   = nomfami
        self.datenaiss = datenaiss
        self.courriel  = courriel
```

Pour l'instant, tu n'as pas besoin de saisir toutes ces lignes, car le code source complet est présenté un peu plus loin. Pour définir la méthode `__init__`, j'utilise une instruction classique `def`. La méthode est suivie d'une liste d'arguments, comme une fonction. Excepté le premier argument `self`, tous les arguments fonctionnent avec une valeur par défaut s'ils ne sont pas fournis. Je vais parler de `self` un peu plus loin.

Il faut bien surveiller l'indentation des lignes. La méthode `__init__` appartient au bloc de la définition de la classe. Autrement dit, toutes les lignes du corps de la méthode doivent être décalées de deux fois quatre espaces.

Tu constates que je n'ai pas oublié de prévoir une doc-chaîne en début de bloc. Parfois, je n'indique pas grand-chose dans cette doc-chaîne, car je n'ai pas besoin d'aide concernant le constructeur de mes classes.

Tu remarques que cette méthode ne se termine pas par une instruction de renvoi de valeur `return`. Il n'y en a pas besoin. À l'intérieur de la méthode, nous affectons des valeurs à différents attributs de l'objet symbolisé par `self` (le premier argument). Si tu remontes pour voir la dernière session interprétée, tu vois que les affectations des attributs ont été faites par rapport à un objet réel (du style `contact1.prenom = “Eric”`). Dans le constructeur, il n'y a encore aucune instance à ce moment. Il n'y a aucun nom d'objet à indiquer en préfixe du nom de l'attribut.

ON N'EST JAMAIS SI BIEN SERVI QUE PAR SELF

L'utilisation de `self` pour symboliser une référence à un objet est le fruit d'une convention. En théorie, on peut utiliser un autre nom, mais la plupart des programmeurs utilisent `self`. Si tu veux que tes programmes soient compréhensibles par les autres, ne joue pas le rebelle.

La méthode constructeur `__init__` doit permettre de créer autant d'instances que nécessaire à partir de la même classe. On ne peut donc pas indiquer un nom d'objet en particulier, car le constructeur serait lié à cet objet. C'est à cela que sert le mot magique `self`.

Quand on définit une méthode dans une classe, Python demande que le premier argument de la méthode soit une référence à l'objet lui-même. Voilà pourquoi on peut écrire les affectations sans connaître le nom de l'objet auquel elles seront appliquées en réalité. Il suffit de connaître le nom de la variable temporaire qui permettra d'effectuer la référence. Ce nom est `self`. La référence à l'objet réel sera transmise au constructeur `__init__` au moment où l'objet sera créé pendant l'exécution.

Construisons une instance avec__init__

Voyons comment mettre à profit cette nouvelle méthode pour donner des valeurs initiales à l'objet que nous allons créer :

1. **Nous commençons par insérer toute la méthode `__init__` dans le corps de la classe `FicheAdr`.**
2. **Dans la section principale, nous modifions l'appel qui permet de créer un objet à partir de la classe :**

```
contact1 = FicheAdr("Eric", "IDLE", 12/06/1999", None)
```

3. **Pour vérifier que le programme fonctionne, nous ajoutons une ligne pour afficher le contenu de l'objet `contact1`.**

```
print(contact1)
```

Le listing suivant montre l'aspect du projet dans son état en version 2.

Listing 9.2 : carnet.py (version 2)

```
"""
carnet.py  Version 2
Gestion de fiches de contacts avec
prenom, nom, naissance, courriel (email)
Brendan Scott (VF par OE)
"""

### Section Classes
class CarnetAdr(object):
    """ Conteneur de fiches """
    pass

class FicheAdr(object):
    """ Fiche d'un contact.
    """
    def __init__(self, prenom=None, nomfami=None,    #AJOUT
                 datenaiss=None, courriel=None):
        """Initialise les 4 attributs. Le format de
        datenaiss est JJ/MM/AAAA
        """
        self.prenom    = prenom
        self.nomfami   = nomfami
```

```
        self.datenaiss = datenaiss
        self.courriel  = courriel

### Section principale main
if __name__ == "__main__":
    carnet_adr = CarnetAdr()
    contact1 = FicheAdr("Eric", "IDLE", "12/06/1999", None)
    print(contact1)
```

L'exécution du code montre quelque chose dans le style suivant :

```
<__main__.FicheAdr object at 0x0289F5D0>
```

Ce n'est pas très agréable à lire, mais c'est parce que Python ne sait pas comment imprimer le contenu de ton objet. Il ne peut pas faire le tri entre les données qui doivent être affichées pour l'utilisateur et les données techniques. C'est pourquoi il se contente d'indiquer de quel type d'objet il s'agit et où il se trouve en mémoire.

Mais comment savoir que les données ont bien été mises en place pendant l'initialisation ? Pour l'instant, nous ne pouvons pas. C'est la prochaine étape de notre projet que de voir comment afficher les détails d'une instance de `FicheAdr`.

Affichons les données d'un objet

Nous allons créer une fonction pour afficher tous les détails d'une instance de la classe `FicheAdr`. Cela va nous permettre de vérifier que notre initialisation a bien fonctionné. Nous réutiliserons plus loin le code de cette fonction pour en faire une méthode de classe.

Voici la procédure :

1. **Nous commençons par ajouter un titre de section pour les fonctions avec ###, juste avant la section principale.**
2. **Nous ajoutons la ligne de tête d'une fonction qui doit porter le nom `__repr__` et qui n'accepte qu'un argument d'entrée.**

   L'argument sera la référence à un objet issu de `FicheAdr`. Rien ne nous interdit de donner à cet argument le nom spécial `self`. Tu devines peut-être pourquoi je choisis ce nom ? (j'ai donné une piste un peu plus haut quand j'ai parlé de méthodes.)

3. **Dans le corps de la fonction, je commence par créer un patron avec des spécificateurs de format `% s` :**

```
patron = "FicheAdr(prenom ='%s', "+\
         "nomfami ='%s',"+\
         "datenaiss ='%s', "+\
    "courriel ='%s')"
```

4. **La seule autre instruction de la fonction consiste à renvoyer par `return` le patron une fois rempli avec les valeurs des attributs que possède nécessairement un objet s'il a été créé à partir de la classe `FicheAdr`, c'est-à-dire un prénom, un nom de famille, une date de naissance et une @dresse de courriel.**
5. **Dans la section principale, nous ajoutons une ligne tout à la fin pour faire afficher le résultat de l'appel à cette nouvelle fonction en lui donnant notre unique objet de tests en l'argument :**

```
print(__repr__(contact1))
```

Voici un extrait de la troisième version du projet. J'ai omis la section des classes car elle n'a pas changé. Je montre la nouvelle section des fonctions et la section principale à laquelle j'ai ajouté une ligne :

Listing 9.3 : Extrait de carnet.py (version 3)

```
### Section Fonctions
def __repr__(self):
    """ Produit une chaine selon l'objet
    FicheAdr fourni en argument self.
    """
    patron = "FicheAdr(prenom ='%s', "+\
         "nomfami ='%s',"+\
         "datenaiss ='%s', "+\
         "courriel ='%s')"
    return patron%(self.prenom,self.nomfami,
                   self.datenaiss,self.courriel)

### Section principale main
if __name__ == "__main__":
    carnet_adr = CarnetAdr()

    contact1 = FicheAdr("Eric", "IDLE", "12/06/1999", None)
    print(contact1)
    print(__repr__(contact1))        #AJOUT
```

Voici ce qui s'affiche lorsque tu exécutes ce projet :

```
<__main__.FicheAdr object at 0x0282F610>
FicheAdr(prenom ='Eric', nomfami ='IDLE',datenaiss
='12/06/1999', courriel ='None')
```

La première ligne est celle de l'instruction qui existait déjà. C'est la suivante qui nous intéresse. Elle est produite par l'appel à la fonction `__repr__`. Deux remarques :

» Les valeurs des quatre attributs sont bien présentes. Cela prouve que la fonction constructeur a bien affecté les valeurs à ces attributs. (Techniquement parlant, la pseudo-valeur `None` devrait être affichée sans les apostrophes, mais y parvenir correctement suppose un peu plus de connaissances, qui ne sont pas indispensables ici.)

» À part cette pseudo-valeur, ce qui est affiché peut être directement copié et collé dans le code pour créer une autre instance de `FicheAdr`.

La magie de __repr__

Avais-tu deviné pourquoi j'avais choisi le nom `self` pour l'argument ? Tu te souviens peut-être qu'au cours du Projet 4 (il y a bien longtemps), j'avais indiqué que les fonctions dont le nom commençait par deux caractères de soulignement ___ étaient considérées à part en langage Python. C'est le cas de la fonction `__repr__`.

Lorsque Python doit afficher le contenu d'un objet, il tente toujours d'appeler la méthode de représentation de cet objet, qui porte obligatoirement le nom `__repr__`.

Si tu fais afficher la liste des attributs de la classe `FicheAdr` au moyen de l'instruction `dir()`, tu peux vérifier qu'il y a une méthode portant ce nom `__repr__`. C'est cette méthode qui est utilisée pour afficher laligne technique avec l'adresse de l'objet en mémoire. Si tu redéfinis cette méthode, c'est ta fonction qui sera appelée par l'instruction d'affichage `print()`. Formidable !

Pour vérifier, nous pouvons ajouter la ligne suivante tout à la fin de la section des fonctions en dehors du corps de la fonction (‘pas de décalage à gauche). Cette instruction va relier une méthode de classe à une fonction. Tu remarques qu'il n'y a pas de parenthèses ici :

```
FicheAdr.__repr__ = __repr__
```

Si tu relances l'exécution, tu constates que dorénavant les deux instructions d'affichage aboutissent au même résultat :

```
FicheAdr(prenom ='Eric', nomfami ='IDLE',datenaiss
='12/06/1999', courriel ='None')
FicheAdr(prenom ='Eric', nomfami ='IDLE',datenaiss
='12/06/1999', courriel ='None')
```

Alors, c'est de la magie ou pas ? Bien sûr que c'est de la magie !

La ligne que nous venons d'ajouter remplace la méthode `__repr__` héritée par défaut dans la classe `FicheAdr` par la fonction homonyme que nous venons de définir. (La méthode `__repr__` par défaut est reçue automatiquement en héritage par la classe puisqu'elle hérite d'une classe ancestrale appelée `object`. Une fois que cette redéfinition est faite, tu peux effectivement utiliser l'instruction beaucoup plus simple `print(contact1)` pour afficher correctement les détails d'une fiche.

Cette façon de faire n'est pas la meilleure, et elle ne m'a servi qu'à te montrer comment cela fonctionne. La bonne technique (j'ai failli dire la bonne méthode) consiste à déplacer tout le code de `__repr__` dans la définition de la classe, pour que cela devienne une de ses méthodes. Voici la procédure.

1. **Nous copions la totalité du code de la fonction `__repr__ (self)`, tête comprise, dans le corps de la classe `FicheAdr`, après la méthode `__init__`. Nous n'oublions pas d'indenter toutes les lignes du corps d'un niveau, puisqu'elle doit devenir une méthode appartenant à la classe.**
2. **En fin de section des fonctions, nous supprimons la ligne isolée qui a servi aux explications :**

```
FicheAdr.__repr__ = __repr__
```

La section des fonctions doit normalement être vide maintenant.

3. **En bas de la section principale, il faut supprimer la ligne d'affichage suivante. Nous n'en avons plus besoin et elle ne fonctionne plus dorénavant. (La fonction n'existant plus, elle provoque même une erreur.) :**

```
print(__repr__(contact1))
```

Voici un extrait de la quatrième version du projet. Je ne montre que le corps de la classe `FicheAdr`, avec sa nouvelle méthode et la section principale. Je laisse la section des fonctions pour montrer qu'elle doit être vide maintenant.

Listing 9.4 : carnet.py (version 4, extrait)

```
"""
carnet.py  Version 4
Gestion de fiches de contacts avec
prenom, nom, naissance, courriel (email)
Brendan Scott (VF par OE)
"""

### Section Classes
class CarnetAdr(object):
    """ Conteneur de fiches """
    pass

class FicheAdr(object):
    """ Fiche d'un contact.
    """
    #AJOUT Methode
    def __init__(self, prenom=None, nomfami=None,
                 datenaiss=None, courriel=None):
        """ Initialise les 4 attributs. Le format de
        datenaiss est JJ/MM/AAAA
```

```
        """
        self.prenom    = prenom
        self.nomfami   = nomfami
        self.datenaiss = datenaiss
        self.courriel  = courriel

    def __repr__(self):
        """ Produit une chaine selon l'objet
        FicheAdr fourni en argument self.
        """
        patron = "FicheAdr(prenom ='%s', "+\
             "nomfami ='%s',"+\
             "datenaiss ='%s', "+\
             "courriel ='%s')"
        return patron%(self.prenom,self.nomfami,
                       self.datenaiss,self.courriel)

### Section Fonctions

### Section principale main
if __name__ == "__main__":
    carnet_adr = CarnetAdr()
    contact1 = FicheAdr("Eric", "IDLE", "12/06/1999", None)
    print(contact1)
```

Créons une instance de CarnetAdr

Pour l'instant, la classe `CarnetAdr` n'a jamais servi à créer un objet ; d'ailleurs, elle est vide. Le carnet est destiné à regrouper les fiches des contacts. L'heure est venue de donner du corps à cette classe. Nous allons en profiter pour raccorder des instances de fiche d'adresse à la liste.

1. **Nous commençons par supprimer l'instruction `pass` dans la classe `CarnetAdr`.**
2. **Nous mettons en place le squelette d'une méthode constructeur `__init__`. Elle n'a besoin que d'un seul argument `self`.**
3. **Nous n'oublions pas de prévoir une doc-chaîne si elle n'existe pas encore ou de modifier celle qui existe.**
4. **Dans le constructeur, nous définissons un attribut nommé `gens` comme étant une liste vide.**
5. **Nous définissons ensuite une méthode portant le nom `ajouter_fiche()`, toujours dans la classe.**
6. **Cette méthode aura deux arguments, `self` et `fiche_nouvo`.**
7. **Nous ajoutons une doc-chaîne pour la méthode en expliquant qu'elle sert à ajouter une nouvelle fiche à la liste des gens.**
8. **La seule instruction de cette méthode va permettre d'ajouter une fiche grâce à la fonction `append()` :**

```
self.gens.append(fiche_nouvo)
```

9. **Nous pouvons alors descendre jusqu'à la section principale pour ajouter deux lignes tout à la fin :**

```
carnet_adr.ajouter_fiche(contact1)
print(carnet_adr.gens)
```

La classe `FicheAdr` n'a pas été modifiée. Voici néanmoins le code source complet du projet dans cette nouvelle version. La classe `CarnetAdr` a été modifiée ainsi que la section principale en bas.

Listing 9.5 : carnet.py (version 5)

```
"""
carnet.py  Version 5
```

```
Gestion de fiches de contacts avec
prenom, nom, naissance, courriel (email)
Brendan Scott (VF par OE)
"""

### Section Classes
class CarnetAdr(object):
    """ Conteneur de fiches """
    #AJOUT 2 methodes
    def __init__(self):
        """ Cree l'attribut gens comme liste vide.
        """
        self.gens = []

    def ajouter_fiche(self, fiche_nouvo):
        """ Ajoute une fiche dans gens.
        """
        self.gens.append(fiche_nouvo)

class FicheAdr(object):
    """ Fiche d'un contact.
    """
    def __init__(self, prenom=None, nomfami=None,
                 datenaiss=None, courriel=None):
        """Initialise les 4 attributs. Le format de
        datenaiss est JJ/MM/AAAA
        """
        self.prenom    = prenom
        self.nomfami   = nomfami
        self.datenaiss = datenaiss
        self.courriel  = courriel

    def __repr__(self):
        """ Produit une chaine selon l'objet
        FicheAdr fourni en argument self.
        """
        patron = "FicheAdr(prenom ='%s', "+\
             "nomfami ='%s',"+\
             "datenaiss ='%s', "+\
             "courriel ='%s')"
        return patron%(self.prenom,self.nomfami,
                       self.datenaiss,self.courriel)

### Section Fonctions
# FicheAdr.__repr__ = __repr__

### Section principale main
if __name__ == "__main__":
carnet_adr = CarnetAdr()
contact1 = FicheAdr("Eric", "IDLE", "12/06/1999", None)
print(contact1)
carnet_adr.ajouter_fiche(contact1)  #AJOUT
print(carnet_adr.gens)              #AJOUT
```

L'exécution semble donner le même résultat que la version précédente, mais observe bien le second affichage :

```
FicheAdr(prenom ='Eric', nomfami ='IDLE',datenaiss
='12/06/1999', courriel ='None')
[FicheAdr(prenom ='Eric', nomfami ='IDLE',datenaiss
='12/06/1999', courriel ='None')]
```

La seconde ligne n'affiche pas le contenu d'une chaîne, mais celui d'une liste, la liste `gens`, ce qui est confirmé par la présence d'un crochet au début et d'un autre tout à la fin. Python a utilisé sa magie en appelant la méthode `__repr__` pour afficher le contenu de `contact1`. Nous pourrions définir la même méthode de préparation à l'affichage pour la classe `CarnetAdr`, mais nous n'en avons pas besoin ici.

Dans la saumure, les objets !

Pour l'instant, dès que le programme termine son exécution, les objets créés disparaissent, avec toutes leurs données. Comment sauver les données du carnet d'adresses ? En écrivant le contenu dans un fichier sur disque ! Nous allons nous servir d'un module qui permet de stocker et de relire les objets tels qu'ils sont en mémoire. Le nom de ce module est `pickle`, ce qui signifie saumure (de l'eau extrêmement salée dans laquelle on conserve les aliments).

Nous allons importer le module `pickle` pour pouvoir se servir de sa méthode de stockage nommée `dump()`. Elle fonctionne avec un fichier déjà ouvert. (Tu peux revoir le Projet 7 si tu ne te souviens pas comment ouvrir un fichier.)

Commençons par une petite session dans l'interpréteur :

```
>>> import pickle
>>> NOMFIC = "test.pickle"
>>> liste_tempo = [x*2 for x in range(10)]
>>> liste_tempo                          # Confirmons
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

>>> # Stockons
>>> with open(NOMFIC,'w') as fico:
pickle.dump(liste_tempo, fico)

>>> # Ouvrons le fichier pour le relire; changer w en r!!!
>>> with open(NOMFIC,'r') as ficor:
print(ficor.read())

(lp0
I0
aI2
aI4
aI6
aI8
aI10
aI12
aI14
aI16
aI18
a.
```

Dans cet essai, nous avons créé un objet de type liste sous le nom `liste_tempo` puis nous l'avons stocké dans un fichier portant le nom *test.pickle*. Nous avons ensuite ouvert à nouveau le fichier en lecture pour récupérer le contenu. Tu constates que ce contenu ne se limite pas aux données de la liste, mais tu peux deviner que les valeurs sont présentes.

D'après l'aide en ligne, le module pickle sert à « créer des représentations sérialisées et portables des objets Python ». Plus facile à dire qu'à comprendre. Les objets Python sont créés dans la mémoire vive (DRAM). La position exacte des différents objets dans l'espace mémoire dépend de l'ordinateur et du modèle du système d'exploitation. Pour pouvoir transférer des objets d'une machine à une autre, il faut donc normaliser la représentation pour que les objets puissent être réimplantés dans la mémoire d'un autre ordinateur dont l'organisation est différente.

Ce travail de normalisation s'appelle la *sérialisation*. Il consiste à placer les objets les uns après les autres, en série. Le résultat est stocké dans un fichier ou transmis à une autre machine. Le module pickle permet

justement de stocker des objets dans un fichier de telle manière que n'importe quel autre programme Python sera capable de relire le contenu du fichier et de remettre en place une copie des objets.

Ceci dit, tous les objets ne peuvent pas être sérialisés. Seuls peuvent l'être les objets simples, et ce sont les objets qui possèdent parmi leurs attributs la méthode nommée `__hash()__` et soit la méthode `_ _eq()__`, soit la méthode `_ _cmp()__`. Pour les objets plus complexes, cela ne fonctionnera pas, mais le jour où tu auras besoin de te pencher sur cette limitation, tu auras assez progressé pour savoir trouver la solution tout seul.

Pour vérifier que notre essai a bien fonctionné, nous allons essayer de remettre en place l'objet que nous avons sauvé. Pour y parvenir, et sans tricher, il faut quitter l'atelier IDLE totalement, en fermant toutes ses fenêtres. Tu peux ensuite le redémarrer et faire l'essai qui suit :

```
>>> import pickle
>>> NOMFIC = "test.pickle"
>>> with open(NOMFIC,'r') as ficor:
recup = pickle.load(ficor)
>>> recup
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Le résultat affiché est bien le contenu de la liste qui avait été sauvée. Il a donc bien survécu à l'arrêt de l'atelier IDLE ou au redémarrage de l'ordinateur, bien au chaud dans le fichier `test.pickle`.

Voici la procédure générale permettant de sauver un objet portant le nom `petit_obj` au moyen du module pickle :

1. **On commence par choisir le nom du fichier destinataire.**

   Dans notre projet, le nom sera le même, quel que soit l'utilisateur du programme. Il sera défini dans une constante.

2. **On ouvre ce fichier en mode écriture, avec l'attribut `'w'` puis on récupère l'objet fichier que renvoie la fonction `open()`. Il ne faut pas hésiter à utiliser la technique basée sur le mot réservé `with` (voir ci-dessus).**

   Dans ce projet, il est logique d'ouvrir toujours le fichier en mode écriture, parce qu'on ne veut pas conserver l'ancienne version éventuelle du contenu du fichier. Il est en théorie possible d'ajouter des objets à un fichier existant grâce à l'attribut d'ouverture `'a'`, mais je n'en parle pas ici.

3. **Il ne reste plus qu'à appeler la fonction de vidage vers le fichier :**

   ```
   pickle.dump(fico, petit_obj)
   ```

Pour réaliser l'opération inverse consistant à charger un objet depuis un fichier avec le même module, on procède ainsi :

1. **On vérifie le nom du fichier dans lequel se trouve l'objet.**
2. **On ouvre ce fichier en mode lecture avec l'attribut `'r'` et on récupère l'objet fichier renvoyé par `open()`. On utilise ici aussi le mot réservé `with` :**

```
with open(NOMFIC,'r') as ficor:
```

3. **On appelle enfin la méthode de chargement nommée `load()` et on récupère ce qu'elle renvoie dans une variable :**

```
recup = pickle.load(ficor)
```

Le module `pickle` peut être personnalisé, mais ce n'est pas nécessaire ici. Il existe une variante de ce module qui porte le nom `cPickle`. Elle n'est pas personnalisable, mais beaucoup plus rapide pour charger et sauver les données.

Je te conseille d'utiliser cette variante `cPickle` dans tes projets. Tu verras peut-être que les gens continuent à parler du module pickle, même lorsqu'ils utilisent la variante `cPickle`. Même dans le code source, on a pris l'habitude d'utiliser le mot `pickle`. Cela ne pose aucun problème en utilisant la technique d'alias de nom de module. Voici comment l'utiliser :

```
import cPickle as pickle
```

Cette manière d'écrire permet d'utiliser le mot `pickle` pour faire référence au module `cPickle` dans toute la suite du code source. Rien ne t'empêche de rebaptiser un module avec `as truc` puis de faire des appels du style `truc. fonction()`. Mais si tu abuses de cette possibilité, tu rends ton programme illisible.

Les possibilités d'entreposage des objets dans un fichier pickle peuvent représenter un risque. Si quelqu'un parvient à te faire exécuter la méthode `pickle.load()` avec un fichier qui contient des objets qui lui appartiennent, il lui devient possible d'exécuter SON code sur TA machine. Il ne faut donc jamais charger un fichier pickle dont tu ne connais pas la provenance. Tu ne devras notamment jamais accepter d'utiliser des fichiers pickle obtenus sur le Web, ou un autre réseau.

Sauvons les objets

Nous allons ajouter une fonction pour sauver les objets dans un fichier, ce qui ne sera pas difficile puisqu'il suffira d'utiliser la méthode `dump()`. En revanche, les tests seront moins faciles.

Dans des projets plus sophistiqués, tu auras intérêt à prévoir des instructions pour demander à l'utilisateur de choisir le nom du fichier dans lequel il veut stocker les objets du carnet d'adresses.

Dans notre projet, le nom du fichier sera figé. Voici la procédure pour pouvoir sauver les objets :

1. **Nous ajoutons une section d'imports au début (avec `####`) et installons une directive d'import :**

```
import cPickle as pickle
```

2. **Nous ajoutons aussi une section de constantes pour y définir le nom de fichier :**

```
NOMFIC_SAUVE = "carnet.pickle"
```

3. **Dans le corps de la classe `CarnetAdr`, nous allons ajouter une méthode portant le nom `sauver()`.**
4. **Dans cette méthode, nous ajoutons une instruction pour ouvrir le fichier en écriture et pour y stocker le contenu :**

```
def sauver(self):
with open(NOMFIC_SAUVE,'w') as ficow:
pickle.dump(self, ficow)
```

5. **Il ne reste plus qu'à ajouter dans la section principale en fin de programme une instruction pour déclencher cette méthode via l'instance. L'appel s'écrit `carnet_adr.sauver()`.**

Voici les deux portions modifiées ou créées dans la version 6 de notre projet. Tout d'abord les sections des imports et des constantes, puis la classe `CarnetAdr` pour montrer sa nouvelle méthode `sauver()`.

Listing 9.6 : carnet.py (version 6), extrait

```
### Section Imports
import cPickle as pickle                #AJOUT
### Section Constantes
NOMFIC_SAUVE = "carnet.pickle"          #AJOUT
### Section Classes
class CarnetAdr(object):
""" Conteneur de fiches """
#AJOUT 2 methodes
```

```
def __init__(self):
        """ Cree l'attribut gens comme liste vide.
        """
        self.gens = []

    def ajouter_fiche(self, fiche_nouvo):
        """ Ajoute une fiche dans gens.
        """
        self.gens.append(fiche_nouvo)

#AJOUT methode sauver()
    def sauver(self):
        with open(NOMFIC_SAUVE,'w') as ficow:
            pickle.dump(self, ficow)
```

Nous n'oublions pas d'ajouter la ligne suivante à la fin de la section principale :

```
carnet_adr.sauver()
```

Ressuscitons les objets avec un contrôleur

Les instructions que nous venons de décrire pour sauver une instance du carnet d'adresses ont été regroupées dans une méthode de cette classe même. Cela revient donc pour l'objet issu de la classe à se sauver lui-même (tout le monde sait se sauver, même les microbes). Mais cela pose un problème. La méthode pour recharger l'objet n'existe qu'une fois l'objet présent en mémoire, donc rechargé. Pour ressusciter quelqu'un, il faut quelqu'un d'autre qui soit vivant, n'est-ce pas ? La question qui se pose est la suivante : à quel endroit placer le code qui va servir à ressusciter l'objet ?

Je te propose une solution radicale qui fera d'une pierre quatre coups. Nous allons définir une troisième classe qui va servir à contrôler le flux d'exécution du programme avec un menu. C'est elle qui se chargera de sauver les objets dans un fichier au moment où l'utilisateur demande de quitter le programme. Le code que nous allons écrire maintenant va donc être situé en dehors de la classe CarnetAdr. Nous pourrons ainsi exploiter cette classe de façon plus naturelle, de l'extérieur.

La nouvelle classe Kontroleur

Voici comment nous allons créer notre nouvelle classe de contrôle.

1. **Nous mettons en place une ligne de tête pour la nouvelle classe en lui donnant le nom `Kontroleur`.**
2. **Nous n'oublions pas de mettre en place immédiatement sa doc-chaîne, même s'il n'y a pas grand-chose à expliquer pour l'instant.**
3. **Nous implantons une méthode constructeur en écrivant ceci :**

```
def __init__(self) :
```

4. **Dans ce constructeur, nous créons une instance de `CarnetAdr` sous forme d'un attribut de la classe `Kontroleur`.**

```
self.carnet_adr = CarnetAdr()
```

5. **Commence alors une petite séance de recyclage de code. Nous allons, dans le bloc principal en bas, couper les deux lignes concernant l'objet, c'est-à-dire :**

```
contact1 = FicheAdr("Eric", "IDLE", "12/06/1999", None)
# Plus besoin de cette ligne :    print(contact1)
carnet_adr.ajouter_fiche(contact1)
```

   et les coller dans le corps de notre nouveau constructeur.

6. **Puisque l'objet appartient maintenant au contrôleur, il faut ajouter `self` en préfixe de la création de la fiche pour `contact1`, donc juste avant `carnet_adr` dans la dernière ligne ci-dessus.**

```
self.carnet_adr.ajouter_fiche(contact1)
```

7. **Il nous reste à définir une méthode `charger()` dans la même classe. C'était notre but initial.**

   Cette méthode va réutiliser l'instruction qui sert à sauver les instances. Les différences sont que nous ouvrons le fichier en lecture et injectons les données renvoyées par la méthode `load()` directement dans l'objet `carnet_adr` du contrôleur, donc dans `self.carnet_adr`. Bien sûr, la méthode `load()` ne fonctionne que si elle trouve dans le fichier mentionné un objet au format de la classe `CarnetAdr`. (Si tu n'as plus le fichier *.pickle*, tu peux en créer un depuis l'interpréteur comme indiqué plus haut.)

8. **Une dernière action dans la section principale : il faut demander la création d'une instance de la classe Kontroleur par `controleur = Kontroleur()` et modifier l'instruction d'affichage de la liste en préfixant avec le nom de l'objet : `print(controleur.carnet_adr. gens)`.**

J'ai appelé cette nouvelle classe `Kontroleur`, puisqu'elle va dorénavant contrôler tout ce qui se passe dans le projet par rapport au carnet d'adresses. Voici le texte source complet de la version 7 du projet.

Listing 9.7 : carnet .py (version 7)

```
"""
carnet.py  Version 7 (avec le Kontroleur)
Gestion de fiches de contacts avec
prenom, nom, naissance, courriel (email)
Brendan Scott (VF par OE)
"""

### Section Imports
import cPickle as pickle              #AJOUT

### Section Constantes
NOMFIC_SAUVE = "carnet.pickle"        #AJOUT

### Section Classes
class CarnetAdr(object):
""" Conteneur de fiches
"""
def __init__(self):
""" Cree l'attribut gens comme liste vide.
"""
self.gens = []
def ajouter_fiche(self, fiche_nouvo):
""" Ajoute une fiche dans gens.
"""
self.gens.append(fiche_nouvo)

def sauver(self):
with open(NOMFIC_SAUVE,'w') as ficow:
pickle.dump(self, ficow)

class FicheAdr(object):
""" Fiche d'un contact.
"""
    def __init__(self, prenom=None, nomfami=None,
                 datenaiss=None, courriel=None):
        """Initialise les 4 attributs. Le format de
        datenaiss est JJ/MM/AAAA
        """
        self.prenom    = prenom
        self.nomfami   = nomfami
```

```
        self.datenaiss = datenaiss
        self.courriel  = courriel

    def __repr__(self):
        """ Produit une chaine selon l'objet
        FicheAdr fourni en argument self.
        """
        patron = "FicheAdr(prenom ='%s', "+\
             "nomfami ='%s',"+\
             "datenaiss ='%s', "+\
             "courriel ='%s')"
        return patron%(self.prenom,self.nomfami,
                       self.datenaiss,self.courriel)

class Kontroleur(object):  #AJOUT de toute la classe
    """ Pour gerer les donnees stockees dans une
    instance de CarnetAdr, le menu et le chargement
    des donnees du fichier.
    """
    def __init__(self):
        """
        Initialise le controleur. Cherche un fichier.
        S'il le trouve, il le lit. Sinon, il cree un
       fichier vide.
        """
        self.carnet_adr = CarnetAdr()
        contact1 = FicheAdr("Eric", "IDLE", "12/06/1999",
None)
        self.carnet_adr.ajouter_fiche(contact1) #RECUP et MODIF

    def charger(self):
        """
        Charge un carnet depuis un fichier pickle.
        """
        with open(NOMFIC_SAUVE, 'r') as ficor:
            self.carnet_adr = pickle.load(ficor)

### Section Fonctions

### Section principale main
if __name__ == "__main__":
controleur = Kontroleur()           #AJOUT
print(controleur.carnet_adr.gens)   #MODIF
#SUPPR Autres lignes devenues inutiles
```

Ce qu'il faut surtout vérifier, c'est d'avoir modifié les références à l'objet `carnet_adr` pour qu'elles soient préfixées avec `self.` En effet, cet objet appartient dorénavant à un objet de la classe `Kontroleur`. J'ai également modifié l'instruction d'affichage tout à la fin pour que le carnet d'adresses soit bien trouvé dans le contrôleur. L'attribut `gens` appartient à l'objet `carnet_adr` qui lui-même appartient à l'objet `controleur`.

Il y a bien deux niveaux d'attribut. Le nombre de niveaux n'est pas limité, mais n'en abuse pas, car cela deviendrait difficile à suivre.

L'exécution de cette version donne ceci :

```
[FicheAdr(prenom ='Eric', nomfami ='IDLE',datenaiss
='12/06/1999', courriel ='None')]
```

Est-ce que ce test nous prouve que tout va bien ? En tout cas, il permet d'être certain que toutes les références concernant la nouvelle classe `Kontroleur` fonctionnent. En revanche, la nouvelle méthode `charger()` n'a encore jamais été appelée, et donc jamais testée. Voyons cela maintenant.

Testons la méthode charger()

Pour pouvoir tester la méthode `charger()`, il te faut un fichier contenant un objet « dans la saumure », et portant le nom *carnet.pickle*. Dans la version finale du projet, nous appellerons automatiquement cette méthode pour charger les objets au démarrage du programme. Mais il faut d'abord vérifier que le fichier existe. S'il n'est pas trouvé, il ne faut pas tenter d'appeler la méthode !

Pour tester l'existence d'un fichier, nous disposons d'une méthode standard nommée `exists()` dans le sous-module `os.path`. En donnant en argument à cette méthode un chemin d'accès ou un nom de fichier, elle répond qu'elle trouve ou pas le fichier désigné. Si tu as fait les essais précédents dans l'interpréteur, tu as créé le fichier dont nous avons besoin. Vérifions cela :

```
>>> NOMFIC_SAUVE = "carnet.pickle"
>>> os.path.exists(NOMFIC_SAUVE)
True
>>> os.path.exists("autre")
False
```

Si la méthode renvoie la valeur False, c'est que le fichier est introuvable. Vérifie le nom de fichier dans la section des constantes. Voici comment créer un nouveau fichier en exploitant le code de notre projet sous forme de module depuis l'interpréteur. Dans les deux premières lignes, le nom *carnet* est celui du fichier de la version 7 du code source du projet. Tu dois l'adapter si tu as choisi un autre nom. Dans mes exemples, le nom est *P1o_carnet_V7*) :

```
>>> from carnet import NOMFIC_SAUVE
>>> from carnet import CarnetAdr, FicheAdr
>>> contact1 = FicheAdr("p", "Nom", "31/06/2011", None)
>>> carnet_adr = CarnetAdr()
>>> carnet_adr.ajouter_fiche(contact1)
>>> carnet_adr.sauver()
>>> # Pour confirmer que le fichier existe :
>>> with open(NOMFIC_SAUVE,'r') as ficor:
print(ficor.read())
```

N. d. T. : le fait d'utiliser avec `import` notre propre projet provoque la création dans le même dossier d'un fichier précompilé à extension *.pyc*.

Suite à ces essais, nous allons pouvoir enrichir notre classe `Kontroleur` pour qu'elle vérifie qu'il y a bien un fichier à charger au démarrage et pour le charger seulement si ce fichier existe. Voici comment nous allons procéder :

1. **Nous ajoutons dans la section d'importation la mention `import os.path`.**
2. **Nous ajoutons l'appel à la fonction de chargement en tant que première ligne dans le constructeur de la classe `Kontroleur`, c'est-à-dire dans `__init__` :**

   ```
   self.carnet_adr = self.charger()
   ```

   Nous allons rendre la méthode `charger()` plus fiable : elle ne doit essayer de charger le fichier que s'il existe. S'il n'y a pas de fichier, on renvoie la pseudo-valeur None.
3. **Toujours dans le constructeur, nous ajoutons un test avec `if` pour créer l'objet s'il n'a pas pu être rechargé depuis le fichier :**

   ```
   self.carnet_adr = CarnetAdr()
   ```

4. **Dans la méthode `charger()`, nous ajoutons un test pour vérifier que le fichier existe.**

5. **Si le fichier est trouvé, nous modifions l'instruction d'appel à la fonction de chargement pour renvoyer la valeur qu'elle renvoie elle-même directement avec `return`. Dans le cas contraire, nous renvoyons None.**

La nouvelle section `import` de la version 8 comporte deux lignes :

```
### Section Imports
import cPickle as pickle
import os.path
```

Voici un extrait de la version 8 du projet montrant la totalité du code source de la classe `Kontroleur`. Il n'y a pas eu de modifications ailleurs :

Listing 9.8 : Extrait de la version 8 de carnet.py (la classe Kontroleur)

```
class Kontroleur(object):  #AJOUT de toute la classe
    """ Pour gerer les donnees stockees dans une
    instance de CarnetAdr, le menu et le chargement
    des donnees du fichier.
    METHODES : __init__, charger()
    """
    def __init__(self):
        """
        Initialise le controleur. Cherche un fichier.
        S'il le trouve, il le lit. Sinon, il cree un
       fichier vide.
        """
        self.carnet_adr = self.charger()        #AJOUT
        if self.carnet_adr is None:             #AJOUT
            self.carnet_adr = CarnetAdr()       #MODIF
        # contact1 = FicheAdr("Eric", "IDLE", "12/06/1999",
None)
        # self.carnet_adr.ajouter_fiche(contact1)

    def charger(self):
        """
        Charge un carnet depuis un fichier pickle.
        """
        if os.path.exists(NOMFIC_SAUVE):           #AJOUT
            with open(NOMFIC_SAUVE, 'r') as ficor: #MODIF
                return pickle.load(ficor)          #MODIF
        else:                                      #AJOUT
            return None                            #AJOUT
```

Pour bien faire, il faut tester le programme lorsque le fichier est présent, mais également lorsqu'il n'est plus accessible, par exemple en changeant une lettre dans son nom.

Et voici le menu !

Il reste trois méthodes à ajouter à notre contrôleur, la plus importante étant celle qui va gérer l'affichage d'un menu pour que l'utilisateur choisisse une lettre parmi quatre correspondant à des commandes. Il pourra ainsi ajouter une fiche, lister les fiches, afficher le menu et quitter.

1. **Dans la section des constantes, nous commençons par mettre en place le texte qui va servir d'instructions ainsi qu'un message demandant de confirmer la sortie :**

```
QUIT_CONFIRMER = "Vous confirmez vouloir quitter (O/N)? "
INSTRUCTIONS = """
******************************************
Application Carnet d'Adresses
(Python en s'amusant Pour les Nuls, Projet 9)
******************************************
```

```
Tapez une des quatre touches suivantes :
A pour Ajouter une personne
L pour la Liste des fiches du carnet
I pour revoir ces Instructions
Q pour Quitter.
"""
```

2. **Relis bien ces instructions, car tu auras besoin toi aussi de tester les quatre commandes.**
3. **Dans le constructeur de contrôleur, nous ajouterons tout à la fin un appel à la nouvelle méthode `self.gerer_menu()`.**
4. **Nous implantons cette nouvelle méthode dans le contrôleur en commençant par ceci :**

```
def gerer_menu(self):
```

5. **Dans cette méthode, nous commençons par afficher les instructions puis entrons dans une boucle infinie basée sur `while True :`. Le programme sera en permanence en train d'aller vers une méthode puis de revenir dans cette boucle. Voici ce qui doit se passer dans chaque tour de boucle :**

   Nous invitons l'utilisateur à taper la lettre de la commande qu'il désire avec la fonction classique `raw_input()` et nous analysons sa saisie.

   Pour réagir à la réponse, nous entrons dans une grande structure conditionnelle `if/elif/else` pour tester chaque commande.

   Pour la lettre a, nous ajoutons un appel à une méthode `ajouter_ fiche()` que nous allons écrire juste après. Pour la lettre l, nous ajoutons un appel à `lister_fiches()`. Voici par exemple comment nous gérons la réponse à la commande d'ajout :

   ```
   if cmd == "a" ior cmd == "A": ### minuscule ou
   Majuscule
   self.ajouter_fiche()
   ```

   Si l'utilisateur tape q pour quitter, nous allons appeler une fonction pour lui demander de confirmer. L'astuce consiste à réutiliser le code source de la fonction qui nous a servi à cela dans le Projet 5. Je donnerai les détails un peu plus loin. Si l'utilisateur confirme, nous affichons un message pour dire que nous allons sauver les données (appel à la méthode `sauver()` du carnet d'adresses) avant de sortir par `break` et de provoquer la fin du programme.

   Si l'utilisateur a tapé une touche non reconnue, nous le lui faisons remarquer dans la branche `else` à la fin du bloc conditionnel. Nous indiquons quelle touche a été refusée.

6. **Nous créons enfin une amorce de méthode pour les deux nouvelles méthodes `ajouter_fiche()` et `lister_fiches()`.**

Chaque amorce doit comporter sa doc-chaîne et pour l'instant une instruction d'affichage pour vérifier que la méthode répond à la commande correspondante. Voici par exemple ce que nous écrirons pour `ajouter_fiche()` :

```
def ajouter_fiche(self):
# Demande la saisie des champs de la nouvelle fiche
print("Dans ajouter_fiche() de Kontroleur")
```

Rien n'a changé au niveau des deux classes `CarnetAdr` et `FicheAdr`, et rien non plus dans la section `import` et dans la section principale. Je ne présente donc que des extraits de cette nouvelle version 9.

Listing 9.9 : carnet.py (version 9) : les constantes

```
### Section Constantes (ajout de 2 constantes)
NOMFIC_SAUVE    = "carnet.pickle"
```

```
QUIT_CONFIRMER = “Vous confirmez vouloir quitter (O/N)? “
INSTRUCTIONS = “””
*****************************************
Application Carnet d’Adresses
(Python en s’amusant Pour les Nuls, Projet 9)
*****************************************
Tapez une des quatre touches suivantes :
A pour Ajouter une personne
L pour la Liste des fiches du carnet
I pour revoir ces Instructions
Q pour Quitter.
“””
```

Le second extrait présente la totalité de la nouvelle version de la classe `Kontroleur`, avec la nouvelle méthode `gerer_menu()` et les amorces des deux autres méthodes, toutes implantées après la méthode `charger()`. Une seule ligne a été ajoutée ailleurs : à la fin du constructeur `__init__`, pour appeler `gerer_menu()`.

Listing 9.10 : carnet.py (version 9) : la classe Kontroleur

```
class Kontroleur(object):
“”” Doc-chaine abrégée     “””
def __init__(self):
“”” Doc-chaine abrégée     “””
self.carnet_adr = self.charger()
if self.carnet_adr is None:
self.carnet_adr = CarnetAdr()

self.gerer_menu()                #AJOUT V9

def charger(self):
“”” Doc-chaine abrégée     “””
if os.path.exists(NOMFIC_SAUVE):
with open(NOMFIC_SAUVE, ‘r’) as ficor:
return pickle.load(ficor)
else:
return None

# TROIS nouvelles methodes
def gerer_menu(self):
# Boucle centrale de l’application.
# Obtient la commande clavier et lance l’action.
print(INSTRUCTIONS)
while True:
cmd = raw_input(“(a)jout,(l)iste,(i)nstr.,(q)
uitter : “)
if cmd == “a” or cmd == “A”:  ### minuscule ou
Majuscule
self.ajouter_fiche()
elif cmd == “q” or cmd == “Q”:  ### minuscule ou
Majuscule
if confirmer_quitter():
print(“Sauvegarde”)
self.carnet_adr.sauver()
print(“Fin de l’application”)
break
elif cmd == “i” or cmd == “I”:  ### minuscule ou
Majuscule
print(INSTRUCTIONS)
elif cmd == “l” or cmd == “L”:  ### minuscule ou
Majuscule
self.lister_fiches()
```

```
else:
modele = "*** Touche de commande inconnue (%s)
!"
print(modele%cmd)

def ajouter_fiche(self):
# Demande la saisie des champs de la nouvelle fiche
print("Dans ajouter_fiche() de Kontroleur")

def lister_fiches(self):
# Affiche la liste des fiches du carnet
print("Dans lister_fiches() de Kontroleur")
```

Et voici la nouvelle fonction `confirmer_quitter()` récupérée du Projet 5 et collée dans la section des fonctions. Si tu n'as pas fait le Projet 5, tu peux récupérer cette fonction dans le fichier de mes exemples portant le nom *JeuDevin4.py*.

Listing 9.11 : carnet.py (version 9) : la fonction confirmer_quitter ()

```
### Section Fonctions (une reprise du Projet 5)
def confirmer_quitter():
""" On sort seulement si saisie de la
lettre n minuscule par renvoi de False. """
confi = raw_input(QUIT_CONFIRMER)
if confi == "n"' or confi == "N":  ### minuscule ou
Majuscule:
return False
else:
return True
```

Pour tester cette version, il y a lieu de taper chacune des quatre commandes pour vérifier que c'est la bonne méthode qui est appelée. Voici ce qui doit se produire selon la commande :

- a (ou A) doit afficher « Dans ajouter_fiche() de Kontroleur »
- l (ou L) doit afficher « Dans lister_fiches() de Kontroleur »
- i (ou I) doit afficher les instructions du menu
- q (ou Q) doit demander confirmation pour quitter

Teste aussi d'autres lettres pour forcer le programme à se plaindre.

Si tu es observateur, tu auras remarqué qu'il existe maintenant deux méthodes portant le nom `ajouter_fiche()`, la première définie dans la classe `CarnetAdr` et l'autre dans la classe `Kontroleur`. Elles ne se gênent pas car le nom de la classe ajouté en préfixe permet de les distinguer. D'ailleurs, je vais me servir de l'ancienne méthode `ajouter_fiche()` en insérant un appel à celle-ci dans son homonyme dans le contrôleur.

Ajoutons deux méthodes

Pour terminer notre projet, il nous reste à donner du corps (des instructions) aux deux méthodes en attente.

La méthode ajouter_fiche()

Cette méthode doit insérer une fiche dans le carnet d'adresses avec des valeurs. Il nous faut donc récupérer les valeurs pour les quatre champs prénom, nom, date de naissance et adresse de messagerie.

1. **Nous commençons par informer l'utilisateur de ce qui va se passer :**

```
print(“Ajout d’une fiche dans le carnet”)
print(“Description de la personne :”)
```

2. **Nous demandons à l’utilisateur de saisir les données par des appels à `raw_input()`. Voici par exemple pour le prénom :**

```
prenom = raw_input(“Son prenom ? “)
```

3. **Nous ajoutons dans chaque retour de saisie un test pour savoir si l’utilisateur veut abandonner en ayant frappé seulement la lettre q. Si c’est le cas, nous n’insérons pas la fiche, et rendons le contrôle par `return`. Voici comment nous procéderons pour le prénom (nous répétons le code en changeant le nom de la variable) :**

```
if prenom == “q”:
print(“On abandonne !”)
return
```

4. **Nous récupérons ensuite les quatre valeurs pour construire une fiche que nous ajoutons à l’objet `carnet_adr`. Dans l’extrait suivant, il n’y a que deux lignes, même si la première est très longue :**

```
nfiche = FicheAdr(prenom, nomfami,
courriel, datenaiss)
self.carnet_adr.ajouter_fiche(nfiche)
```

5. **Nous terminons par un affichage de contrôle.**

La méthode lister_fiches()

Pour la méthode `lister_fiches()`, nous utilisons une énumération pour balayer toutes les fiches trouvées dans l’objet `carnet_adr`.

Chacun des objets est une instance de `FicheAdr`.

1. **Commençons par définir un patron de format dans la section des constantes :**

```
PATRON_LISTE = “%s %s Naissance: %s Email: %s”
```

2. **Nous descendons ensuite dans la nouvelle méthode pour installer une boucle `for` exploitant ce patron pour contrôler le format des quatre attributs correspondant aux données. Nous stockons le résultat dans une variable temporaire, puis nous faisons afficher un indice suivi de l’entrée avec une petite chaîne de format du style `“% s : % s”`. Cela va permettre d’afficher une ligne numérotée par fiche. Voici les quatre instructions de cette boucle de listage :**

```
for indice, e in enumerate(self.carnet_adr.gens):
contact = (e.prenom,e.nomfami,e.datenaiss,e.
courriel)
fiche = PATRON_LISTE%contact
print(“%s: %s”%(indice + 1, fiche))
```

Finition du projet

La dernière retouche consiste à supprimer une ligne. Facile. Tout à la fin de la section principale, nous supprimons la ligne d’affichage :

```
print(controleur.carnet_adr.gens)
```

Cet affichage ne servait que pour la mise au point. La section principale est devenue extrêmement réduite : nous créons une instance de la classe `Kontroleur` c’est tout ! Le flux d’exécution n’est plus le même qu’au début du projet. Une fois que le contrôleur a été créé, nous donnons le contrôle à son constructeur puis de

là, à sa méthode `gerer_menu()`. On ne quitte cette méthode que pour quitter l'application. Tu rencontreras ce genre de structure assez souvent lorsque tu étudieras des projets écrits en Python.

Code source du projet complet

Voici la version finale (version 10) du projet. Le code source s'étend sur plusieurs pages, mais tu devrais facilement trouver tes repères dorénavant.

Listing 9.12 : carnet.py (version10 finale)

```
"""
carnet.py  Version 10 (finale)
Gestion de fiches de contacts avec
prenom, nom, naissance, courriel (email)
Brendan Scott (VF par OE)
"""

### Section Imports
import cPickle as pickle
import os.path              #AJOUT

### Section Constantes (ajout de 2 constantes)
NOMFIC_SAUVE   = "carnet.pickle"
QUIT_CONFIRMER = "Vous confirmez vouloir quitter (O/N)? "
INSTRUCTIONS = """
*****************************************
Application Carnet d'Adresses
(Python en s'amusant Pour les Nuls, Projet 9)
*****************************************
Tapez une des quatre touches suivantes :
A pour Ajouter une personne
L pour la Liste des fiches du carnet
I pour revoir ces Instructions
Q pour Quitter."""
PATRON_LISTE = "%s %s Naissance: %s Email: %s"

### Section Classes
class CarnetAdr(object):
    """ Conteneur de fiches
        METHODES : __init__,ajouter_fiche(), sauver()
    """
    def __init__(self):
        """ Cree l'attribut gens comme liste vide.
        """
        self.gens = []

    def ajouter_fiche(self, fiche_nouvo):
        """ Ajoute une fiche dans gens.
        """
        self.gens.append(fiche_nouvo)

    def sauver(self):
        with open(NOMFIC_SAUVE,'w') as ficow:
            pickle.dump(self, ficow)

class FicheAdr(object):
    """ Fiche d'un contact.
    METHODES : __init__, __repr__
    """
    def __init__(self, prenom=None, nomfami=None,
                 datenaiss=None, courriel=None):
```

```
        """Initialise les 4 attributs. Le format de
        datenaiss est JJ/MM/AAAA
        """
        self.prenom    = prenom
        self.nomfami   = nomfami
        self.datenaiss = datenaiss
        self.courriel  = courriel

    def __repr__(self):
        """ Produit une chaine selon l'objet
        FicheAdr fourni en argument self.
        """
        patron = "FicheAdr(prenom ='%s', "+\
             "nomfami ='%s',"+\
             "datenaiss ='%s', "+\
             "courriel ='%s')"
        return patron%(self.prenom,self.nomfami,
                       self.datenaiss,self.courriel)

class Kontroleur(object):
    """ Pour gerer les donnees stockees dans une
    instance de CarnetAdr, le menu et le chargement
    des donnees du fichier.
    METHODES : __init__, charger(), gerer_menu(),
    ajouter_fiche(), lister_fiches
    """
    def __init__(self):
        """
        Initialise le controleur. Cherche un fichier.
        S'il le trouve, il le lit. Sinon, il cree un
       fichier vide.
        """
        self.carnet_adr = self.charger()
        if self.carnet_adr is None:
            self.carnet_adr = CarnetAdr()

        self.gerer_menu()               #AJOUT

    def charger(self):
        """
        Charge un carnet depuis un fichier pickle.
        """
        if os.path.exists(NOMFIC_SAUVE):           #AJOUT
            with open(NOMFIC_SAUVE, 'r') as ficor: #MODIF
                return pickle.load(ficor)          #MODIF
        else:                                      #AJOUT
            return None                            #AJOUT

    def gerer_menu(self):
        # Boucle centrale de l'application.
        # Obtient la commande clavier et lance l'action
        print(INSTRUCTIONS)
        while True:
            cmd = raw_input("\n(a)jout,(l)iste,(i)nstr.,(q)
uitter : ")
            if cmd == "a" or cmd == "A":  ### minuscule ou
Majuscule
                self.ajouter_fiche()
            elif cmd == "q" or cmd == "Q":  ### minuscule ou
Majuscule
                if confirmer_quitter():
                    print("Sauvegarde")
                    self.carnet_adr.sauver()
                    print("Fin de l'application")
```

```
                    break
            elif cmd == “i” or cmd == “I”:  ### minuscule ou
Majuscule
                print(INSTRUCTIONS)
            elif cmd == “l” or cmd == “L”:  ### minuscule ou
Majuscule
                self.lister_fiches()
            else:
                modele = “*** Touche de commande inconnue (%s)
 !”
                print(modele%cmd)

    def ajouter_fiche(self): #Methode rendue active
        # Demande la saisie des champs de la nouvelle fiche
        print(“Ajout d’une fiche dans le carnet”)
        print(“Description de la personne :”)
        prenom = raw_input(“Son prenom ? “)
        if prenom == “q”:
            print(“On abandonne !”)
            return

        nomfami = raw_input(“Son nom de famille ? “)
        if nomfami == “q”:
            print(“On abandonne !”)
            return

        M_DATENAISS = “Sa date de naissance (Jour/Mois/An) ? “
        datenaiss = raw_input(M_DATENAISS)
        if datenaiss == “q”:
            print(“On abandonne !”)
            return

        courriel = raw_input(“Son @dresse ? “)
        if courriel == “q”:
            print(“On abandonne !”)
            return

        nfiche = FicheAdr(prenom, nomfami, courriel, datenaiss)
        self.carnet_adr.ajouter_fiche(nfiche)
        ncontact = (prenom, nomfami)
        print(“Nouvelle fiche faite pour %s %s.\n”%ncontact)

    def lister_fiches(self):
        “”” Affiche la liste des fiches du carnet “””
        print(“\nListe des personnes”)
        for indice, e in enumerate(self.carnet_adr.gens):
            contact = (e.prenom, e.nomfami,
                       e.datenaiss, e.courriel)
           fiche = PATRON_LISTE%contact
            print(“%s: %s”%(indice + 1, fiche))
            # Le comptage commence a 1

### Section Fonctions
def confirmer_quitter():
    “”” On sort seulement si saisie de la
    lettre n minuscule par renvoi de False. “””
    confi = raw_input(QUIT_CONFIRMER)
    if confi == “n” or confi == “N”:  ### minuscule ou Majuscule
        return False
    else:
        return True

### Section principale main
if __name__ == “__main__”:
```

```
    controleur = Kontroleur()
#    print(controleur.carnet_adr.gens)  #SUPPR V10
```

Récapitulons

Ouf ! C'était un projet sérieux, n'est-ce pas ? Ce projet donne une bonne base pour créer une application de carnet d'adresses, en y ajoutant notamment la possibilité de modifier une fiche et d'en supprimer. Tu peux adjoindre le projet de cryptage pour rendre des notes secrètes, permettre de trier la liste dans l'ordre alphabétique, ajouter d'autres champs, calculer l'âge de chaque contact à partir de la date de naissance (cela suppose d'utiliser le module `datetime`). Quand tu auras appris comment utiliser les modules pour créer les programmes à interface graphique, tu pourras réutiliser ce code.

Faisons un tour d'horizon de tout ce que nous avons découvert dans ce projet :

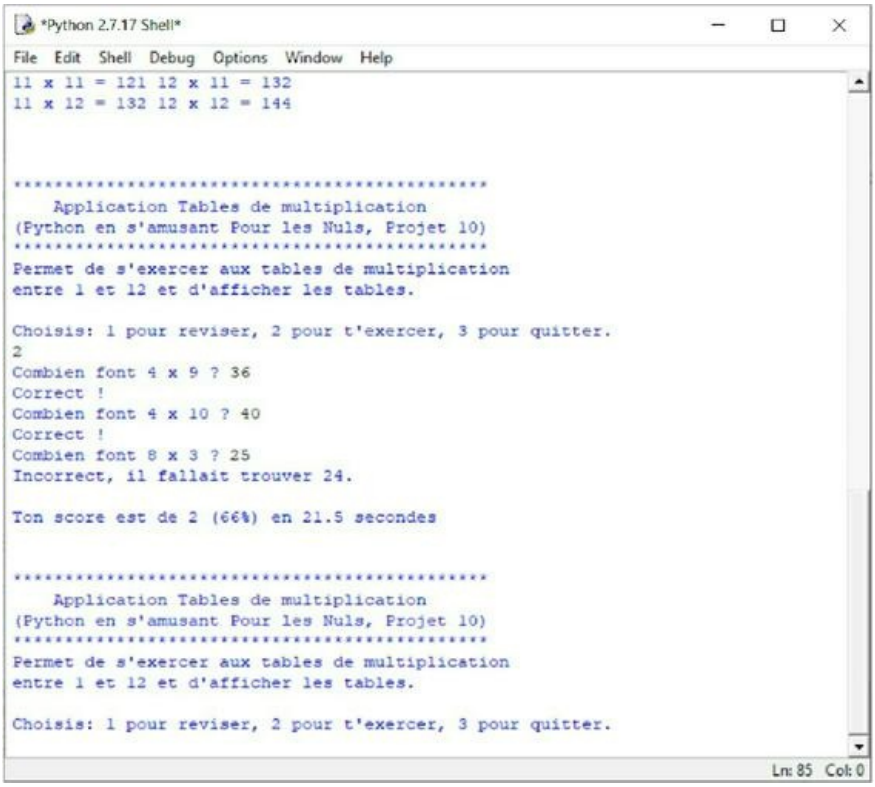
- Comment définir des objets et des classes.
- Comment définir des classes à partir desquelles on peut créer des objets.
- Comment créer des objets sous forme d'instance.
- Les classes possèdent des attributs qui peuvent être des données ou des méthodes.
- Les différences entre attributs d'une classe et attributs d'une instance.
- Comment une instance peut hériter des attributs de sa classe, mais pas l'inverse.
- Le constructeur qui est la méthode portant le nom `__init__`. Elle est exécutée à chaque création d'une instance.
- Comment transmettre des arguments à `__init__` pour donner des valeurs par défaut à un nouvel objet.
- Comment redéfinir des éléments, et notamment la méthode `__repr__` pour contrôler l'affichage des données d'un objet.
- Comment définir de nouvelles méthodes pour une classe.
- Comment l'élément `self` est transmis automatiquement comme premier argument quand on appelle une méthode de classe.
- Comment sauver et recharger des objets Python grâce au module `pickle`.
- Le module `os.path` permet de vérifier qu'un fichier existe.
- Nous avons enfin utilisé des techniques déjà connues : boucles conditionnelles `if/elif/else` et l'ouverture des fichiers, `enumerate`, `raw_input()` et les chaînes de format.

Projet 10

Multiplions !

N'aimerais-tu pas jouer au professeur à ton tour ? Dans ce projet, nous allons proposer à l'utilisateur de réviser ses tables de multiplication et de faire des exercices. C'est un jeu, car nous gérons un score et nous chronométrons le temps mis pour répondre.

Tu vas pouvoir rafraîchir tes souvenirs de tables de multiplication, et faire ainsi un peu de gymnastique cérébrale.



Définissons nos objectifs

Notre jeu de table va demander de répondre à des multiplications choisies dans les tables de 1 jusqu'à 12 (pas jusqu'à 10, et c'est le défi). Le jeu vérifie que chaque réponse est correcte.

Nous allons prévoir un affichage de toutes les tables. Tu pourras modifier le code source pour n'afficher que les premières tables, ou bien aller jusqu'à la table des 12.

Nous ajouterons finalement un compteur de score et un mécanisme pour chronométrer la durée de chaque exercice. À table !

Commençons en douceur

Nous créons le fichier source et nous commençons par quelques instructions pour poser une question et vérifier qu'elle est correcte ou pas.

Voici comment nous allons procéder pour débuter :

1. **Création d'un nouveau fichier projet portant le nom** *tablesmul.py*.

2. **Rédaction de la doc-chaîne du fichier.**
3. **Mise en place immédiate des titres de sections : imports, constantes, fonctions et tests.**

   Le projet des phrases en folie t'a appris que quand tu cites le nom d'un tuple après l'opérateur de formatage `%`, Python va déballer le tuple et répartir les valeurs trouvées dans les spécificateurs de format de la chaîne de format. Autrement dit, l'instruction `"Combien font % s x % s"% (4,6)` donne "`Combien font 4 x 6 ?`". Chaque question va être stockée en tant que tuple regroupant les deux chiffres à multiplier (les opérandes).

4. **Nous prévoyons le texte d'une question. Tu vas créer une constante portant le nom Q_TEST dans la section des constantes. Pour le moment, les valeurs seront figées. Nous choisissons deux nombres et faisons en sorte que la constante Q_TEST soit égale au tuple qui contient les deux nombres. Je propose en guise d'exemple `(4,6)`.**
5. **Nous devons ensuite définir une autre constante pour le patron de format de la question. Nous choisirons le nom PATRON_Q.**

   C'est ce patron qui va servir de base aux textes affichés pour la question `"Combien font % s x % s"`.

6. **Nous implantons une variable portant le nom `question` et nous y stockons le texte de la constante Q_TEST.**

   C'est cette variable qui va servir dans les instructions. Dans une version plus avancée, nous créerons une liste de questions et les afficherons l'une après l'autre. Nous y parviendrons en changeant la valeur de la variable `question`. Le programme fonctionnera avec les nouvelles valeurs sans réclamer d'autres retouches.

7. **Dans la section des tests, nous ajoutons une instruction de demande de saisie avec un texte d'invite. Comme texte, nous utilisons le contenu de la variable question que nous injectons dans la chaîne de format PATRON_Q.**
8. **Nous ajoutons ensuite une instruction pour calculer la bonne réponse en multipliant `question[0]` par `question[1]`.**
9. **Nous utilisons le texte d'invite utilisé lors de la demande de saisie `raw_input()`.**

   Nous récupérons dans une autre variable ce que le joueur a saisi devant l'instruction de demande de saisie.

10. **Nous convertissons la réponse saisie en une valeur numérique entière avec `int()`.**
11. **Il ne reste plus qu'à comparer la réponse fournie par l'utilisateur à la réponse calculée et afficher un message adéquat.**

Voici le code source correspondant à toute la procédure :

Listing 10.1 : tablesmul.py (version 1)

```
"""
tablesmul.py Version 1
Les tables de multiplication.
Brendan Scott (VF par OE)
"""

#### Section des constantes
Q_TEST   = (4, 6)
PATRON_Q = "Combien font %s x %s ? "

#### Section des fonctions

#### Section principale (et de test)
question = Q_TEST
invite   = PATRON_Q%question
repon_ok = question[0]*question[1]      # On commence a 0
repon_saisie = raw_input(invite)
if int(repon_saisie)== repon_ok:
```

```
print(“Correct !”)
else:
print(“C’est faux.”)
```

Voici ce que donne l’exécution de cette première version :

```
Combien font 4 x 6 ? 24
Correct !
>>> ============================== RESTART ==================
>>>
Combien font 4 x 6 ? 35
C’est faux.
```

Il faut lancer au moins deux tests, un en répondant correctement et l’autre en se trompant volontairement.

Créons nos questions

On pourrait faire tirer les questions au hasard avec la fonction `random.randint()` ou même `random.choice()`. Mais cela pose un problème : tu ne pourras jamais être sûr que toutes les tables seront pratiquées. Le hasard peut faire que jamais la table des huit ne soit visitée, par exemple.

Mieux vaut générer toutes les questions possibles, c’est-à-dire 12 * 12 ou 144 questions, puis choisir au hasard dans cette liste. Il suffit de mémoriser les questions déjà posées pour puiser parmi celles qui restent.

Dans le Projet 2, j’avais prévenu qu’on pouvait utiliser l’instruction `range` dans Python 2, mais qu’elle occupait beaucoup de mémoire. Dans notre projet, le nombre de données est important, mais reste raisonnable. Nous pouvons continuer à utiliser `range`.

Voici comment nous allons créer la liste des questions :

1. **Dans la section des fonctions, nous commençons la définition de la fonction `creer_listeq()` avec sa doc-chaîne.**

2. **Il serait intéressant que les exercices soient à géométrie variable pour pouvoir adapter le nombre de tables au niveau scolaire du joueur. Pour y parvenir, nous prévoyons deux arguments pour les limites inférieure et supérieure des tables à pratiquer. Nous en profitons pour donner des valeurs par défaut aux deux arguments `nmin` et `nmax` :**

```
def creer_listeq(nmin = MINI, nmax = MAXI):
```

   Nous utilisons des constantes, en écrivant par exemple `nmin = MINI`. Le mot en capitales est la constante, et le mot en minuscules est la variable dont la fonction va se servir.

3. **Nous arrivons à une partie un peu plus délicate : une double liste par compréhension pour créer la liste des tuples que nous renvoyons par `return` :**

```
return [(x+1, y+1) for x in range(nmin-1, nmax)
for y in range(nmin-1, nmax)]
```

4. **Nous pouvons ensuite neutraliser toute la section des tests pour l’instant. Il ne faut pas la supprimer, car nous allons en avoir besoin plus tard.**

5. **Toujours la section des tests, nous insérons une instruction pour appeler la fonction que nous venons de définir et une autre pour afficher ce qu’elle va renvoyer. Je rappelle que j’ai ajouté deux constantes pour éviter le phénomène de nombres magiques (des valeurs qui apparaissent sans qu’on comprenne à quoi elles correspondent) :**

```
MINI = 1
MAXI = 12
```

Le listing complet suivant montre notamment la nouvelle fonction `creer_listeq()` :

Listing 10.2 : tablesmul.py (version 2)

```
"""
tablesmul.py Version 2 (et 2.1)
Les tables de multiplication
Brendan Scott (VF par OE)
"""

#### Section des constantes
Q_TEST = (4, 6)
PATRON_Q = "Combien font %s x %s ? "
MINI = 1                              #AJOUT V2
MAXI = 12                             #AJOUT V2

#### Section des fonctions
def creer_listeq(nmin = MINI, nmax = MAXI):       #AJOUT V2
""" Produit une liste de questions au format (x,y)
avec x et y entre MINI et MAXI (compris)
"""
return [(x+1, y+1) for x in range(nmin-1, nmax)
for y in range(nmin-1, nmax)]

#### Section principale (et de test)
# question = Q_TEST
# invite = PATRON_Q%question
# repon_ok = question[0]*question[1]     # On commence a 0
# repon_saisie = raw_input(invite)
# if int(repon_saisie)== repon_ok:
#     print("Correct !")
# else:
#     print("C'est faux.")

liste_q = creer_listeq(nmin,nmax)
print(liste_q)
```

Il n'y a en fait qu'une seule instruction dans la fonction, la double liste par compréhension. Son écriture n'est pas très simple parce que l'expression `range(nmin, nmax)` croît du minimum au maximum sans inclure ce maximum. Ce n'est pas ce que nous voulons. Nous voulons que la dernière table soit elle aussi affichée. C'est ce qui explique que les deux valeurs résultantes soient augmentées de 1 dans les deux tuples : `(x+1, y+1).` C'est pour la même raison que la limite inférieure comporte une réduction de un dans les deux listes par compréhension.

La section des tests, si l'on oublie la partie neutralisée, ne contient dorénavant que deux lignes.

Pendant l'exécution, tu vois s'afficher 144 tuples allant de (1,1) à (12,12). C'est ce que nous voulions.

Le test prouve que la fonction sait travailler avec les deux valeurs par défaut. Il n'est pas inutile de vérifier avec quelques autres paires. Pour y parvenir, il suffit d'ajouter une boucle de répétition dans la section des tests.

1. **On choisit trois valeurs pour `nmin` et trois autres pour `nmax` et on construit des couples.**
2. **On appelle trois fois de suite la fonction de création de la liste des questions en transmettant les deux valeurs qui changent à chaque tour.**
3. **On affiche à chaque tour ce qu'on a récupéré de la fonction.**

Voici la nouvelle variante de cette section de test :

```
for nmin,nmax in [(2, 5), (4, 6), (7, 11)]:
    liste_q = creer_listeq(nmin,nmax)
    print(liste_q)
```

Revenons sur le fonctionnement de cette boucle `for`. Elle visite le contenu d'une liste de trois éléments, chaque élément étant un tuple de deux éléments. Ces éléments sont « déballés » dans les deux variables `nmin` et `nmax`, en conservant bien l'ordre. Pour chaque couple de valeurs, nous appelons la fonction puis affichons la liste de questions générées. Voici à quoi ressemble l'exécution de cette version :

```
[(2, 2), (2, 3), (2, 4), (2, 5), (3, 2), (3, 3), (3, 4), (3,
5), (4, 2), (4, 3), (4, 4), (4, 5), (5, 2), (5, 3), (5, 4),
(5, 5)]
[(4, 4), (4, 5), (4, 6), (5, 4), (5, 5), (5, 6), (6, 4), (6,
5), (6, 6)]
[(7, 7), (7, 8), (7, 9), (7, 10), (7, 11), (8, 7), (8, 8),
(8, 9), (8, 10), (8, 11), (9, 7), (9, 8), (9, 9), (9, 10), (9,
11), (10, 7), (10, 8), (10, 9), (10, 10), (10, 11), (11, 7),
(11, 8), (11, 9), (11, 10), (11, 11)]
```

Posons bien nos questions

Où en sommes-nous ? Nous avons généré une liste de questions et nous savons comment poser une des questions et récupérer la réponse. Il ne reste plus qu'à bombarder le joueur en enchaînant les questions en rafale. Mais si nous envoyons les questions dans l'ordre actuel de la liste, nous allons avoir deux petits soucis :

> » La liste est actuellement dans l'ordre croissant des tables, et les questions vont vite devenir trop prévisibles. Mieux vaut les proposer dans un ordre aléatoire.
>
> » La liste contient 144 questions. À moins d'être particulièrement méchant, tu ne veux sans doute pas obliger le joueur à répondre d'une traite aux 144 questions. Il est préférable de n'en poser que quelques-unes à la fois.

Nous allons résoudre ces deux problèmes en un seul coup de cuillère à pot.

Mélangeons la liste des questions

Voici comment nous allons mélanger la liste pour que les questions ne soient plus dans l'ordre des tables.

1. **Tout en haut du fichier, dans la section des imports, nous ajoutons une directive pour importer le module `random`.**
2. **Dans la ligne de tête de la fonction `creer_listeq()`, nous ajoutons un troisième argument facultatif en lui donnant le nom `hasard` (je n'ai pas choisi par hasard). Il sera de type logique (soit True, soit False). Comme valeur par défaut, si l'argument n'est pas fourni, nous choisissons True. Avec False, les questions restent dans l'ordre naturel.**
3. **Nous pouvons mettre à jour la doc-chaîne pour expliquer à quoi sert le nouvel argument.**
4. **Nous récupérons la liste une fois mélangée dans une variable de travail purement locale. Son nom peut donc être choisi sans trop réfléchir. J'ai choisi `tempo`.**
5. **Nous allons ensuite nous servir de la valeur de l'argument `hasard` dans une boucle conditionnelle.**

   Lorsqu'une variable contient une valeur logique (True ou False), le simple fait de citer son nom permet d'exploiter sa valeur. Voilà pourquoi on peut écrire directement `if hasard:`. Si cela te semble bizarre, tu peux écrire la version complète comme ceci :

   ```
   if hasard is True:
   ```

   Si la variable vaut True, nous faisons appel à la fonction standard de mélange par `random.shuffle(tempo)`. Ici, nous n'avons pas besoin de seconde branche `else :` . Si tu en as envie, tu peux basculer dans l'interpréteur

pour obtenir quelques informations au sujet de cette nouvelle fonction en écrivant `help(random.shuffle)`. Je t'avertis cependant qu'il y a une petite astuce.

6. **Il ne reste plus qu'à renvoyer la variable contenant la liste mélangée.**

Voici à quoi ressemble la nouvelle section des importations :

```
#### Section des imports
import random
```

Voici un extrait de la nouvelle version du projet en version 3, ne montrant que la fonction `creer_listeq()` :

Listing 10.3 : extrait de tablesmul.py (Version 3)

```
def creer_listeq(nmin = MINI, nmax = MAXI, hasard=True):
#MODIF V3
""" Doc-chaine abrégée     """
tempo = [(x+1, y+1) for x in range(nmin-1, nmax)
for y in range(nmin-1, nmax)]
if hasard is True:
random.shuffle(tempo)
return(tempo)
```

Ne touchons pas à la section des tests. Pour l'instant, il est plus facile de faire les vérifications avec un sous-ensemble des tables.

Voici ce qui apparaît lors de l'exécution :

```
[(3, 4), (5, 4), (4, 2), (5, 5), (3, 2), (2, 3), (5, 2), (4,
4), (2, 4), (3, 5), (2, 2), (4, 3), (3, 3), (2, 5), (4, 5),
(5, 3)]
[(4, 4), (5, 6), (4, 6), (5, 5), (4, 5), (6, 5), (6, 6), (6,
4), (5, 4)]
[(9, 11), (10, 8), (11, 11), (7, 8), (10, 10), (9, 10), (11,
10), (8, 11), (7, 7), (9, 8), (10, 7), (7, 9), (8, 10), (7,
11), (8, 9), (9, 9), (11, 9), (8, 8), (8, 7), (10, 11), (11,
7), (7, 10), (9, 7), (11, 8), (10, 9)]
```

Dans ce cas précis, ce que tu verras sur ton écran sera évidemment différent, puisque l'ordre a été défini par hasard. La version non mélangée doit commencer par `(2,2)`.

Notre nouvelle variable locale `hasard` possède par défaut la valeur vraie (True). Nous n'avons donc testé que cette possibilité pour l'instant. Il faudrait également tester le cas où `hasard` vaut False. Dans ce cas, la fonction doit renvoyer la liste inchangée. Il suffit de modifier le code qui appelle la fonction en transmettant la valeur False comme troisième argument, puis en vérifiant qu'elle renvoie bien la liste dans l'ordre naturel :

```
liste_q = creer_listeq(nmin, nmax, False)
```

Des questions, mais pas trop

Pour enchaîner plusieurs questions, il suffit de mettre en place une boucle. Nous en profitons pour définir une variable qui va mémoriser le score. Voici comment répondre à ces deux besoins :

1. **Dans la section des tests, nous neutralisons les instructions qui servaient à tester le mélange de la liste dans `creer_listeq()`.**
2. **Dans la section des constantes, nous ajoutons une constante qui va déterminer le nombre maximal de questions à poser : `Q_MAX = 3`. Pour l'instant, nous n'enchaînons par trop de questions, car nous sommes**

en train de faire des tests.

3. **En descendant dans la section des fonctions, nous ajoutons une nouvelle fonction sans argument d'entrée et lui donnons le nom `faire_test()`.**

4. **Dans le corps de cette fonction, nous créons une variable qui va mémoriser le score et lui donnons la valeur initiale zéro. Donner une première valeur à une variable s'appelle *initialiser*.**

5. **Toujours dans la même fonction, nous ajoutons une ligne permettant de construire une liste de questions. C'est ici que nous allons appeler la fonction `creer_listeq()`.**

6. **Nous entrons ensuite dans une boucle utilisant `enumerate` pour parcourir tout le contenu de `listeq`.**

```
for i, question in enumerate(liste_q):
```

7. **Lors de chaque tour de boucle, nous récupérons un nombre dans `i` et un tuple pour la question provenant de la liste. Le nombre `i` va servir d'indice pour savoir où nous en sommes dans la progression dans la liste, en commençant à zéro.**

   Lors de chaque tour, nous testons si l'indice est devenu supérieur ou égal à la constante Q_MAX. Lorsque c'est le cas, c'est qu'il faut arrêter de poser des questions. Nous utilisons alors `break` pour sortir de la boucle.

8. **Comme dans le Projet 9, nous avons maintenant l'occasion de faire un peu de recyclage de code. Ici, il s'agit de récupérer sept lignes de la version 1 de ce projet. Pour savoir lesquelles, tu regarderas le listing complet un peu plus loin. Ce sont les lignes de la section principale des tests. Il faut les décommenter, oublier la première et recoller le tout à la fin de la fonction `faire_test()`. Il faut ensuite penser à indenter toutes ces lignes d'un niveau pour qu'elles fassent partie du corps de la boucle `for`.**

9. **Dans la branche conditionnelle qui indique que la réponse est correcte, nous ajoutons une instruction pour augmenter `score` de 1 et nous l'affichons une fois sortis de la boucle.**

10. **Il ne reste plus dans qu'à ajouter dans la section principale des tests un appel à la nouvelle fonction `faire_test()`.**

Dans le listing de la version 4 qui suit, il faut remarquer la nouvelle constante et la nouvelle fonction `faire_test()` avec son code recyclé. La section principale et des tests doit être entièrement vidée, sauf l'appel à notre nouvelle fonction `faire_test()`.

Listing 10.4 : tablesmul.py (Version 4)

```
""" tablesmul.py Version 4 (...) """

#### Section des imports
import random

#### Section des constantes
Q_TEST   = (4, 6)
PATRON_Q = "Combien font %s x %s ? "
MINI = 1
MAXI = 12
Q_MAX = 3                    #AJOUT V4 (modifier pour tester)

#### Section des fonctions
def creer_listeq(nmin = MINI, nmax = MAXI, hasard=True):
#MODIF V3
""" (abrégé)     """
tempo = [(x+1, y+1) for x in range(nmin-1, nmax)
for y in range(nmin-1, nmax)]
if hasard:
random.shuffle(tempo)
return(tempo)

def faire_test():                    #AJOUT V4
```

```
""" Lance une passe de test """
liste_q = creer_listeq()
score = 0
for i, question in enumerate(liste_q):
if i >= Q_MAX:
break
invite = PATRON_Q%question
repon_ok = question[0]*question[1]
# Les indices commencent a 0
repon_saisie = raw_input(invite)

if int(repon_saisie) == repon_ok:
print("Correct !")
score = score+1
else:
print("Incorrect, c'etait %s."%(repon_ok))

print("Nombre de calculs corrects : %s"%score)

#### Section principale (et de test)
# Onze lignes supprimees
faire_test()
```

Là où nous utilisons `enumerate`, nous aurions pu écrire `for i in range(Q_MAX):`. Mais une boucle `for` provoquerait une erreur si Q_MAX est supérieur au nombre de questions disponibles dans la liste. La technique basée sur `enumerate` permet de s'arrêter à la fin de la liste, quelle que soit la valeur trouvée dans Q_MAX. De plus, `enumerate` est plus proche de l'esprit Python.

Voici ce que donne l'exécution de cette version 4 :

```
Combien font 2 x 12 ? 24
Correct !
Combien font 1 x 2 ? 2
Correct !
Combien font 1 x 7 ? 5
Incorrect, il fallait trouver 7.
Nombre de calculs corrects : 2
```

Affichons plusieurs tables

Afficher une seule table est facile, puisque nous disposons déjà d'un couple du style `(4,6)` et de l'instruction pour calculer la réponse et l'afficher dans un format correct. Mais pour afficher plusieurs tables, il faut prendre une décision. Voulons-nous afficher chaque table sur une ligne ou regrouper plusieurs tables côte à côte ? La première approche consomme beaucoup d'espace. Si on place plusieurs tables côte à côte, il faut connaître la largeur disponible dans la fenêtre. Ce sont des problèmes d'interface utilisateur. Pour y répondre, il nous faut d'abord écrire ce qu'il faut pour afficher toutes les tables.

1. **Nous définissons une constante qui servira de patron pour chaque table. Ce patron doit permettre d'afficher trois nombres selon le format suivant :**

```
#LIGNE_TMUL = "%s x %s = %s"
```

2. **Il nous faut une fonction pour afficher les tables. Nous insérons dans la section des fonctions la ligne de tête de la fonction `afficher_tablesmul()`, avec sa doc-chaîne. Elle attend un seul argument nommé `nmax` qui sera par défaut égal à la constante MAXI qui vaut 12 pour l'instant :**

```
def afficher_tablesmul(nmax = MAXI):
```

Tu pourras en fin de projet augmenter la valeur de cette limite supérieure, mais sache que les tables deviennent vite très volumineuses.

3. **Nous mettons en place deux boucles `for` imbriquées.**

   Chacune des deux boucles aura comme butée supérieure la valeur de la variable locale `nmax`. Les variables de travail seront nommées `x` et `y`, et contiendront les deux nombres à multiplier.

```
for x in range(nmax):
for y in range(nmax):
```

4. **Dans la boucle intérieure basée sur `y`, nous exploitons la chaîne du patron pour créer la chaîne à afficher avec les deux nombres et le résultat de leur multiplication puis nous l'affichons.**

   Voici comment nous allons construire chaque ligne à partir du patron :

```
ligmul = LIGNE_TMUL%(x+1, y+1, (x+1)*(y+1))
```

5. **Nous mettons en commentaires l'instruction qui se trouve dans la section des tests.**
6. **Nous ajoutons dans cette section un appel à notre nouvelle fonction.**

Voici un extrait de la version 5 du projet montrant la nouvelle fonction et la section principale (et de tests). Le seul autre changement est la constante de patron.

Listing 10.5 : tablesmul.py (version 5), extrait

```
def afficher_tablesmul(nmax = MAXI):            #AJOUT V5
# Affiche la table des mult. jusque MAXI
for x in range(nmax):
for y in range(nmax):
ligmul = LIGNE_TMUL%(x+1, y+1, (x+1)*(y+1))
print(ligmul)

#### Section principale (et de test)
# faire_test()               #MODIF V5
afficher_tablesmul()         #AJOUTV5
```

Si tu oublies d'ajouter par exemple `1` à `y` dans la ligne remplissant `ligmul`, les tables commencent à 1 fois 0 et se terminent à 1 fois 11.

Voici un extrait de ce qui s'affiche pendant l'exécution :

```
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3

## Lignes omises, c'est long !

12 x 9 = 108
12 x 10 = 120
12 x 11 = 132
12 x 12 = 144
```

Tout fonctionne, sauf que les colonnes sont mal alignées à cause des nombres sur un, deux ou trois chiffres.

Nous avions découvert le spécificateur `%s` dans le projet des phrases en folie. Pour les valeurs numériques, mieux vaut utiliser `%i`, car on peut préciser le nombre minimum de chiffres. Par exemple, `%2i` fait afficher les valeurs sur deux positions et `%3i` sur trois, même si le nombre n'a qu'un chiffre. C'est la technique du remplissage par la gauche.

Pour les deux opérateurs, nous utilisons une largeur de deux puisque les tables vont jusqu'à 12. Pour le résultat, il nous faut trois positions parce que le plus grand résultat sera 144. Voici le patron de formatage revisité :

```
LIGNE_TMUL = "%2i x %2i = %3i"
```

Une fois faite cette simple retouche, l'exécution donne ceci (affichage écourté) :

```
1 x  1 =   1
1 x  2 =   2
1 x  3 =   3

## Lignes omises, c'est long !

12 x  8 =  96
12 x  9 = 108
12 x 10 = 120
12 x 11 = 132
12 x 12 = 144
```

C'est beaucoup plus agréable ! Tout est bien aligné du côté droit. Bien sûr, si tu demandes d'afficher des tables pour des valeurs encore plus grandes, l'affichage ne sera plus aligné.

Il est possible d'écrire des instructions pour gérer n'importe quelle largeur sans que cela mette en péril l'alignement, mais en programmation, comme dans toutes choses, il faut savoir faire des compromis. Pour notre projet, nous n'irons pas au-delà de 12. Ne perdons pas de temps.

Affichons plusieurs tables en largeur

L'affichage des tables forme une liste interminable. Tout l'espace du côté droit est inutilisé. Ce serait mieux de pouvoir afficher plusieurs tables côte à côte, mais il faut d'abord savoir combien d'espace occupe chaque table en largeur. Il est facile d'obtenir la longueur d'un objet en nombre de caractères, pour une chaîne ou une liste, au moyen de la fonction standard `len()`. Essayons-la dans la fenêtre de l'interpréteur Shell en créant d'abord une variable d'essai pour en demander sa longueur :

```
>>> LIGNE_TMUL = "%2i x %2i = %3i"
>>> ligmul = LIGNE_TMUL%(12,12,144)
>>> len(ligmul)
13
```

D'après ce test, `12 x 12 = 144` occupe 13 caractères. Il faut y ajouter une espace pour séparer les tables, soit 14. Les programmes considèrent que l'écran ou la fenêtre leur offre 70 caractères en largeur. Il suffit de diviser 70 par 14 pour obtenir 5. Nous pouvons donc afficher cinq tables côte à côte.

Pour cette nouvelle version compacte, il faut revoir entièrement la fonction `afficher_tablesmul`.

1. **Dans la section des constantes, nous ajoutons une simple espace à la fin de la chaîne LIGNE_TMUL :**

```
LIGNE_TMUL = "%2i x %2i = %3i "
```

2. **Dans la fonction `afficher_tablesmul`, nous définissons une variable locale `tab_par_lig` et nous lui donnons la valeur initiale 5.**

3. **Nous produisons une liste de toutes les tables à afficher :**

```
tab_total = range(1, nmax+1)
```

4. **Nous retranchons du réservoir `tab_total` le nombre de tables que nous voulons afficher (indiqué dans `tab_par_lig`), et nous restockons ce qui reste à traiter dans `tab_total`.**

```
lot_tab = tab_total[:tab_par_lig]
tab_total = tab_total[tab_par_lig:]
```

5. **Nous entrons ensuite dans une boucle `while` qui travaille avec notre lot de tables à afficher :**

```
while lot_tab != []:       # Stop quand vide
```

6. **Nous mettons en place une sous-boucle `for` qui va aller de un à la valeur `nmax+1` (pour traiter les nombres de un à la limite, limite comprise).**

```
for x in range(1, nmax+1):
```

7. **Dans la boucle `for`, nous créons la liste vide `accumul` qui va stocker une ligne de tables.**

   En créant la liste à cet endroit, elle est recréée à chaque tour de boucle.

8. **Nous créons un troisième niveau de boucle avec `for` pour traiter `y`, le second opérande.**

9. **Dans cette troisième boucle (attention aux indentations), nous stockons une ligne de tables dans la liste `accumul` :**

```
accumul.append(LIGNE_TMUL%(y, x, x*y))
```

10. **Nous remontons ensuite d'un niveau, celui de la boucle `for x` pour joindre le contenu de l'accumulateur à une chaîne vide "" avant de l'afficher :**

```
print("".join(accumul))
```

11. **Nous remontons traiter un autre lot de tables et vidons un peu la table servant de réservoir, comme dans l'étape 4 (sauf que nous indentons).**

Dans la version 6, il y a donc une espace de plus à la fin de la constante LIGNE_TMUL et une fonction `afficher_tablesmul()` entièrement refaite, à tel point qu'il vaut mieux supprimer son contenu antérieur et reprendre celui proposé ci-dessous.

Listing 10.6 : tablesmul.py (version 6), extrait

```
def afficher_tablesmul(nmax = MAXI): #REFONTE V6
# Affiche la table des mult. jusque MAXI
tab_par_lig = 5
tab_total = range(1, nmax+1)
# Prend un lot de 5 pour affichage
lot_tab = tab_total[:tab_par_lig]
# Les enleve du stock restant
tab_total = tab_total[tab_par_lig:]
while lot_tab != []: # Stop quand vide
for x in range(1, nmax+1):
# Lignes vont de 1 a 12
accumul = []
for y in lot_tab:
# Construit les colonnes
accumul.append(LIGNE_TMUL%(y, x, x*y))
print("".join(accumul)) # Affiche une ligne
print("\n") # Separation verticale entre blocs de
tables
# Lot suivant
lot_tab = tab_total[:tab_par_lig]
tab_total = tab_total[tab_par_lig:]
```

Notre boucle `while` puise dans le lot de tables (de 1 à 12) par paquet de cinq. La boucle `y` construit chaque ligne de cinq tables. Pour la première ligne, `x+1` vaut 1 et `y` va de 1 à 5, ce qui donne 1 x 1, 2 x 1,

jusqu'à 5 x 1. Pour la deuxième ligne, `x+1` vaut 2 et `y` va toujours de 1 à 5, ce qui donne 2 x 1, 2 x 2, etc. On progresse ainsi jusqu'à avoir affiché toutes les tables.

Figure 10.1 : Affichage multi-colonnes

N'oublions pas l'utilisateur

Pour l'instant, nous n'avons travaillé que sur la partie traitement des données, la partie en coulisses, celle que l'utilisateur ne voit pas (à part l'affichage des tables pour les tests). Il ne peut pas contrôler le fonctionnement du programme. Il n'y a aucune instruction et l'utilisateur ne peut pas décider de faire des exercices ou de réafficher les tables pour réviser.

Même si nous travaillons en mode texte, il s'agit d'une interface utilisateur. Il faut toujours prendre soin de la conception de cette interface pour que le programme soit facile à utiliser. Il est inutile de faire un programme fantastique s'il est trop difficile à contrôler. Inversement, combien d'applications soignent trop l'apparence de l'interface au détriment de la qualité du traitement des données. Il faut trouver un juste milieu.

Puisque nous travaillons avec du texte, l'interface utilisateur reste simple. Cela évite de perdre du temps dans le peaufinage des boutons radio et fenêtres. Nous restons concentrés sur le contrôle de l'exécution. Voici l'interface que doit offrir notre programme d'entraînement.

- Le programme va d'abord se présenter puis afficher les trois options : révision, exercices ou sortie. Pour permettre le choix entre ces options, nous décidons de faire taper un des trois chiffres 1, 2 ou 3.
- Nous demandons ensuite à l'utilisateur de saisir son choix.
- Nous transmettons le contrôle à la partie du programme correspondante.
- Chaque sous-programme doit rendre le contrôle à la partie principale. En revenant toujours au niveau du programme principal, l'utilisateur peut décider de faire une deuxième révision, ou de se lancer dans un nouvel exercice.

Bonne nouvelle : la plupart des instructions requises existent déjà pour la partie révision et les exercices. Voici la procédure pour ajouter cette interface :

1. **Commençons par ajouter une constante nommée INSTRUCTIONS en lui donnant comme valeur le texte sur plusieurs lignes qui explique comment utiliser le programme.**

   Tu peux considérer qu'il s'agit d'une doc-chaîne en délimitant le texte par des triples guillemets.

Pour le contenu des instructions, sers-toi de mon modèle. Tu peux aussi laisser la partie vide, pour la reprendre plus tard. Voici mon exemple :

```
INSTRUCTIONS = """
*****************************************
Application Tables de multiplication
(Python en s'amusant Pour les Nuls, Projet 10)
*****************************************
Permet de s'exercer aux tables de multiplication
entre 1 et 12 et d'afficher les tables.
"""
```

2. **Dans la section des tests, nous neutralisons toutes les instructions éventuelles par # et ajoutons dans le titre qu'il s'agit de la section principale.**
3. **Dans cette section, nous débutons un bloc conditionnel pour utiliser le programme comme module :**

```
if __name__ == "__main__":
```

4. **Nous installons dans ce bloc une boucle infinie `while True:` .**
5. **Dans ce bloc, nous affichons les instructions puis un message d'invite qui propose de taper 1, 2 ou 3. La réponse saisie au clavier est stockée dans une variable nommée `selection` (pas d'accent dans les noms de variables !)** Nous n'ajouterons la fonction pour quitter que dans la prochaine version. Pour l'instant, il faudra utiliser Ctrl+C pour quitter le programme.
6. **Nous devons analyser ce que l'utilisateur a saisi. Nous nettoyons la saisie grâce à la méthode standard `strip()` en écrivant `selection = selection.strip()`. Cela permet de supprimer les espaces éventuelles en début et en fin de saisie.**
7. **Si `selection` ne contient ni 1, ni 2, ni 3 (ce sont des chaînes), nous demandons à l'utilisateur de choisir à nouveau. Nous restons dans cette boucle jusqu'à ce que l'utilisateur se décide pour l'une des trois options possibles.**
8. **Nous pouvons mettre en place l'amorce pour la fonction `quitter()`. Les deux autres fonctions dont nous avons besoin existent déjà : `faire_test()` et `afficher_tablesmul()`.**
9. **Nous créons déjà la doc-chaîne de la nouvelle fonction. Nous ajoutons pour l'instant une seule instruction d'affichage pour vérifier que nous parvenons à y entrer.**
10. **Dans le bloc principal, nous mettons en place un bloc conditionnel `if/elsif/else` pour appeler les trois fonctions selon le choix de l'utilisateur.**

La version 7 du projet contient donc une nouvelle constante pour les instructions, une amorce pour la fonction `quitter()` (qui ne fait que signaler son existence par un message), une section de test totalement vide et la partie principale suivante :

Listing 10.7 : tablesmul.py (version 7), extrait

```
#### Section principale "main"
if __name__ == "__main__":
while True:
print(INSTRUCTIONS)
menu_txt = "Choisis: 1 pour reviser,"+\
" 2 pour t'exercer, 3 pour quitter.\n"
menu_choix = raw_input(menu_txt)
menu_choix = menu_choix.strip()
```

```
while menu_choix not in [“1”, “2”, “3”]:
menu_choix = raw_input(“Seulement 1, 2 ou 3: “)
menu_choix = menu_choix.strip()
if menu_choix == “1”:
afficher_tablesmul()
elif menu_choix == “2”:
faire_test()
else: # Doit etre 1, 2 ou 3, donc c’est 3 pour quitter
quitter()
```

Il faut exécuter quatre fois le programme pour bien tester chacune des trois options et le cas de la frappe d’une autre touche que celles des trois chiffres. Je rappelle qu’il faut utiliser la combinaison clavier Ctrl+C pour quitter.

Ne nous quittons pas sans quitter

Il ne reste plus qu’à écrire les instructions de la fonction `quitter()`, ce qui ne devrait pas être difficile puisque nous avons déjà utilisé ce genre de fonction dans les projets 5 et 9.

1. **Nous ajoutons d’abord la directive `import sys`. Elle nous servira pour pouvoir utiliser la fonction `exit()` qui va forcer la fin d’exécution.**
2. **Nous récupérons ensuite la fonction `confirmer_quitter()` du Projet 5 (*JeuDevin4.py*) et nous l’installons dans la section des fonctions.**
3. **N’oublions pas de copier ou de définir la constante dont `confirmer_quitter()` a besoin. Elle porte le nom QUIT_CONFIRMER.**
4. **Dans la nouvelle fonction, nous appelons `confirmer_quitter()`. Si l’utilisateur confirme qu’il veut quitter, nous appelons la fonction `sys.exit()`. Dans le cas contraire, nous ne faisons rien. La fonction rend le contrôle et nous reprenons au niveau de la boucle du menu.**

Voici la nouvelle section import :

```
#### Section des imports
import random
import sys
```

Voici la constante récupérée du Projet 5 (il peut être plus rapide de la ressaisir) :

```
QUIT_CONFIRMER = “Es-tu certain de vouloir quitter (O/n) ?”
```

J’ai souvent envie de placer mes constantes dans l’ordre alphabétique, mais je pense que ce n’est pas une bonne idée. Mieux vaut les grouper par fonction. MINI et MAXI doivent logiquement se trouver l’une après l’autre, par exemple.

Voici l’extrait de la version 8 du projet montrant la fonction `quitter()` et celle que nous avons récupérée du Projet 5 :

Listing 10.8 : tablesmul.py (version 8), extrait

```
def quitter():
“”” Pour quitter l’application”””
if confirmer_quitter():                          #AJOUT V8
sys.exit()
print(“Dans quitter(), mais on fait demi-tour.”) #MODIF V8

#AJOUT V8 Provient de JeuDevin4.py
def confirmer_quitter():
“”” On sort seulement si saisie de la
lettre n minuscule par renvoi de False. “””
```

```
confi = raw_input(QUIT_CONFIRMER)
if confi == 'n':
return False
else:
return True
```

Nous laissons l'instruction d'affichage dans `quitter()` parce que nous avons besoin de faire encore des tests. L'affichage ne se produit que si on décide d'abandonner la sortie. La sortie se fait par l'appel à `sys.exit()`.

Si tu lances l'exécution depuis l'atelier IDLE (pas de panique, c'est comme cela que nous faisons depuis le Projet 4), il peut arriver que tu tombes sur un message d'erreur système, du style "`Traceback (most recent call last): File ...sys.exit() SystemExit`". C'est lié à la façon dont IDLE intègre les scripts avec l'interpréteur. Si tu lances le programme dans le terminal (Python Command Line) avec `python nomprog.py`, Python ne se plaindra plus.

Le temps passe...

Il ne reste qu'à ajouter de quoi chronométrer les performances arithmétiques du joueur.

> » Nous allons ajouter des instructions pour démarrer puis pour arrêter le chronomètre dans la fonction `faire_test()` et calculer le temps passé par soustraction.
>
> » Nous utiliserons ce temps dans l'affichage des statistiques de la partie et calculerons un pourcentage de bonnes réponses.
>
> » Il ne restera plus qu'à augmenter le nombre de questions par partie et à faire un peu de nettoyage en enlevant les instructions d'affichage superflues.

Tu n'as presque plus besoin de mes explications pour y arriver, non ?

Chronométrons la partie

Dès qu'on a besoin de chronométrer dans Python, on appelle à la rescousse le module standard nommé `time`. Vite, essayons ce nouvel animal dans l'interpréteur :

```
>>> import time
>>> time.time()     # Renvoie l'heure courante
1442234788.395
```

Ces 1442 milliards et des poussières, c'est l'heure courante ? Ils ont d'étranges montres, les pythons. C'est le nombre de secondes depuis le moment choisi comme date de naissance du système d'exploitation, appelé *Epoch*. Sous Linux et Mac OS, c'est le 1er janvier 1970 à minuit et sous Windows, le 1er janvier 1601 (mais n'en déduis pas que Windows est un aussi vieux système).

Le module `time` permet donc de mesurer une durée entre deux événements :

```
>>> t1 = time.time()  # Heure actuelle
>>> t2 = time.time()  # Idem
>>> t2 - t1           # Nombre de secondes entre les appels
8.202999830245972
```

En plus de `time()`, tu pourras avoir besoin de `ctime()` et de `gmtime()` pour convertir vers une chaîne et vers un tuple. Va voir dans l'aide.

Passons à la réalisation de ce chronométrage.

1. **Nous ajoutons une directive dans la section Imports pour le module `time` par `import time`.**
2. **Dans la fonction `faire_test()`, nous interrogeons l'heure avec `time.time()` et la stockons dans `t_debut`:**

3. **Juste avant d'afficher le score, nous interrogeons l'heure une seconde fois et stockons le renvoi dans une autre variable puis faisons la soustraction.**

4. **Nous affichons le temps passé avec le score.**

Nous en profitons pour calculer le pourcentage de réussite aux questions (le score divisé par Q_MAX multiplié par 100). Le résultat sera plus avenant avec un patron que nous ajoutons aux constantes. Le double formateur `%%` permet d'afficher le symbole %. Le formateur `%.1f` demande de traiter la valeur comme un numérique à virgule avec un seul chiffre après la virgule :

```
PATRON_SCORE = "Ton score est de %s (%i%%) en %.1f secondes\n"
```

Dans cette version finale, nous avons donc une troisième directive d'import, une constante de patron et nous avons retouché la fonction `faire_test()` que je reproduis en entier ici :

Listing 10.9 : tablesmul.py (version 9 finale), extrait

```
def faire_test():
""" Lance une passe de test """
liste_q = creer_listeq()
score = 0
t_debut = time.time()                                #AJOUT
V9
for i, question in enumerate(liste_q):
if i >= Q_MAX:
break
invite = PATRON_Q%question
repon_ok = question[0]*question[1]
        # Les indices commencent a 0
        repon_saisie = raw_input(invite)

        if int(repon_saisie) == repon_ok:
            print("Correct !")
            score = score+1
        else:
            print("Incorrect, il fallait trouver %s."%(repon_
ok))
    t_fin = time.time()                                  #AJOUT
V9
    t_total = t_fin - t_debut                            #AJOUT
V9
    pourcent_juste = int(score/float(Q_MAX)*100)         #AJOUT
V9
    print(PATRON_SCORE%(score, pourcent_juste, t_total))
#AJOUT V9

    # print("Nombre de calculs corrects : %s"%score)
#SUPPR V9
```

Finition et nettoyage

L'heure est venue de faire des essais et de retoucher quelques constantes, notamment le nombre de questions par partie. Tu es libre de faire ce que tu veux.

1. **Dans la section des constantes, on peut augmenter Q_MAX pour poser 10 ou 20 questions par partie. N'exagère pas non plus.**

2. **Tu peux enlever (supprimer, même) les instructions d'affichage qui ne servaient que pour les tests et toutes les instructions actuellement neutralisées. Vive le code propre !**

Un bon exercice : l'affichage des INSTRUCTIONS est pour l'instant dans la boucle principale. Le joueur va vite ne plus en avoir besoin. Tu peux créer une quatrième option (chiffre 4) pour les afficher seulement à la

demande.

Récapitulons

Voici ce que nous avons appris au cours de ce projet :

- Mélanger le contenu d'une liste avec `random.shuffle()`.
- Construire une interface utilisateur en mode texte avec des instructions, et la possibilité pour l'utilisateur de choisir parmi plusieurs options.
- Chronométrer l'utilisation d'une fonction avec le module `time`.
- Exploiter une table comme un réservoir de données en extrayant des portions dans une boucle `while`.
- Construire des boucles de répétition et conditionnelles à trois niveaux.

Tu peux dorénavant ouvrir n'importe quel fichier de code source Python et te sentir en terrain connu. Bonne chance dans tes aventures pythonesques !

Annexe

DANS CETTE ANNEXE

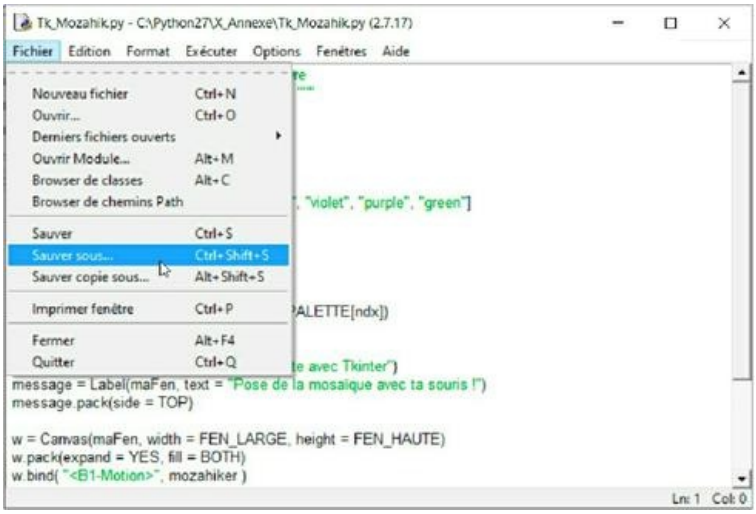
Annexe

Encore cinq choses

N.d.T. : cette annexe n'existe que dans la version française du livre.

L'atelier IDLE en français

Python est un langage vraiment plus facile d'accès que d'autres, sans sacrifier à la qualité des apprentissages fondamentaux (à la différence du langage BASIC, selon de grands chercheurs). Pourtant, comme l'interface utilisateur de l'atelier IDLE est en anglais, tu risques de ne pas t'y sentir chez toi, alors que tu vas y consacrer des heures.



Après avoir cherché qui avait travaillé sur une version française de IDLE, nous te proposons de profiter d'une version française presque complète (les rares éléments restés en anglais auraient obligé à modifier beaucoup plus de fichiers).

Voici par exemple l'aspect général de la fenêtre de l'éditeur IDLE avec le menu Fichier ouvert, puis avec le menu Format ouvert.

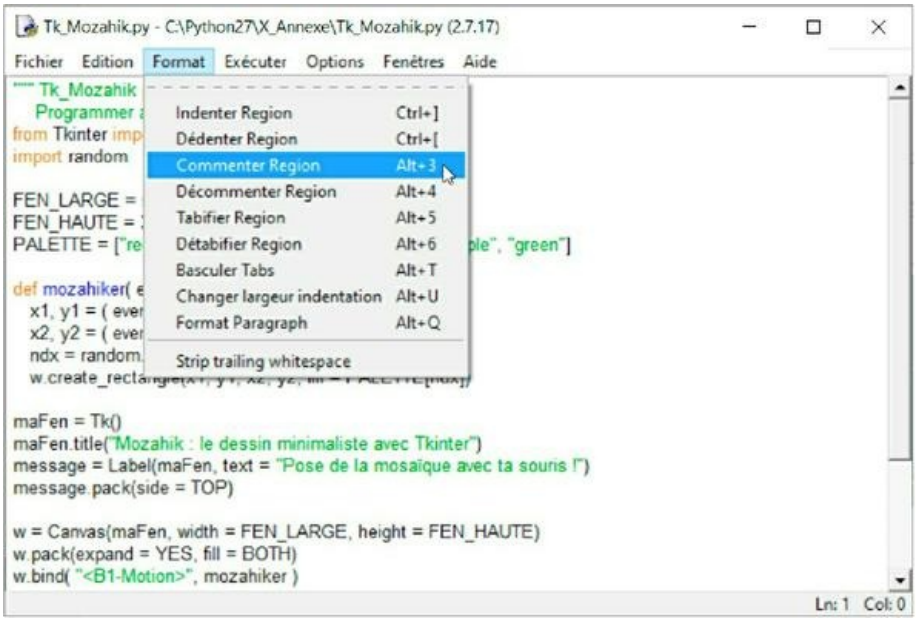


Figure A.1 : La fenêtre de l'éditeur IDLE en français.

Voici les noms des neuf fichiers concernés et à quoi chacun sert dans IDLE.

Fichier dans `idlelib`	Domaine fonctionnel
`AutoComplete.py`	Mécanisme de suggestion de saisie
`Bindings.py`	Menus de la fenêtre de l'interpréteur Shell
`configDialog.py`	Boîte de configuration des couleurs, etc.
`EditorWindow.py`	Menus de la fenêtre d'édition de source
`PyShell.py`	Menus de la fenêtre de l'interpréteur Shell
`ReplaceDialog.py`	Commandes de recherche et remplacement
`ScriptBinding.py`	Commandes pour les modules
`SearchDialog.py`	Commandes de recherche
`SearchDialogBase.py`	Commandes de recherche

Pour en profiter, il te suffit `d'aller voir dans le répertoire IDLE_FR, sous le dossier qui contient tous les exemples de ce livre !`

Il contient ces neufs fichiers en version française ainsi que des explications détaillées.

Attention : il n'est pas certain que cette opération de françisation fonctionne si les options de sécurité sur ta machine le bloquent.

Pour information, voici où se trouve par défaut le sous-dossier de Python dans lequel il faut intervenir.

Système	Sous-dossier standard
Windows	`C:\Python27\Lib\idlelib`
Mac OS	`/Library/Frameworks/Python.framework/Versions/3.4/lib/ python3.4/idlelib/`
Linux Ubuntu	`usr/lib/python2.7/idlelib`

Mieux vaut se faire aider par un adulte averti pour intervenir dans ce dossier.

Normalement, le fichier lié (extension .pyc) est régénéré automatiquement si le fichier .py est plus récent que lui. Si cela ne se produit pas, il suffit de supprimer le fichier .pyc (sans se tromper de fichier).

Le hit-parade des étourderies

Se tromper dans les niveaux d'indentation

Il vaut mieux ne jamais prendre l'habitude d'utiliser la touche TAB pour descendre d'un niveau. En revanche, la touche Ret. Arr. est parfaite pour remonter d'un niveau dans IDLE.

Confondre = et ==

L'opérateur de stockage (affectation) `=` et celui de comparaison `==` n'ont rien à voir. Le premier effectue une copie de données en mémoire. L'autre renvoie soit True (1), soit False (0), les deux seules valeurs booléennes possibles.

Se tromper dans la ponctuation

Les lignes de tête des définitions, des boucles et des conditions se terminent toujours par un signe deux-points (`:`). Attention aussi aux parenthèses : `(1+2) * (3+4)`, `1 + ((2*3) +4)` et `1 + (2* (3+4))` donnent trois résultats différents que je te laisse trouver toi-même !

Se tromper de 1 dans les répétitions

La butée supérieure du générateur `range ()` n'est jamais comprise dans le résultat. Si tu crées une boucle utilisant `range(1,5)`, tu ne feras que 4 tours de boucle.

Mal gérer l'écriture DroMaDaire

Python distingue les majuscules (capitales) des minuscules. Voici cinq variables différentes : `mavar`, `maVar`, `Mavar`, `MaVar`, `MAvar`.

Inverser des noms de fonctions dans les appels

Par exemple :

```
maChaine.strip().center(21, "*")
```

ne donne pas le même résultat que :

```
maChaine.center(21, "*").strip()
```

Se tromper sur la portée des variables locales

Quand on définit une variable localement dans une fonction ou une méthode, elle masque celle qui porte le même nom. Ce programme tente d'augmenter sa valeur avant de la définir (*ERR_01.py*)

```
>>> var = 10
>>> def truc():
...     var = var + 1        # Ce n'est pas la var du dessus !
...     print var
...
>>> truc()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 2, in foo
UnboundLocalError: local variable 'var' referenced before
```

```
assignment
```

Pour en savoir plus sur Python

Accéder à la documentation de Python n'est pas difficile : il suffit d'ouvrir le menu Aide (Help) de IDLE ou d'utiliser pour les fonctions la fonction `help()`. Le problème est qu'elle est en anglais. Quelques bénévoles avaient commencé à traduire l'aide d'une ancienne version, mais elle n'est plus à jour…

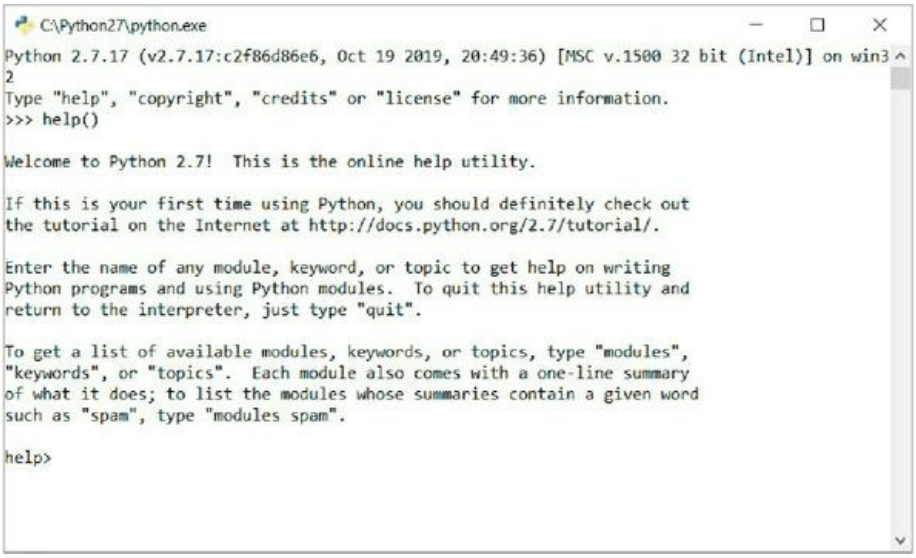


Figure A.2 : Accès à l'aide des fonctions par help() dans Python.

Même si les explications sont en anglais, quelqu'un qui ne lit pas cette langue pourra trouver des informations utiles, ne serait-ce que dans les nombreux exemples. Voici un écran de l'aide de Python, cette fois largement traduite en français.

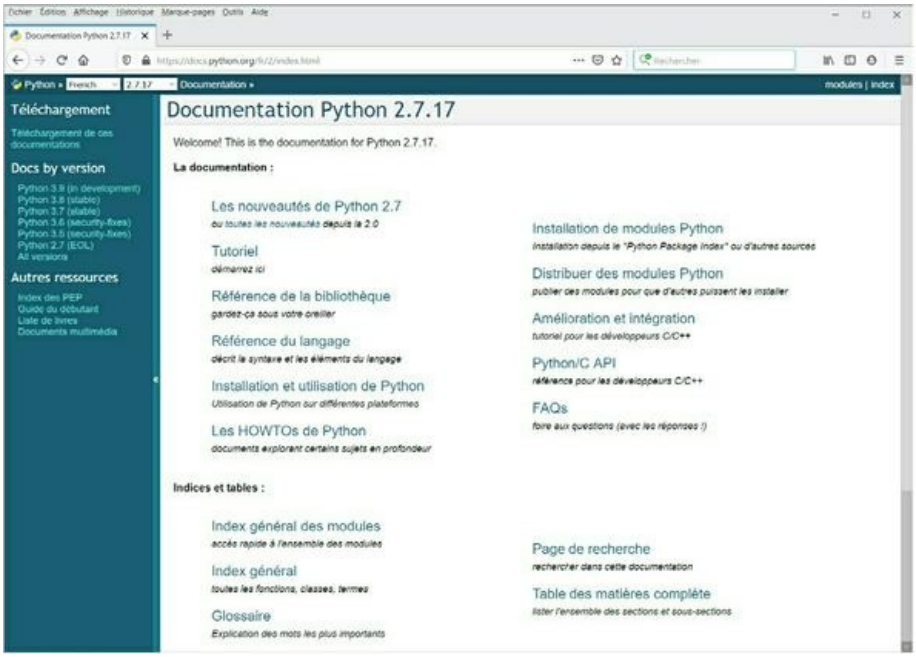


Figure A.3 : Une page du manuel en ligne de Python.

Sur le Web, les sources d'information sur Python en français sont plus rares que celles disponibles en anglais, mais tu peux toujours chercher en choisissant bien tes mots. Il y a aussi plusieurs forums d'utilisateurs Python dans lesquels tu pourras facilement trouver de l'aide.

Si tu as envie de consulter d'autres livres pour aller plus loin, je te conseille chez le même éditeur :

- *Python pour les Nuls ;*
- *Programmer pour les Nuls.*

Les lettres accentuées sous Python

Les Anglais et les Américains ne connaissent pas les soucis que nous avons en Europe (et ailleurs) pour utiliser notre langue maternelle dans les logiciels. Dans ce livre, j'ai volontairement évité les lettres accentuées parce que l'essentiel n'était pas là. Pourtant, on les veut, ces accents ! Voici mes conseils.

D'abord un rappel : pas de lettres accentuées dans les identifiants des programmes (variables, fonctions, méthodes, classes et objets). Un point, c'est tout. Les codes numériques des caractères doivent être de l'ASCII pur, valeurs numériques de 32 à 127.

Pour les messages à afficher et les textes à saisir, le mieux est de faire tes propres essais en partant des indices suivants.

» En première ligne des fichiers source, il faut ajouter ce pseudo-commentaire :

```
# -*- coding: utf-8 -*-
```

» Les fichiers doivent être sauvegardés dans le format UTF-8. Dans IDLE, il faut aller dans Options/Configurer IDLE, page Général, quatrième option Encodage source natif (Default Source Encoding) et choisir UTF-8.

» Si tes accents s'affichent bien sous Windows, cela ne sera pas nécessairement le cas sous Mac OS ou Linux. Fais des essais !

» Dans Python 2, tu peux forcer une chaîne à être codée en Unicode en ajoutant la lettre `u` en préfixe avant le guillemet ou l'apostrophe ouvrant :

```
u"Ma chaîne pour l'été du limaçon"
```

» La gestion des chaînes accentuées a changé entre Python 2 et Python 3. Comme tu vas un jour ou l'autre basculer vers Python 3, cela clôt la discusssion, car Python 3 travaille dès le départ avec des chaînes mondiales Unicode. Informes-toi sur Unicode et sur sa convention de stockage à longueur variable UTF-8.

Une interface graphique avec Tkinter

Tous les projets de ce livre interagissent avec l'utilisateur en mode texte dans la fenêtre IDLE de l'interpréteur Python. Ce n'est pas comme cela que se présentent les logiciels que tu utilises au quotidien sur ton ordinateur, ta tablette ou ton smartphone. As-tu vraiment appris à faire des programmes modernes dans ce livre ? Oui !

Quand les ordinateurs sont devenus assez puissants, on a décidé d'ajouter une couche de logiciel par dessus le système pour afficher des fenêtres en mode graphique, avec des boutons et des menus, le tout pilotable avec une souris. La nouveauté principale n'est pas le côté graphique, mais le fait qu'il peut y avoir plusieurs fenêtres ouvertes en même temps. Chaque fenêtre est la partie visible d'un programme.

Pour savoir dans quelle fenêtre afficher ce qui est par exemple tapé au clavier, il faut un arbitre qui est un programme du système. C'est cet arbitre qui garde le contrôle du flux d'exécution, ce n'est plus ton programme. Toi, tu écris tes fonctions et méthodes comme tu l'as appris dans ce livre, puis tu définis des connexions entre des événements (une action faite par l'utilisateur avec sa souris ou son clavier) et tes fonctions.

Tout ce qui apparaît dans ta fenêtre est prédéfini. Cela représente des milliers de lignes de code pour dessiner et gérer les composants de l'interface : boutons action, boutons radio, menus, listes déroulantes, boîtes de dialogue, etc.

Il existe plusieurs librairies d'interface utilisateur. La plus connue pour Python se nomme `Tkinter` (*Tk interface*). Voici un programme de dessin minimaliste écrit avec Tkinter suivi du code source (`Tk_Mozahik.py`). Il ne contient que vingt lignes de code source.

Figure A.4 : Le programme Mozahik pour faire de la mosaïque.

```
""" Tk_Mozahik par OE. Projet bonus du livre
    Programmer avec Python en s'amusant. """
from Tkinter import *
import random

FEN_LARGE = 500
FEN_HAUTE = 300
PALETTE = ["red", "blue", "yellow", "black", "violet",
"purple", "green"]
def mozahiker( evene ):
x1, y1 = ( evene.x - 3 ), ( evene.y - 3 )
x2, y2 = ( evene.x + 3 ), ( evene.y + 3 )
ndx = random.randint(0,6)
w.create_rectangle(x1, y1, x2, y2, fill = PALETTE[ndx])

maFen = Tk()
maFen.title("Mozahik : le dessin minimaliste avec Tkinter")
message = Label(maFen, text = "Pose de la mosaïque avec ta
souris !")
message.pack(side = TOP)

w = Canvas(maFen, width = FEN_LARGE, height = FEN_HAUTE)
w.pack(expand = YES, fill = BOTH)
w.bind( « <B1-Motion> », mozahiker )    # LIAISON EVENEMENT

btnQuitter = Button(maFen, text="Quitter Mozahik",
command=maFen.destroy)
btnQuitter.pack()

mainloop()
```

La ligne la plus importante est la dernière : ton programme y entre dans une boucle dont il ne sort plus de lui-même. C'est le gestionnaire de fenêtres (l'arbitre) qui va décider quand déclencher l'une des deux actions possibles :

» Si l'utilisateur clique le bouton Quitter Mozahik, le système déclenche la fonction indiquée dans la ligne de définition de ce bouton, donc la fonction de Tkinter nommée `destroy()` et appliquée à l'objet `maFen` qui est ta fenêtre (ce qui la referme).

» Quand l'utilisateur enfonce le bouton gauche et déplace le pointeur dans la surface de dessin (l'objet de type `Canvas` nommé `w`), le système détecte le déplacement et émet un événement `B1-Motion`. Comme tu as relié cet événement par `bind()` à `mozahiker()`, cela déclenche l'exécution de ta fonction.

Toutes les autres instructions ne sont que des préparatifs et ne sont exécutées qu'une fois au début. Quand ta fonction `mozahiker()` se termine, l'exécution repart dans la boucle contrôlée par le système, jusqu'au prochain événement qui concerne ton programme.

La librairie Tkinter correspond à plus de 3 000 lignes de code source (tu peux les consulter car c'est un logiciel libre). Et Tkinter n'est qu'une passerelle vers la librairie Tk qui corespond à plusieurs milliers de lignes encore.

Le plus important à retenir est que de nos jours la programmation est une activité en réseau :

» Tu utilises des fonctions écrites par d'autres et réunies dans des librairies.

» Ton programme cohabite avec d'autres sur la même surface d'affichage et réagit à des événements.

Plusieurs librairies d'interface sont disponibles. Certaines sont multi-plates-formes, c'est-à-dire que le code source reste le même pour Windows, Linux et Mac OS. C'est le cas de Tkinter mais aussi de PyQT. Une librairie est spécialement prévue pour créer des jeux : `pygame`. Le site `http://pygame.org` contient des exemples à essayer. Il y a aussi un site en français (`https://fr.wikibooks.org/wiki/Pygame`).

Bonne chance dans la suite de tes aventures pythonesques !

Sommaire