Vincent Le Goff

APPRENEZ À PROGRAMMER EN

PYTHON

4e édition

Éditions EYROLLES

APPRENEZ À PROGRAMMER EN

PYTHON

4e édition

Vous n'y connaissez rien en programmation et vous souhaitez apprendre un langage clair et intuitif ? Python est fait pour vous ! Vous découvrirez dans ce livre, conçu pour les débutants, tout ce dont vous avez besoin pour programmer, des bases à la bibliothèque standard, en passant par la programmation orientée objet et l'acquisition d'outils avancés ou professionnels pour devenir plus efficace.

La 4e édition de cet ouvrage est enrichie et mise à jour pour tirer parti des nouveautés de Python 3.1x.

QU'ALLEZ-VOUS APPRENDRE ?

- **Qu'est-ce que la programmation ? Quel langage choisir ? Pourquoi Python ?**
- **Installation de Python et découverte du langage**
- **Les concepts de la programmation orientée objet**
- **Initiation aux interfaces graphiques avec Tkinter**
- **Communication en réseau dans les programmes Python**
- **Les bonnes pratiques pour améliorer vos codes**
- **Les réflexes du « bon programmeur » pour tirer parti de votre code et de celui des autres**
- **Les outils du programmeur professionnel (chasse aux erreurs, utilisation de bibliothèques…)**

À PROPOS DE L'AUTEUR

Passionné d'informatique, Vincent Le Goff découvre au lycée la programmation en Python, un langage qu'il affectionne tout particulièrement pour son aspect simple et puissant. Étudiant à IN'TECH, il se spécialise en Système et Réseaux. Sur son temps libre, il publie des cours sur OpenClassrooms et participe également à plusieurs projets open source. Une belle réussite quand on sait que Vincent est non-voyant et malentendant !

www.editions-eyrolles.com

APPRENEZ À PROGRAMMER EN

PYTHON

SUR LE MÊME THÈME

H. Bersini, K. Hasselmann. – **L'intelligence artificielle en pratique avec Python**.
N°0100456, 2021, 136 pages.

E. Matthes. – **Cartes mémo Python**.
N°67860, 2020, 101 cartes.

J-B. Civet, B. Hanuš. – **Algorithmique et programmation en Python**.
N°67769, 2019, 96 pages.

M. O'Hanlon, D. Whale. – **Apprendre à coder en Python avec Minecraft**.
N°67721, 2e édition, 2019, 304 pages.

J. Briggs. – **Python pour les kids**.
N°14088, 2015, 332 pages.

G. Swinnen. – **Apprendre à programmer avec Python 3**.
N°13434, 3e édition, 2012, 435 pages.

DANS LA MÊME COLLECTION

É. Lalitte. – **Apprenez le fonctionnement des réseaux TCP/IP**.
N°67776, 4e édition, 2019, 452 pages.

C. Herby. – **Apprenez à programmer en Java**.
N°67521, 3e édition, 2018, 788 pages.

M. Nebra. – **Concevez votre site web avec PHP et MySQL.**
N°67475, 3e édition, 2017, 392 pages.

M. Nebra. – **Réalisez votre site web avec HTML 5 et CSS 3**.
N°67476, 2e édition, 2017, 364 pages.

V. Thuillier. – **Programmez en orienté objet en PHP**.
N°14472, 2e édition, 2017, 474 pages.

J. Pardanaud, S. De la Marck. – **Découvrez le langage JavaScript**.
N°14399, 2017, 478 pages.

Vincent Le Goff

APPRENEZ À PROGRAMMER EN

PYTHON

4e édition

Éditions EYROLLES

Éditions Eyrolles
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Mentions légales :
Conception couverture : Sophie Bai
Illustrations chapitres : Fan Jiyong et Sophie Bai

© Éditions Eyrolles, 2022, ISBN : 978-2-416-00655-5

Avant-propos

J'ai commencé à m'intéresser à l'informatique, et plus particulièrement au monde de la programmation, au début du lycée, il y a maintenant près de quinze ans. J'ai abordé ce terrain inconnu avec une grande curiosité... qui n'a pas encore faibli puisque je suis aujourd'hui étudiant à IN'TECH, une école supérieure d'ingénierie informatique ! Au premier abord, la programmation me semblait un monde aride et froid, rempli d'équations compliquées et de notions abstraites.

Heureusement, le premier langage à avoir attiré mon attention s'est trouvé être Python : à la fois simple et extrêmement puissant, je le considère aujourd'hui comme le meilleur choix quand on souhaite apprendre à programmer. Il est d'ailleurs resté le langage que j'utilise le plus dans les projets libres auxquels je contribue.

Cependant, Python n'est pas seulement simple : c'est un langage puissant. Il existe une différence entre connaître un langage et coder efficacement avec. Plusieurs années de pratique m'ont été nécessaires pour comprendre pleinement cette différence.

Les cours sur le langage Python s'adressant aux débutants ne sont pas rares sur le Web et beaucoup sont de grande qualité. Toutefois, il en existe trop peu, à mon sens, qui permettent de comprendre à la fois la syntaxe et la philosophie du langage.

Mon objectif ici est qu'après avoir lu ce livre, vous sachiez programmer en Python. Et par « programmer », je n'entends pas seulement maîtriser la syntaxe du langage, mais aussi comprendre sa philosophie.

Étant non-voyant et malentendant, je me suis efforcé de rendre ce cours aussi accessible que possible à tous. Ainsi, ne soyez pas surpris si vous y trouvez moins de schémas et d'illustrations que dans d'autres cours. J'ai fait en sorte que leur présence ne soit pas indispensable à la compréhension du lecteur.

Pour ceux qui se demandent comment je travaille, j'ai un ordinateur absolument semblable au vôtre. Pour l'utiliser, j'installe sur mon système un logiciel qu'on appelle *lecteur d'écran*. Ce dernier me dicte une bonne partie des informations affichées dans la fenêtre du logiciel que j'utilise, comme le navigateur Internet. Le lecteur, comme

son nom l'indique, lit grâce à une voix synthétique les informations qu'il détecte sur la fenêtre et peut également les transmettre à une *plage tactile*. C'est un périphérique qui se charge d'afficher automatiquement en braille les informations que lui transmet le lecteur d'écran. Avec ces outils, je peux donc me servir d'un ordinateur, aller sur Internet et même programmer !

FIGURE 1 – La plage tactile et le casque transmettent les informations affichées à l'écran

Qu'allez-vous apprendre en lisant ce livre ?

Ce livre s'adresse au plus grand nombre :

- si le mot programmation ne vous évoque rien de précis, ce livre vous guidera pas à pas dans la découverte du monde du **programmeur** ;
- si vous connaissez déjà un autre langage de programmation, ce livre présente de façon aussi claire que possible la syntaxe de Python et des exemples d'utilisation concrète de ce langage ;
- si vous connaissez déjà Python, ce cours peut vous servir de support comparatif avec d'autres livres et cours existants ;
- si vous enseignez le langage Python, j'ai espoir que ce livre pourra être un support utile, autant pour vous que pour vos étudiants.

Ce livre est divisé en cinq parties. Les trois premières sont à lire dans l'ordre, sauf si vous avez déjà de solides bases en Python :

1. **Introduction à Python**. Vous y apprendrez d'abord, si vous l'ignorez, ce que signifie **programmer**, ce qu'est Python et la syntaxe de base du langage.
2. **La Programmation Orientée Objet côté utilisateur**. Après avoir vu les bases de Python, nous allons étudier la **façade objet** de ce langage. Dans cette partie, vous apprendrez à utiliser les **classes** que définit Python. Ne vous inquiétez pas, les concepts d'objet et de classe seront largement détaillés ici. Donc, si ces mots ne vous disent rien au premier abord, pas d'inquiétude !
3. **La Programmation Orientée Objet côté développeur**. Cette partie poursuit l'approche de la façade objet débutée dans la partie précédente. Cette fois, cependant, au lieu d'être utilisateur des classes déjà définies par Python, vous allez apprendre à en créer. Là encore, ne vous inquiétez pas : nous verrons tous ces concepts pas à pas.
4. **Les merveilles de la bibliothèque standard**. Cette partie étudie plus en détail certains modules déjà définis par Python. Vous y apprendrez notamment à manipuler les dates et heures, créer des interfaces graphiques, construire une architecture réseau... et bien plus !
5. **Annexes**. Enfin, cette partie regroupe les annexes et résumés du cours. Il s'agit de notions qui ne sont pas absolument nécessaires pour développer en Python mais que je vous encourage tout de même à lire attentivement.

Comment lire ce livre ?

Suivez l'ordre des chapitres

Lisez ce livre comme on lit un roman. Il a été conçu pour cela.

Contrairement à beaucoup de livres techniques où il est courant de lire en diagonale et de sauter certains chapitres, il est ici très fortement recommandé de suivre l'ordre du cours, à moins que vous ne soyez déjà un peu expérimentés.

Pratiquez en même temps

Pratiquez régulièrement. N'attendez pas d'avoir fini de lire ce livre pour allumer votre ordinateur et faire vos propres essais.

Utilisez les codes web !

Afin de tirer parti d'OpenClassrooms dont ce livre est issu, celui-ci vous propose ce qu'on appelle des « codes web ». Ce sont des codes à six chiffres à saisir sur une page d'OpenClassrooms pour être automatiquement redirigé vers un site web sans avoir à en recopier l'adresse.

Pour les utiliser, rendez-vous sur la page suivante :

`http://fr.openclassrooms.com/codeweb.html`

Un formulaire vous invite à rentrer votre code. Faites un premier essai avec le suivant :

▷ Tester le code web
Code web : 123456

Ces codes ont deux intérêts :

- ils vous redirigent vers les sites web présentés tout au long du cours, vous facilitant l'obtention des logiciels dans leur toute dernière version ;
- ils vous permettent de télécharger les codes sources inclus dans ce livre, ce qui vous évitera d'avoir à recopier certains programmes un peu longs.

Ce système de redirection nous permet de tenir à jour le livre que vous avez entre les mains sans que vous ayez besoin d'acheter systématiquement chaque nouvelle édition. Si un site web change d'adresse, nous modifierons la redirection mais le code web à utiliser restera le même. Si un site web disparaît, nous vous redirigerons vers une page d'OpenClassrooms expliquant ce qui s'est passé et vous proposant une alternative.

En clair, c'est un moyen de nous assurer de la pérennité de cet ouvrage sans que vous ayez à faire quoi que ce soit !

Remerciements

De nombreuses personnes ont, plus ou moins directement, participé à ce livre. Mes remerciements leurs sont adressés :

- à ma famille avant tout, qui a su m'encourager, dans ce projet comme dans tout autre, du début jusqu'à la fin ;
- aux personnes, trop nombreuses pour que j'en dresse ici la liste, qui ont contribué, par leurs encouragements, leurs remarques et parfois leurs critiques, à faire de ce livre ce qu'il est ;
- à l'équipe d'OpenClassrooms qui a rendu ce projet envisageable et a travaillé d'arrache-pied pour qu'il se concrétise ;
- aux membres d'OpenClassrooms (Site du Zéro à l'époque) qui ont contribué à sa correction ou son enrichissement.

Table des matières

Première partie

Introduction à Python

Chapitre 1

Qu'est-ce que Python ?

Difficulté :

Vous avez décidé d'apprendre Python et je ne peux que vous en féliciter. J'essaierai d'anticiper vos questions et de ne laisser personne en arrière.

Dans ce chapitre, je vais d'abord vous expliquer ce qu'est un langage de programmation. Nous verrons ensuite brièvement l'histoire de Python, afin que vous sachiez au moins d'où vient ce langage ! Ce chapitre est théorique mais je vous conseille vivement de le lire quand même.

La dernière section portera sur l'installation de Python, une étape essentielle pour continuer ce cours. Que vous travailliez avec Windows, Linux ou macOS, vous y trouverez des explications précises sur l'installation.

Allez, on attaque !

Un langage de programmation ? Qu'est-ce que c'est ?

La communication humaine

Non, ceci n'est pas une explication biologique ou philosophique, ne partez pas ! Très simplement, si vous arrivez à comprendre ces suites de symboles étranges et déconcertants que sont les lettres de l'alphabet, c'est parce que nous respectons certaines conventions, dans le langage et dans l'écriture. En français, il y a des règles de grammaire et d'orthographe, je ne vous apprends rien. Vous communiquez en connaissant plus ou moins consciemment ces règles et en les appliquant plus ou moins bien, selon les cas. Cependant, ces règles peuvent être aisément contournées : personne ne peut prétendre connaître l'ensemble des règles de la grammaire et de l'orthographe françaises, et peu de gens s'en soucient. Après tout, même si vous faites des fautes, les personnes avec qui vous communiquez pourront facilement vous comprendre. Quand on communique avec un ordinateur, cependant, c'est très différent.

Mon ordinateur communique aussi !

Eh oui, votre ordinateur communique sans cesse avec vous et vous communiquez sans cesse avec lui. D'accord, il vous dit très rarement qu'il a faim, que l'été s'annonce caniculaire et que le dernier disque de ce groupe très connu était à pleurer. Il n'y a rien de magique si, quand vous cliquez sur la petite croix en haut à droite de l'application en cours, celle-ci comprend qu'elle doit se fermer.

Le langage machine

En fait, votre ordinateur se fonde aussi sur un langage pour communiquer avec vous ou avec lui-même. Les opérations qu'un ordinateur peut effectuer à la base sont des plus classiques : l'addition de deux nombres, leur soustraction, leur multiplication, leur division, entière ou non. Et pourtant, ces cinq opérations suffisent amplement à faire fonctionner les logiciels de simulation les plus complexes ou les jeux super-réalistes. Tous ces logiciels fonctionnent en gros de la même façon :

— une suite d'instructions écrites en langage machine compose le programme ;
— lors de l'exécution du programme, ces instructions décrivent à l'ordinateur ce qu'il faut faire (l'ordinateur ne peut pas le deviner).

Une liste d'instructions ? Qu'est-ce que c'est encore que cela ?

En schématisant volontairement, une instruction pourrait demander au programme de se fermer si vous cliquez sur la croix en haut à droite de votre écran, ou de rester en tâche de fond si tel est son bon plaisir. Toutefois, en langage machine, une telle action demande à elle seule un nombre assez important d'instructions. Mais bon, vous pouvez

vous en douter, parler avec l'ordinateur en langage machine, qui ne comprend que le binaire, ce n'est ni très enrichissant, ni très pratique et en tout cas pas très drôle. On a donc inventé des langages de programmation pour faciliter la communication avec l'ordinateur.

Le langage binaire est uniquement constitué de 0 et de 1. « 01000010011011110110111001101010011011110111010101110010 », par exemple, signifie « Bonjour ». Bref, autant vous dire que discuter en binaire avec un ordinateur peut être long (surtout pour vous).

Les langages de programmation

Les langages de programmation sont bien plus faciles à comprendre pour nous, pauvres êtres humains que nous sommes. Le mécanisme reste le même, mais le langage est bien plus compréhensible. Au lieu d'écrire les instructions dans une suite assez peu intelligible de 0 et de 1, les ordres donnés à l'ordinateur sont écrits dans un « langage », souvent en anglais, avec une syntaxe particulière qu'il est nécessaire de respecter. Toutefois, avant que l'ordinateur puisse comprendre ces instructions, celles-ci doivent être traduites en langage machine (figure 1.1).

FIGURE 1.1 – Traduction d'un programme en langage binaire

En gros, le programmeur « n'a qu'à » écrire des **lignes de code** dans le langage qu'il a choisi, les étapes suivantes sont automatisées pour permettre à l'ordinateur de les décoder.

Il existe un grand nombre de langages de programmation et Python en fait partie. Il n'est pas nécessaire pour le moment de donner plus d'explications sur ces mécanismes très schématisés. Si vous n'avez pas réussi à comprendre les mots de vocabulaire et l'ensemble de ces explications, cela ne vous pénalisera pas pour la suite. Néanmoins, je trouve intéressant de donner ces précisions quant aux façons de communiquer avec son ordinateur.

Pour la petite histoire

La première version de Python est sortie en 1991. Créé par **Guido van Rossum**, il a voyagé du Macintosh de son créateur, qui travaillait à cette époque au *Centrum voor*

Wiskunde en Informatica aux Pays-Bas, jusqu'à se voir associer une organisation à but non lucratif particulièrement dévouée, la **Python Software Foundation**, créée en 2001. Ce langage a été baptisé ainsi en hommage à la troupe de comiques les « Monty Python ».

À quoi peut servir Python ?

Python est un langage puissant, à la fois facile à apprendre et riche en possibilités. Dès l'instant où vous l'installez sur votre ordinateur, vous disposez de nombreuses fonctionnalités intégrées que nous allons découvrir tout au long de ce livre.

Il est, en outre, très facile d'étendre les fonctionnalités existantes, comme nous allons le voir. Ainsi, il existe ce qu'on appelle des **bibliothèques** qui aident le développeur à travailler sur des projets particuliers. Plusieurs bibliothèques peuvent être installées pour, par exemple, développer des interfaces graphiques en Python.

Concrètement, voilà ce qu'on peut faire avec Python :

— de petits programmes très simples, appelés **scripts**, chargés d'une mission très précise sur votre ordinateur ;
— des programmes complets, comme des jeux, des suites bureautiques, des logiciels multimédias, des clients de messagerie...
— des projets très complexes, comme des progiciels (groupes de plusieurs logiciels fonctionnant ensemble, principalement utilisés dans le monde professionnel).

Voici quelques-unes des fonctionnalités offertes par Python et ses bibliothèques :

— créer des interfaces graphiques ;
— faire circuler des informations au travers d'un réseau ;
— dialoguer d'une façon avancée avec votre système d'exploitation ;
— créer et héberger un site web dynamique ;
— et bien d'autres...

Bien entendu, vous n'allez pas apprendre tout cela en quelques minutes. Ce cours vous donnera des bases suffisamment larges pour développer des projets qui deviendront, par la suite, assez importants.

Un langage de programmation interprété

Eh oui, vous allez devoir patienter encore un peu car il me reste deux ou trois choses à vous expliquer et il est important de connaître un minimum ces détails qui peuvent sembler peu pratiques de prime abord. Python est un langage de programmation **interprété**, c'est-à-dire que les instructions que vous lui envoyez sont « transcrites » en langage machine au fur et à mesure de leur lecture. D'autres langages (comme le C / C++) sont appelés « langages **compilés** » car, avant de pouvoir les exécuter, un logiciel spécialisé se charge de transformer le code du programme en langage machine. On appelle cette étape la « **compilation** ». À chaque modification du code, il faut rappeler une étape de compilation.

Les avantages d'un langage interprété sont la simplicité (on ne passe pas par une étape de compilation avant d'exécuter son programme) et la portabilité (un langage tel que Python est censé fonctionner aussi bien sous Windows que sous Linux ou macOS et on ne devrait avoir à effectuer aucun changement dans le code pour le passer d'un système à l'autre). Cela ne veut pas dire que les langages compilés ne sont pas portables, loin de là ! Toutefois, on doit souvent utiliser des compilateurs différents et, d'un système à l'autre, certaines instructions ne sont pas compatibles, voire se comportent différemment.

En contrepartie, un langage compilé se révélera bien plus rapide qu'un langage interprété (la traduction à la volée de votre programme ralentit l'exécution), bien que cette différence tende à se faire de moins en moins sentir au fil des améliorations. De plus, il faudra installer Python sur le système d'exploitation que vous utilisez pour que l'ordinateur comprenne votre code.

Différentes versions de Python

Lors de la création de la Python Software Foundation, en 2001, et durant les années qui ont suivi, le langage est passé par une suite de versions que l'on a englobées dans l'appellation Python 2.x (2.3, 2.5, 2.6...). Depuis le 13 février 2009, la version 3.0.1 est disponible. Elle casse la **compatibilité ascendante** qui prévalait lors des dernières versions.

Compatibilité quoi ?

Quand un langage de programmation est mis à jour, les développeurs se gardent bien de supprimer ou de trop modifier d'anciennes fonctionnalités. L'intérêt est qu'un programme qui fonctionne sous une certaine version marchera toujours avec la nouvelle version en date. Cependant, la Python Software Foundation, observant un bon nombre de fonctionnalités obsolètes, mises en œuvre plusieurs fois... a décidé de nettoyer tout le projet. Un programme qui tourne à la perfection sous Python 2.x devra donc être mis à jour un minimum pour fonctionner de nouveau sous Python 3. C'est pourquoi je vais vous conseiller ultérieurement de télécharger et d'installer la dernière version en date de Python. Je m'attarderai en effet sur les fonctionnalités de Python 3 et certaines d'entre elles ne seront pas accessibles (ou pas sous le même nom) dans les anciennes versions.

Notons que la branche 2.x de Python n'est plus mise à jour depuis 2020. Vous pourrez toujours apprendre et utiliser Python dans cette version, mais les bogues rencontrés ne seront plus corrigés. C'est une raison supplémentaire de passer à Python 3.

Ceci étant posé, tous à l'installation !

Installer Python

L'installation de Python est un jeu d'enfant, aussi bien sous Windows que sous les systèmes Unix. Quel que soit votre système d'exploitation, vous devez vous rendre sur le site officiel de Python.

▷ Site officiel de Python
Code web : 199501

Sous Windows

1. Dans la section `Download` (téléchargement), vous devriez voir la version la plus récente de Python (3.10.0 à l'heure où j'écris ces lignes). Cliquez sur le numéro de version.
2. On vous propose un (ou plusieurs) lien(s) : sélectionnez la version qui conviendra à votre processeur. Si vous avez un doute, téléchargez une version « x86 ».
3. Enregistrez puis exécutez le fichier d'installation et suivez les étapes. Ce n'est ni très long ni très difficile.
4. Une fois l'installation terminée, vous pouvez vous rendre dans le menu `Démarrer` > `Tous les programmes`. Python devrait apparaître dans cette liste (figure 1.2). Nous verrons bientôt comment le lancer, pas d'impatience...

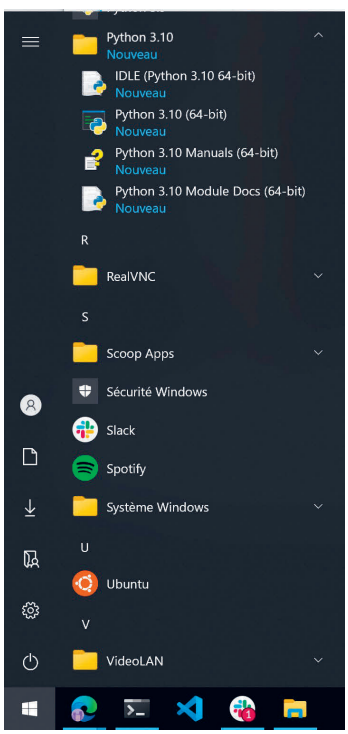


FIGURE 1.2 – Python est installé sous Windows

Sous Linux

Python est pré-installé sur la plupart des distributions Linux. Cependant, il est possible que vous n'ayez pas la dernière version en date. Pour le vérifier, tapez dans un terminal la commande `python -V`. Elle vous renvoie la version de Python actuellement installée sur votre système.

Cliquez sur `download` et téléchargez la dernière version de Python (actuellement « Python 3.10 gzipped source tarball (for Linux, Unix or OS X) »). Ouvrez un terminal, puis rendez-vous dans le dossier où se trouve l'archive :

1. Décompressez l'archive en tapant : `tar -xzf Python-3.10.0.tgz` (cette commande est bien entendu à adapter suivant la version et le type de compression).
2. Attendez quelques instants que la décompression se termine, puis rendez-vous dans le dossier qui vient d'être créé dans le répertoire courant (`Python-3.10.0` dans mon cas).
3. Exécutez le script `configure` en tapant `./configure` dans la console.
4. Une fois que la configuration s'est déroulée, il n'y a plus qu'à compiler en tapant `make` puis `make install` en tant que super-utilisateur.

Sous macOS

Téléchargez la dernière version de Python. Ouvrez le fichier `.dmg` et double-cliquez sur le paquet d'installation `Python.mpkg`. Un assistant d'installation s'ouvre, laissez-vous guider : Python est maintenant installé !

Lancer Python

Ouf ! Voilà qui est fait !

Bon, en théorie, on commence à utiliser Python dès le prochain chapitre mais, pour que vous soyez un peu récompensés de votre installation exemplaire, voici les différents moyens d'accéder à la ligne de commande Python que nous allons tout particulièrement étudier dans les prochains chapitres.

Sous Windows

Vous avez plusieurs façons d'accéder à la ligne de commande Python, la plus évidente consistant à passer par les menus `Démarrer` > `Tous les programmes` > `Python 3.10` > `Python (Command Line)`. Si tout se passe bien, vous devriez obtenir une magnifique console (figure 1.3). Il se peut que les informations affichées dans la vôtre ne soient pas les mêmes, mais ne vous en inquiétez pas.

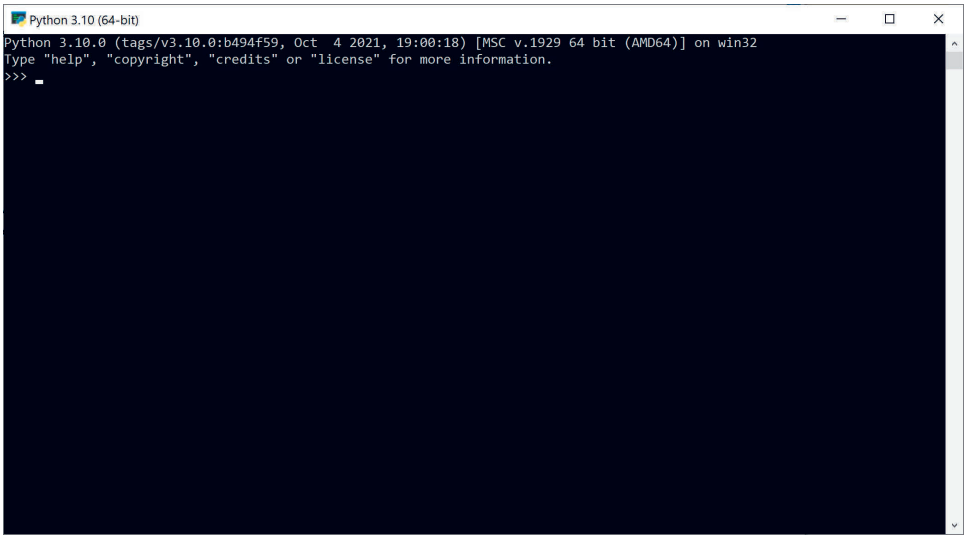


FIGURE 1.3 – La console Python sous Windows

Qu'est-ce que c'est que cela ?

On verra plus tard. L'important, c'est que vous ayez réussi à ouvrir la console d'interprétation de Python ; le reste attendra le prochain chapitre.

Vous pouvez également passer par la ligne de commande Windows ; à cause des raccourcis, je privilégie en général cette méthode, mais c'est une question de goût. Allez dans le menu `Démarrer`, puis cliquez sur `Exécuter`. Dans la fenêtre qui apparaît, tapez simplement `python` et la ligne de commande devrait s'afficher de nouveau. Sachez que vous pouvez directement vous rendre dans `Exécuter` en tapant le raccourci `Windows` + `R`.

Sous Windows 8 ou 10, vous pouvez tout simplement rechercher `python`. Cliquez sur le menu `Démarrer` et tapez `python`. Windows devrait vous proposer la version installée en premier choix.

Pour fermer l'interpréteur de commandes Python, tapez `exit()` puis appuyez sur la touche `Entrée`.

Sous Linux

Lorsque vous l'avez installé sur votre système, Python a créé un lien vers l'interpréteur sous la forme **python3.X** (le X étant le numéro de la version installée).

Si, par exemple, vous avez installé Python 3.10, vous pouvez y accéder grâce à la commande :

```
$ python3.10
Python 3.10.0 (default, Dec  1 2021, 13:28:04) [GCC 9.3.0] on
 ↪  linux
Type "help", "copyright", "credits" or "license" for more
 ↪  information.
>>>
```

Pour fermer la ligne de commande Python, n'utilisez pas CTRL + C mais CTRL + D (nous verrons plus tard pourquoi).

Sous macOS

Cherchez un dossier `Python` dans le dossier `Applications`. Pour lancer le programme, ouvrez l'application `IDLE` de ce dossier. Vous êtes prêts à passer au concret !

En résumé

— Python est un langage de programmation interprété, à ne pas confondre avec un langage compilé.
— Il permet de créer toutes sortes de programmes, comme des jeux, des logiciels, des progiciels, etc.
— Il est possible d'associer des **bibliothèques** à Python afin d'étendre ses possibilités.
— Il est portable, c'est-à-dire qu'il peut fonctionner sous différents systèmes d'exploitation (Windows, Linux, macOS).

Chapitre 2

Premiers pas avec l'interpréteur de commandes Python

Difficulté :

Après les premières notions théoriques et l'installation de Python, il est temps de découvrir un peu l'interpréteur de commandes de ce langage. Même si ces petits tests vous semblent anodins, vous découvrirez dans ce chapitre les premiers rudiments de la syntaxe du langage et je vous conseille fortement de me suivre pas à pas, surtout si vous êtes face à votre premier langage de programmation.

Comme tout langage de programmation, Python a une syntaxe claire : on ne peut pas lui envoyer n'importe quelle information dans n'importe quel ordre. Nous allons voir ici ce que Python mange... et ce qu'il ne mange pas.

Où est-ce qu'on est, là ?

Pour commencer, je vais vous demander de retourner dans l'interpréteur de commandes Python (je vous ai montré, à la fin du chapitre précédent, comment y accéder en fonction de votre système d'exploitation).

Je vous rappelle les informations qui figurent dans cette fenêtre, même si elles peuvent être différentes chez vous en fonction de votre version et de votre système d'exploitation.

```
1 Python 3.10.0 (tags/v3.10.0:b494f59, Oct  4 2021, 18:46:30)
   ↪  [MSC v.1929 32 bit (Intel)] on win32
2 Type "help", "copyright", "credits" or "license" for more
   ↪  information.
3
4 >>>
```

À sa façon, Python vous souhaite la bienvenue dans son interpréteur de commandes.

Attends, attends. C'est quoi cet interpréteur ?

Souvenez-vous, au chapitre précédent, je vous ai donné une brève explication sur la différence entre langages compilés et langages interprétés. Eh bien, cet interpréteur de commandes va nous permettre de tester directement du code. Je saisis une ligne d'instructions, j'appuie sur la touche Entrée de mon clavier, je regarde ce que me répond Python (s'il me dit quelque chose), puis j'en saisis une deuxième, une troisième... Cet interpréteur est particulièrement utile pour comprendre les bases de Python et réaliser nos premiers petits programmes. Le principal inconvénient, c'est que le code que vous saisissez n'est pas sauvegardé (sauf si vous l'enregistrez manuellement, mais chaque chose en son temps).

Dans la fenêtre que vous avez sous les yeux, l'information qui ne change pas d'un système d'exploitation à l'autre est la série de trois chevrons qui se trouve en bas à gauche des informations : `>>>`. Ces trois signes signifient : « je suis prêt à recevoir tes instructions ».

Comme je l'ai dit, les langages de programmation respectent une syntaxe claire. Vous ne pouvez pas espérer que l'ordinateur comprenne si, dans cette fenêtre, vous commencez par lui demander : « j'aimerais que tu me codes un jeu vidéo génial ». Et autant que vous le sachiez tout de suite (bien qu'à mon avis, vous vous en doutiez), on est très loin d'obtenir des résultats aussi spectaculaires à notre niveau.

Tout cela pour dire que, si vous saisissez n'importe quoi dans cette fenêtre, la probabilité est grande que Python vous indique, clairement et fermement, qu'il n'a rien compris.

Si, par exemple, vous saisissez « premier test avec Python », vous obtenez le résultat suivant :

```
>>> premier test avec Python
  File "<stdin>", line 1
    premier test avec Python
                  ^^^^^^^^^^
SyntaxError: invalid syntax. Perhaps you forgot a comma?
>>>
```

Eh oui, l'interpréteur parle en anglais et les instructions que vous saisirez, comme pour l'écrasante majorité des langages de programmation, seront également en anglais. Mais pour l'instant, rien de bien compliqué : l'interpréteur vous indique qu'il a trouvé un problème dans votre ligne d'instruction. Il vous indique le numéro de la ligne (en l'occurence la première), qu'il vous répète obligeamment (ceci est très utile quand on travaille sur un programme de plusieurs centaines de lignes). Puis il vous dit ce qui l'arrête, ici : `SyntaxError: invalid syntax`. Limpide n'est-ce pas ? Ce que vous avez saisi est incompréhensible pour Python. Enfin, la preuve qu'il n'est pas rancunier, c'est qu'il vous affiche à nouveau une série de trois chevrons, montrant bien qu'il est prêt à retenter l'aventure.

Bon, c'est bien joli de recevoir un message d'erreur au premier test mais je me doute que vous aimeriez bien voir quelque chose qui fonctionne maintenant. C'est parti donc.

Vos premières instructions : un peu de calcul mental pour l'ordinateur

C'est assez trivial, quand on y pense, mais je trouve qu'il s'agit d'une excellente manière d'aborder pas à pas la syntaxe de Python. Nous allons donc essayer d'obtenir les résultats de calculs plus ou moins compliqués. Je vous rappelle encore une fois qu'exécuter les tests en même temps que moi sur votre machine est une très bonne façon de vous rendre compte de la syntaxe et surtout, de la retenir.

Saisir un nombre

Vous avez pu voir sur notre premier (et à ce jour notre dernier) test que Python n'aimait pas particulièrement les suites de lettres qu'il ne comprend pas. En revanche, l'interpréteur adore les nombres. D'ailleurs, il les accepte sans sourciller, sans une seule erreur :

```
>>> 7
7
>>>
```

D'accord, ce n'est pas extraordinaire. On saisit un nombre et l'interpréteur le renvoie. Pourtant, dans bien des cas, ce simple retour indique que l'interpréteur a bien compris et que votre saisie est en accord avec sa syntaxe. De même, vous pouvez saisir des nombres à virgule.

```
>>> 9.5
9.5
>>>
```

Attention : on utilise ici la notation anglo-saxonne, c'est-à-dire que le point remplace la virgule. La virgule a un tout autre sens pour Python ; prenez donc cette habitude dès maintenant.

Il va de soi que l'on peut tout aussi bien saisir des nombres négatifs (faites d'ailleurs l'essai).

Opérations courantes

Bon, il est temps d'apprendre à utiliser les principaux opérateurs de Python, qui vont vous servir pour la grande majorité de vos programmes.

Addition, soustraction, multiplication, division

Pour effectuer ces opérations, on utilise respectivement les symboles +, -, * et /.

```
>>> 3 + 4
7
>>> -2 + 93
91
>>> 9.5 + 2
11.5
>>> 3.11 + 2.08
5.1899999999999995
>>>
```

Pourquoi ce dernier résultat approximatif ?

Python n'y est pas pour grand-chose. En fait, le problème vient en grande partie de la façon dont les nombres à virgule sont écrits dans la mémoire de votre ordinateur. C'est pourquoi, en programmation, on préfère travailler autant que possible avec des nombres entiers. Cependant, vous remarquerez que l'erreur est infime et qu'elle n'aura pas de réel impact sur les calculs. Les applications qui ont besoin d'une précision mathématique à toute épreuve essayent de pallier ces défauts par d'autres moyens mais ici, ce ne sera pas nécessaire.

Faites également des tests pour la soustraction, la multiplication et la division : il n'y a rien de difficile.

Division entière et modulo

Si vous avez pris le temps de tester la division, vous vous êtes rendu compte que le résultat est donné avec une virgule flottante.

```
>>> 10 / 5
2.0
>>> 10 / 3
3.3333333333333335
>>>
```

Il existe deux autres opérateurs qui permettent de connaître le résultat d'une division entière et le reste de cette division.

Le premier opérateur utilise le symbole « // ». Il renvoie la partie entière d'une division.

```
>>> 10 // 3
3
>>>
```

L'opérateur « % », que l'on appelle le « modulo », retourne le reste de la division.

```
>>> 10 % 3
1
>>>
```

Ces notions de *partie entière* et de *reste de division* ne sont pas bien difficiles à comprendre et vous serviront très probablement par la suite.

Si vous avez du mal à en saisir le sens, sachez donc que :

— La partie entière de la division de 10 par 3 est le résultat de cette division, sans tenir compte des chiffres au-delà de la virgule (en l'occurence, 3).
— Pour obtenir le modulo d'une division, on « récupère » son reste. Dans notre exemple, $10/3 = 3$ et il reste 1. Une fois que l'on a compris cela, ce n'est pas bien compliqué.

Souvenez-vous bien de ces deux opérateurs, et surtout du modulo « % », dont vous aurez besoin dans vos programmes futurs.

En résumé

— L'interpréteur de commandes Python permet de tester du code au fur et à mesure qu'on l'écrit.
— L'interpréteur Python accepte des nombres et est capable d'effectuer des calculs.
— Un nombre décimal s'écrit avec un point et non une virgule.
— Les calculs impliquant des nombres décimaux donnent parfois des résultats approximatifs. C'est pourquoi on préfèrera, dans la mesure du possible, travailler avec des nombres entiers.

Chapitre 3

Le monde merveilleux des variables

Difficulté :

Au chapitre précédent, vous avez saisi vos premières instructions en langage Python, bien que vous ne vous en soyez peut-être pas rendu compte. Il est également vrai que les instructions saisies auraient fonctionné dans la plupart des langages. Ici, cependant, nous allons commencer à approfondir un petit peu la syntaxe du langage, tout en découvrant un concept important de la programmation : les variables.

Ce concept est essentiel et vous ne pouvez absolument pas faire l'impasse dessus. Cependant, je vous rassure, il n'y a rien de compliqué ; que de l'utile et de l'agréable.

Qu'est-ce qu'une variable ? Et à quoi cela sert-il ?

Les variables sont l'un des concepts qui se retrouvent dans tous les langages de programmation. Autant dire que sans variable, on ne peut pas programmer ; et ce n'est pas une exagération.

Qu'est-ce qu'une variable ?

Une variable est une donnée de votre programme, stockée dans votre ordinateur. C'est un code alpha-numérique que vous allez lier à une donnée de votre programme, afin de pouvoir l'utiliser à plusieurs reprises et faire des calculs un peu plus intéressants avec. C'est bien joli de savoir faire des opérations mais, si on ne peut pas stocker le résultat quelque part, cela devient très vite ennuyeux.

Voyez la mémoire de votre ordinateur comme une grosse armoire avec plein de tiroirs. Chaque tiroir peut contenir une donnée ; certaines de ces données seront des variables de votre programme.

Comment cela fonctionne-t-il ?

Le plus simplement du monde. Vous allez dire à Python : « je veux que, dans une variable que je nomme `âge`, tu stockes mon âge, pour que je puisse le retenir (si j'ai la mémoire très courte), l'augmenter (à mon anniversaire) et l'afficher si besoin est ».

Comme je vous l'ai dit, on ne peut pas passer à côté des variables. Vous ne voyez peut-être pas encore tout l'intérêt de stocker des informations de votre programme et pourtant, si vous ne stockez rien, vous ne pouvez pratiquement rien faire.

En Python, pour donner une valeur à une variable, il suffit d'écrire
`nom_de_la_variable = valeur`.

Une variable doit respecter quelques règles de syntaxe incontournables :

1. Le nom de la variable ne peut être composé que de lettres, majuscules ou minuscules, de chiffres et du symbole souligné « _ » (appelé *underscore* en anglais).

2. Le nom de la variable ne peut pas commencer par un chiffre.

3. Le langage Python est sensible à la casse, ce qui signifie que des lettres majuscules et minuscules ne constituent pas la même variable (la variable `AGE` est différente de `aGe`, elle-même différente de `age`).

4. Le nom de la variable peut contenir des accents. Notez cependant que Python fera alors la différence entre `age` et `âge`.

L'utilisation des accents dans des variables est encore relativement récente. À l'heure actuelle, peu de langages de programmation l'autorisent et beaucoup de programmeurs, que ce soit en Python ou ailleurs, recommandent d'éviter les accents dans les noms de variables. Ce n'est pas mon cas. Python le permet, le français comporte des accents, autant les utiliser. Rien ne vous empêche de retirer les accents dans les exemples que je vous donne, bien entendu.

Au-delà de ces règles de syntaxe incontournables, il existe des conventions définies par les programmeurs eux-mêmes. L'une d'elles, que j'utilise assez souvent, consiste à écrire la variable en minuscules et à remplacer les espaces éventuels par le caractère souligné « _ ». Si je dois créer une variable contenant mon âge, elle se nommera donc `mon_âge`. Une autre convention utilisée consiste à passer en majuscule l'initiale de chaque mot constituant le nom de la variable, sauf le premier ; cela donnerait par exemple `monÂge`.

Vous pouvez appliquer la convention qui vous plaît, ou même en créer une bien à vous, mais essayez de rester cohérent et de n'en utiliser qu'une seule. En effet, il est essentiel de pouvoir vous repérer dans vos variables dès que vous commencez à travailler sur des programmes volumineux.

Ainsi, si je veux associer mon âge à une variable, la syntaxe sera :

```
1 mon_âge = 21
```

L'interpréteur vous affiche aussitôt trois chevrons sans aucun message. Cela signifie qu'il a bien compris et qu'il n'y a eu aucune erreur.

Sachez qu'on appelle cette étape *l'affectation de valeur à une variable* (parfois raccourci en « affectation de variable »). On dit en effet qu'on a affecté la valeur 21 à la variable `mon_âge`.

On peut afficher la valeur de cette variable en la saisissant simplement dans l'interpréteur de commandes.

```
1 >>> mon_âge
2 21
3 >>>
```

Les espaces séparant « = » du nom et de la valeur de la variable sont facultatifs. Je les mets pour des raisons de lisibilité.

Bon, c'est bien joli tout cela, mais qu'est-ce qu'on fait avec cette variable ?

Eh bien, tout ce que vous avez déjà fait au chapitre précédent, mais cette fois en utilisant la variable comme un nombre à part entière. Vous pouvez même affecter à d'autres variables des valeurs obtenues en effectuant des calculs sur la première et c'est là toute la puissance de ce mécanisme.

Essayons par exemple d'augmenter de 2 la variable `mon_âge`.

```
>>> mon_âge = mon_âge + 2
>>> mon_âge
23
>>>
```

Encore une fois, lors de l'affectation de la valeur, rien ne s'affiche, ce qui est parfaitement normal.

Maintenant, essayons d'affecter une valeur à une autre variable d'après la valeur de `mon_âge`.

```
>>> mon_âge_x2 = mon_âge * 2
>>> mon_âge_x2
46
>>>
```

Encore une fois, je vous invite à tester en long, en large et en travers cette possibilité. Le concept n'est pas compliqué mais extrêmement puissant. De plus, comparé à certains langages, affecter une valeur à une variable est extrêmement simple. Si la variable n'est pas créée, Python s'en charge automatiquement. Si la variable existe déjà, l'ancienne valeur est supprimée et remplacée par la nouvelle. Quoi de plus simple ?

Certains mots-clés de Python sont **réservés**, c'est-à-dire que vous ne pouvez pas vous en servir pour nommer des variables.

En voici la liste pour Python 3 :

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

Notez que cette liste concerne Python 3.10. De nouveaux mots-clés apparaissent de temps en temps dans les nouvelles versions de Python. Notez également que `match` et `case` que nous verrons plus tard ne sont pas considérés comme des mots-clés et n'apparaissent donc pas dans ce tableau.

Ces mots-clés sont utilisés par Python, vous ne pouvez pas construire de variables portant ces noms. Vous allez découvrir dans la suite de ce cours la majorité de ces mots-clés et comment ils s'utilisent.

Les types de données en Python

Là se trouve un concept très important, que l'on retrouve dans beaucoup de langages de programmation. Ouvrez grand vos oreilles, ou plutôt vos yeux, car vous devrez être parfaitement à l'aise avec ce concept pour continuer la lecture de ce livre. Rassurez-vous toutefois, du moment que vous êtes attentifs, il n'y a rien de compliqué à comprendre.

Qu'entend-on par « type de donnée » ?

Jusqu'ici, vous n'avez travaillé qu'avec des nombres. Et, s'il faut bien avouer qu'on ne fera que très rarement un programme sans aucun nombre, c'est loin d'être la seule donnée que l'on peut utiliser en Python. À terme, vous serez même capables de créer vos propres types de données, mais n'anticipons pas.

Python a besoin de connaître quels types de données sont utilisés pour savoir quelles opérations il peut effectuer avec. Dans ce chapitre, vous allez apprendre à travailler avec des chaînes de caractères et multiplier une chaîne de caractères ne se fait pas du tout comme la multiplication d'un nombre. Pour certains types de données, la multiplication n'a d'ailleurs aucun sens. Python associe donc à chaque donnée un type, qui va définir les opérations autorisées sur cette donnée en particulier.

Les différents types de données

Nous n'allons voir ici que les incontournables et les plus faciles à manier. Des chapitres entiers seront consacrés aux types plus complexes.

Les nombres entiers

Eh oui, Python différencie les entiers des nombres à virgule flottante !

Pourquoi cela ?

Initialement, c'est surtout pour une question de place en mémoire mais, pour un ordinateur, les opérations que l'on effectue sur des nombres à virgule ne sont pas les mêmes que celles sur les entiers et cette distinction reste encore d'actualité de nos jours.

Le type entier se nomme `int` en Python (qui correspond à l'anglais « integer », c'est-à-dire entier). La forme d'un entier est un nombre sans virgule.

```
1 3
```

Nous avons vu au chapitre précédent les opérations que l'on pouvait effectuer sur ce type de données et, même si vous ne vous en souvenez pas, les deviner est assez élémentaire.

Les nombres flottants

Les flottants sont les nombres à virgule. Ils se nomment `float` en Python (ce qui signifie « flottant » en anglais). N'oubliez pas de remplacer la virgule par un point. Si vous voulez qu'un nombre sans partie décimale soit considéré par le système comme un flottant, ajoutez-lui une partie flottante de 0 (exemple **52.0**).

```
3.152
```

Les nombres après la virgule ne sont pas infinis, puisque rien n'est infini en informatique. Néanmoins, la précision est assez importante pour travailler sur des données très fines.

Les chaînes de caractères

Heureusement, les types de données disponibles en Python ne sont pas limités aux seuls nombres, bien loin de là. Le dernier type « simple » que nous verrons dans ce chapitre est la chaîne de caractères. Ce type de donnée sert à stocker une série de lettres, voire une phrase.

On peut écrire une chaîne de caractères de différentes façons :

— entre guillemets (`"ceci est une chaîne de caractères"`) ;
— entre apostrophes (`'ceci est une chaîne de caractères'`) ;
— entre triples guillemets (`"""ceci est une chaîne de caractères"""`) ;
— plus rarement, entre triples apostrophes (`'''ceci est une chaîne de caractères'''`).

On peut, à l'instar des nombres (et de tous les types de données) stocker une chaîne de caractères dans une variable (`ma_chaîne = "Bonjour, la foule !"`)

Si vous utilisez les délimiteurs simples (le guillemet ou l'apostrophe) pour encadrer une chaîne de caractères, il se pose le problème des guillemets ou apostrophes que peut contenir ladite chaîne. Par exemple, si vous tapez `chaîne = 'J'aime Python !'`, vous obtenez le message suivant :

```
  File "<stdin>", line 1
    chaîne = 'J'aime Python !'
                  ^
SyntaxError: invalid syntax. Perhaps you forgot a comma?
```

Ceci est dû au fait que l'apostrophe de « J'aime » est considérée par Python comme la fin de la chaîne et qu'il ne sait pas quoi faire de tout ce qui se trouve au-delà. Pour pallier ce problème, il faut **échapper** les apostrophes se trouvant au cœur de la chaîne. On insère ainsi une barre oblique inverse « \ » avant les apostrophes contenues dans le message.

```
chaîne = 'J\'aime Python !'
```

On doit également échapper les guillemets si on les utilise comme délimiteurs.

```
chaîne2 = "\"Le seul individu formé, c'est celui qui a appris
↪  comment apprendre (...)\" (Karl Rogers, 1976)"
```

Le caractère d'échappement « \ » est utilisé pour créer d'autres signes très utiles. Ainsi, « \n » symbolise un saut de ligne (`"essai\nsur\nplusieurs\nlignes"`). Pour écrire une véritable barre oblique inverse dans une chaîne, il faut l'échapper elle-même (et donc écrire « \\ »).

L'interpréteur affiche les sauts de lignes comme on les saisit, c'est-à-dire sous forme de « \n ». Nous verrons dans la partie suivante comment afficher réellement ces chaînes de caractères et pourquoi l'interpréteur ne les affiche pas comme il le devrait.

Utiliser les triples guillemets pour encadrer une chaîne de caractères dispense d'échapper les guillemets et apostrophes et permet d'écrire plusieurs lignes sans recourir à « \n ».

```
>>> chaîne3 = """Ceci est un nouvel
... essai sur plusieurs
... lignes"""
>>>
```

Notez que les trois chevrons sont remplacés par trois points : cela signifie que l'interpréteur considère que vous n'avez pas fini d'écrire cette instruction. En effet, celle-ci ne s'achève qu'une fois la chaîne refermée avec trois nouveaux guillemets. Les sauts de lignes seront automatiquement remplacés, dans la chaîne, par des « \n ».

Vous pouvez utiliser, à la place des trois guillemets, trois apostrophes qui jouent exactement le même rôle. Je n'utilise personnellement pas ces délimiteurs, mais sachez qu'ils existent et ne soyez pas surpris si vous les voyez un jour dans un code source.

Les chaînes de caractères formatées

Depuis Python 3.6, une nouvelle syntaxe très pratique est apparue pour insérer le contenu de variables dans des chaînes de caractères. Il n'est pas encore temps de la décrire précisément et vous en verrez un très grand nombre d'exemples dans la suite de ce cours, mais en voici une petite présentation.

```
>>> âge = 21
>>> prénom = "Vincent"
>>> phrase = f"Je m'appelle {prénom} et j'ai {âge} ans."
>>> phrase
"Je m'appelle Vincent et j'ai 21 ans."
>>>
```

La ligne qui nous intéresse est la définition de la variable `phrase`. Les guillemets délimitant une chaîne de caractères sont bien présents, mais notez quelques petites choses :

— Avant le guillemet de gauche se trouve la lettre `f`. Elle indique à Python qu'il s'agit d'une chaîne formatée. Si vous oubliez ce petit `f`, Python ne cherchera pas à remplacer vos variables dans la chaîne de caractères.

— Si Python rencontre des noms de variables entourés d'accolades dans la chaîne, il les remplace par leur valeur.

Utiliser des chaînes de caractères formatées de cette façon est bien plus lisible que les alternatives existantes (certaines que nous verrons plus loin). Cette syntaxe est aussi bien plus puissante qu'il n'y paraît, mais il est inutile d'entrer trop dans le détail à ce stade. Gardez simplement à l'esprit qu'elle est encore relativement nouvelle à ce jour et qu'elle ne fonctionne que depuis Python 3.6 (si vous avez installé une version antérieure, c'est une raison supplémentaire pour la mettre à jour).

Voilà, nous avons bouclé le rapide tour d'horizon des types simples. Qualifier les chaînes de caractères de type simple n'est pas strictement vrai mais nous n'allons pas, dans ce chapitre, entrer dans le détail des opérations que l'on peut effectuer dessus. C'est inutile pour l'instant et ce serait hors sujet. Cependant, rien ne vous empêche de tester vous-mêmes quelques opérations comme l'addition et la multiplication (dans le pire des cas, Python vous dira qu'il ne peut pas faire ce que vous lui demandez et, comme nous l'avons vu, il est peu rancunier).

Un petit bonus

Au chapitre précédent, nous avons vu les opérateurs « classiques » pour manipuler des nombres mais aussi, comme on le verra plus tard, d'autres types de données. D'autres opérateurs ont été créés afin de simplifier la manipulation des variables.

Vous serez amenés par la suite, et assez régulièrement, à incrémenter des variables. L'incrémentation désigne l'augmentation de la valeur d'une variable d'un certain nombre. Jusqu'ici, j'ai procédé comme ci-dessous pour augmenter une variable de 1 :

```
variable = variable + 1
```

Cette syntaxe est claire et intuitive mais assez longue et les programmeurs, tout le monde le sait, sont des fainéants nés. On a donc trouvé plus court.

```
variable += 1
```

L'opérateur `+=` revient à ajouter à la variable la valeur qui le suit. Les opérateurs `-=`, `*=` et `/=` existent également, bien qu'ils soient moins utilisés.

Quelques trucs et astuces pour vous faciliter la vie

Python propose un moyen simple de permuter deux variables (échanger leur valeur). Dans d'autres langages, il est nécessaire de passer par une troisième variable qui retient l'une des deux valeurs... ici c'est bien plus simple :

```
>>> a = 5
>>> b = 32
>>> a, b = b, a # permutation
>>> a
32
>>> b
```

```
5
>>>
```

Comme vous le voyez, après l'exécution de la ligne 3, les variables a et b ont échangé leurs valeurs. On retrouvera cette distribution d'affectation bien plus loin.

On peut aussi affecter assez simplement une même valeur à plusieurs variables :

```
>>> x = y = 3
>>> x
3
>>> y
3
>>>
```

Enfin, ce n'est pas encore d'actualité pour vous, mais sachez qu'on peut couper une instruction Python, pour l'écrire sur deux lignes ou plus.

```
>>> 1 + 4 - 3 * 19 + 33 - 45 * 2 + (8 - 3) \
...           -6 + 23.5
-86.5
>>>
```

Comme vous le voyez, le symbole « \ », avant un saut de ligne, indique à Python que « cette instruction se poursuit à la ligne suivante ». Vous pouvez ainsi morceler votre instruction sur plusieurs lignes.

Notez que cette syntaxe est peu agréable à lire. Les programmeurs Python recommandent de ne pas l'utiliser sauf si c'est vraiment nécessaire. Il existe d'autres moyens d'écrire une instruction sur plusieurs lignes et le cas ne se présente pas si souvent de toute façon.

Première utilisation des fonctions

Eh bien, tout cela avance gentiment. Je vais présenter succinctement ici, dans ce chapitre sur les variables, l'utilisation des fonctions. Il s'agit finalement bien davantage d'une application concrète de ce que vous avez appris à l'instant. Un chapitre entier sera consacré aux fonctions, mais utiliser celles que je vais vous montrer n'est pas sorcier et pourra vous être utile.

Utiliser une fonction

À quoi servent les fonctions ?

Une fonction exécute un certain nombre d'instructions déjà enregistrées. En gros, c'est comme si vous enregistriez un groupe d'instructions pour réaliser une action précise

et que vous lui donniez un nom. Vous n'avez plus ensuite qu'à appeler cette fonction par son nom autant de fois que nécessaire (cela évite bon nombre de répétitions). Nous verrons tout cela plus en détail par la suite.

Pour la plupart, les fonctions ont besoin de paramètres pour travailler sur une donnée ; il s'agit d'informations que vous leur passez au moment où vous les appelez. Les fonctions que je vais vous montrer ne font pas exception. Ce concept vous semble peut-être un peu difficile à saisir dans son ensemble mais rassurez-vous, les exemples devraient tout rendre limpide.

Les fonctions s'utilisent en respectant la syntaxe suivante :
`nom_de_la_fonction(paramètre_1,paramètre_2,...,paramètre_n)`.

— Vous commencez par écrire le nom de la fonction.
— Vous placez entre parenthèses les paramètres séparés par des virgules. Si la fonction n'attend aucun paramètre, vous devrez quand même écrire les parenthèses, sans rien entre elles.

La fonction « type »

Dans la partie précédente, je vous ai présenté les types de données simples, du moins une partie d'entre eux. Une des grandes puissances de Python est qu'il comprend automatiquement de quel type est une variable dès son affectation. Toutefois, il vous sera parfois nécessaire de connaître le type d'une variable.

La syntaxe de cette fonction est simple :

```
type(nom_de_la_variable)
```

La fonction renvoie le type de la variable passée en paramètre. Vu que nous sommes dans l'interpréteur de commandes, cette valeur sera affichée. Si vous saisissez dans l'interpréteur les lignes suivantes :

```
>>> a = 3
>>> type(a)
```

Vous obtenez :

```
<class 'int'>
```

Python vous indique donc que la variable `a` appartient à la classe des entiers. Cette notion de classe ne sera pas approfondie avant bien des chapitres mais sachez qu'on peut la rapprocher d'un type de donnée.

Vous pouvez faire le test sans passer par des variables :

```
>>> type(3.4)
<class 'float'>
>>> type("un essai")
<class 'str'>
>>>
```

`str` est l'abréviation de « string » qui signifie chaîne (sous-entendu, de caractères) en anglais.

La fonction `print`

La fonction `print` affiche la valeur d'une ou plusieurs variable(s).

Mais... ne fait-on pas exactement la même chose en saisissant juste le nom de la variable dans l'interpréteur ?

Oui et non. L'interpréteur affiche bien la valeur de la variable car il affiche automatiquement tout ce qu'il peut, pour que vous suiviez les étapes d'un programme. Cependant, quand vous ne travaillerez plus avec l'interpréteur, taper simplement le nom de la variable n'aura aucun effet. De plus, et vous l'aurez sans doute remarqué, l'interpréteur entoure les chaînes de caractères de délimiteurs et affiche les caractères d'échappement, tout ceci encore pour des raisons de clarté.

La fonction `print` est dédiée à l'affichage uniquement. Le nombre de ses paramètres est variable, c'est-à-dire que vous pouvez lui demander d'afficher une ou plusieurs variables. Considérez cet exemple :

```
>>> a = 3
>>> print(a)
>>> a = a + 3
>>> b = a - 2
>>> print("a =", a, "et b =", b)
```

Le premier *appel* à `print` se contente d'afficher la valeur de la variable `a`, c'est-à-dire « 3 ». Le second appel à `print` affiche :

```
a = 6 et b = 4
```

Ce deuxième appel à `print` est peut-être un peu plus dur à comprendre. En fait, on passe quatre paramètres à `print` : deux chaînes de caractères et les variables `a` et `b`. Quand Python interprète cet appel de fonction, il va afficher les paramètres dans l'ordre de passage, en les séparant par un espace.

Relisez bien cet exemple, il montre tout l'intérêt des fonctions. Si vous avez du mal à le comprendre dans son ensemble, décortiquez-le en prenant indépendamment chaque paramètre.

Note : avec les chaînes de caractères formatées, on préférera écrire la ligne précédente comme suit :

```
>>> print(f"a = {a} et b = {b}")
```

C'est bien plus lisible finalement, une fois que l'on est habitué à la syntaxe. Le premier exemple visait à démontrer l'utilisation de fonctions avec plusieurs paramètres. Vous rencontrerez du code utilisant les deux syntaxes.

Testez l'utilisation de `print` avec d'autres types de données et en insérant des chaînes avec des sauts de lignes et des caractères échappés, pour bien vous rendre compte de la différence.

Un petit « Hello World ! » ?

Quand on donne un cours sur un langage, quel qu'il soit, il est d'usage de présenter le programme « Hello World ! », qui en illustre assez rapidement la syntaxe superficielle.

Le but du jeu est très simple : écrire un programme qui affiche « Hello World ! » à l'écran. Dans certains langages, notamment ceux qui sont compilés, vous devrez écrire jusqu'à une dizaine de lignes pour obtenir ce résultat. En Python, comme nous venons de le voir, il en suffit d'une seule :

```
1 >>> print("Hello World !")
```

Pour plus d'informations, n'hésitez pas à consulter la page Wikipédia consacrée à « Hello World ! » ; vous avez même des codes rédigés en différents langages de programmation, cela peut être intéressant.

▷ Wikipédia - « Hello World ! »
Code web : 696192

En résumé

- Les variables permettent de conserver dans le temps des données de votre programme.
- Vous pouvez vous servir de ces variables pour différentes choses : les afficher, faire des calculs avec, etc.
- Pour affecter une valeur à une variable, on utilise la syntaxe `nom_de_variable = valeur`.
- Il existe différents types de variables, en fonction de l'information que vous désirez conserver : `int`, `float`, chaîne de caractères etc.
- Pour afficher une donnée, par exemple la valeur d'une variable, on utilise la fonction `print`.

Chapitre 4

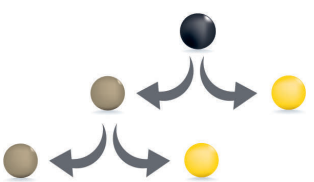
Les structures conditionnelles

Difficulté :

Jusqu'à présent, nous avons testé des instructions d'une façon linéaire : l'interpréteur exécutait au fur et à mesure le code que vous saisissiez dans la console. Cependant, nos programmes seraient bien pauvres si nous ne pouvions, de temps à autre, demander à exécuter certaines instructions dans un cas et d'autres instructions dans un autre cas.

Dans ce chapitre, je vais vous parler des structures conditionnelles, qui vont vous permettre de faire des tests et d'aller plus loin dans la programmation.

Vous finirez ce chapitre en créant votre premier « vrai » programme : même si vous ne pensez pas encore pouvoir faire quelque chose de très consistant, à la fin de ce chapitre vous aurez assez de matière pour coder un petit programme dans un but très précis.



Vos premières conditions et blocs d'instructions

Forme minimale en `if`

Les conditions sont un concept essentiel en programmation (oui oui, je me répète à force mais il faut avouer que des concepts essentiels, on n'a pas fini d'en voir). Grâce à elles, vous programmerez une action précise si, par exemple, une variable est positive, une autre action si cette variable est négative, ou une troisième action si la variable est nulle. Comme un bon exemple vaut mieux que plusieurs lignes d'explications, voici un exemple clair d'une condition prise sous sa forme la plus simple.

Dès à présent dans mes exemples, j'utiliserai des commentaires. Il s'agit de messages qui sont ignorés par l'interpréteur et qui donnent des indications sur le code (car, vous vous en rendrez compte, relire ses programmes après plusieurs semaines d'abandon, sans commentaire, ce peut être parfois plus qu'ardu). En Python, un commentaire débute par un dièse (#) et se termine par un saut de ligne. Tout ce qui est compris entre ce # et ce saut de ligne est ignoré. Un commentaire peut donc occuper la totalité d'une ligne (# en début de ligne) ou une partie seulement, après une instruction (# après la ligne de code pour la commenter plus spécifiquement).

Cela étant posé, revenons à nos conditions :

```
>>> # Premier exemple de condition
... a = 5
>>> if a > 0: # Si a est supérieur à 0
...     print("a est supérieur à 0.")
...
a est supérieur à 0.
>>>
```

Détaillons ce code, ligne par ligne :

1. La première ligne est un commentaire décrivant qu'il s'agit du premier test de condition. Elle est ignorée par l'interpréteur et sert juste à vous renseigner sur le code qui va suivre.
2. Cette ligne, vous devriez la comprendre sans aucune aide. On se contente d'affecter la valeur 5 à la variable `a`.
3. Ici se trouve notre test conditionnel. Il se compose, dans l'ordre :
   — du mot-clé `if` qui signifie « si » en anglais ;
   — de la condition proprement dite, `a > 0`, qu'il est facile de lire (une liste des opérateurs autorisés pour la comparaison sera présentée plus bas) ;
   — du signe deux-points, « : », qui termine la condition et est indispensable : Python affichera une erreur de syntaxe si vous l'omettez.
4. Ici se trouve l'instruction à exécuter dans le cas où `a` est supérieur à `0`. Après que vous avez appuyé sur Entrée à la fin de la ligne précédente, l'interpréteur

vous présente la série de trois points qui signifie qu'il attend la saisie du **bloc d'instructions** concerné avant de l'interpréter. Cette instruction (et les autres instructions à exécuter s'il y en a) est indentée, c'est-à-dire décalée vers la droite. Des explications supplémentaires seront données un peu plus bas sur les indentations.

5. L'interpréteur vous affiche à nouveau la série de trois points et vous pouvez en profiter pour saisir une nouvelle instruction dans ce bloc d'instructions. Ce n'est pas le cas pour l'instant. Vous appuyez donc sur Entrée sans avoir rien écrit et l'interpréteur vous affiche le message « a est supérieur à 0 », ce qui est assez logique vu que `a` vaut `5`.

Il y a deux notions importantes et complémentaires sur lesquelles je dois à présent revenir.

La première est celle de bloc d'instructions. Il s'agit d'une série d'instructions qui s'exécutent dans un cas précis (par condition, comme on vient de le voir, ou par répétition, comme on le verra plus tard). Ici, nous n'avons écrit qu'une seule instruction (la ligne 4 qui fait appel à `print`), mais rien ne vous empêche de mettre plusieurs instructions dans un bloc.

```
1 a = 5
2 b = 8
3 if a > 0:
4     # On incrémente la valeur de b
5     b += 1
6     # On affiche les valeurs des variables
7     print("a =", a, "et b =", b)
```

La seconde notion importante est celle d'indentation. Cela consiste en un certain décalage vers la droite, obtenu par un (ou plusieurs) espace(s) ou tabulation(s).

Les indentations sont essentielles pour Python. Il ne s'agit pas d'un confort de lecture, comme dans d'autres langages tels que C++ ou Java, mais bien d'un moyen pour l'interpréteur de savoir où se trouvent le début et la fin d'un bloc.

Forme complète (`if`, `elif` et `else`)

Les limites de la condition simple en `if`

La première forme de condition que l'on vient de voir est pratique, mais assez incomplète.

Considérons, par exemple, une variable `a` de type entier. On souhaite faire une action si cette variable est positive et une action différente si elle est négative. Il est possible d'obtenir ce résultat avec la forme simple d'une condition :

```
1 >>> a = 5
2 >>> if a > 0: # Si a est positif
3 ...     print("a est positif.")
4 ... if a < 0: # a est négatif
5 ...     print("a est négatif.")
```

Amusez-vous à changer la valeur de `a` et exécutez à chaque fois les conditions ; vous obtiendrez des messages différents, sauf si `a` vaut `0`. En effet, aucune action n'a été prévue pour ce cas.

Cette méthode n'est pas optimale, tout d'abord parce qu'elle nous oblige à écrire deux conditions séparées pour tester une même variable. De plus, et même si c'est dur à concevoir par cet exemple, dans le cas où la variable remplirait les deux conditions (ici c'est impossible bien entendu), les deux portions de code s'exécuteraient.

La condition `if` est donc bien pratique mais insuffisante.

L'instruction `else`

Le mot-clé `else`, qui signifie « sinon » en anglais, permet de définir une première forme de complément à notre instruction `if`.

```
>>> âge = 21
>>> if âge >= 18: # Si âge est supérieur ou égal à 18
...     print("Vous êtes majeur.")
... else: # Sinon (âge inférieur à 18)
...     print("Vous êtes mineur.")
```

Je pense que cet exemple suffit amplement à exposer l'utilisation de `else`. La seule subtilité est de bien se rendre compte que Python exécute soit l'un, soit l'autre, mais jamais les deux. Notez que cette instruction `else` doit se trouver au même niveau d'indentation que le `if` qu'elle complète. De plus, elle se termine également par deux points puisqu'il s'agit d'une condition, même si elle est sous-entendue.

L'exemple précédent pourrait donc se présenter comme suit, avec l'utilisation de `else` :

```
>>> a = 5
>>> if a > 0:
...     print("a est supérieur à 0.")
... else:
...     print("a est inférieur ou égal à 0.")
```

Mais... le résultat n'est pas tout à fait le même, si ?

Non, en effet. Vous vous rendrez compte que, cette fois, le cas où `a` vaut `0` est bien pris en compte. En effet, la condition initiale prévoit d'exécuter le premier bloc d'instructions si `a` est strictement supérieur à `0`. Sinon, on exécute le second bloc d'instructions.

Si l'on veut faire la différence entre les nombres positifs, négatifs et nuls, il va falloir utiliser une condition intermédiaire.

L'instruction `elif`

Le mot-clé `elif` est une contraction de « else if », que l'on peut traduire très littéralement par « sinon si ». Dans l'exemple que nous venons juste de voir, l'idéal serait d'écrire :

— si `a` est strictement supérieur à `0`, on dit qu'il est positif ;
— sinon si `a` est strictement inférieur à `0`, on dit qu'il est négatif ;
— sinon, (`a` ne peut qu'être égal à `0`), on dit alors que `a` est nul.

Traduit en langage Python, cela donne :

```
>>> if a > 0: # Positif
...     print("a est positif.")
... elif a < 0: # Négatif
...     print("a est négatif.")
... else: # Nul
...     print("a est nul.")
```

De même que le `else`, le `elif` est sur le même niveau d'indentation que le `if` initial. Il se termine aussi par deux points. Cependant, entre le `elif` et les deux points se trouve une nouvelle condition. Linéairement, le schéma d'exécution se traduit comme suit :

1. On regarde si `a` est strictement supérieur à `0`. Si c'est le cas, on affiche « a est positif » et on s'arrête là.
2. Sinon, on regarde si `a` est strictement inférieur à `0`. Si c'est le cas, on affiche « a est négatif » et on s'arrête.
3. Sinon, on affiche « a est nul ».

Attention : quand je dis « on s'arrête », c'est uniquement pour cette condition. S'il y a du code après les trois blocs d'instructions, il sera exécuté dans tous les cas.

Vous pouvez mettre autant de `elif` que vous voulez après une condition en `if`. Tout comme le `else`, cette instruction est facultative et, quand bien même vous construiriez une instruction en `if`, `elif`, vous n'êtes pas du tout obligé de prévoir un `else` après. En revanche, l'instruction `else` ne peut figurer qu'une fois, clôturant le bloc de la condition. Deux instructions `else` dans une même condition ne sont pas envisageables et n'auraient de toute façon aucun sens.

Sachez qu'il est heureusement possible d'imbriquer des conditions et, dans ce cas, l'indentation permet de comprendre clairement le schéma d'exécution du programme. Je vous laisse essayer cette possibilité, je ne vais pas tout faire à votre place non plus. :-)

En quête de correspondance

Vous avez pu le voir, parfois on cherche à tester une variable plusieurs fois. Voici un petit exemple :

```
>>> choix = 2
>>> if choix == 0:
...     print("Vous avez choisi zéro.")
... elif choix == 1:
...     print("Vous avez choisi un.")
... elif choix == 2:
...     print("Vous avez choisi deux.")
... else:
...     print("Vous avez choisi autre chose.")
...
Vous avez choisi deux.
```

Dans cet exemple simpliste, on teste la même variable contre plusieurs valeurs possibles (`0`, `1`, `2` ou autre chose). Il y a plus d'une partie redondante dans ce code. Celle qui nous intéresse est la répétition `choix == ...` pour chaque `if` ou `elif`.

Il existe depuis Python 3.10 une nouvelle manière d'écrire la condition précédente. Elle est plus concise et un peu plus lisible. Elle s'aligne sur une fonctionnalité proposée par d'autres langages de programmation (si vous connaissez le C, elle se rapproche du `switch`). Je présente ici cette syntaxe, mais elle est bien plus puissante et utile que ce que je vais pouvoir démontrer au travers de l'exemple qui suit ; nous reviendrons à ces deux mots-clés plus tard.

Voici le même code réécrit avec deux nouveaux mots-clés : `match` et `case` :

```
>>> choix = 2
>>> match choix: # On va comparer la variable choix à
 ↪  plusieurs choix possibles
...     case 0: # choix vaut 0
...         print("Vous avez choisi zéro.")
...     case 1: # choix vaut 1
...         print("Vous avez choisi un.")
...     case 2: # choix vaut 2
...         print("Vous avez choisi deux.")
...     case _: # choix vaut autre chose
...         print("Vous avez choisi autre chose.")
...
Vous avez choisi deux.
```

Ce code reste relativement semblable au premier. Voyons dans le détail :

1. On affecte la variable `choix` à `2`.
2. On entre le mot-clé `match` (correspondre) suivi du nom de la variable et du signe deux points. À l'intérieur du `match`, on va donc essayer de trouver une correspondance, une branche qui correspond à la valeur de la variable (ici `choix`).
3. On entre la première branche avec le mot-clé `case` (cas). Le mot-clé est suivi de la valeur proposée et de deux points. On peut donc traduire cette ligne par

`si la variable choix vaut 0`. C'est d'ailleurs ce que nous avons fait au premier exemple. Notez que `case` est indenté au-dessous de `match`.

4. On rentre dans le cas (si `choix` vaut `0`). C'est un bloc d'instructions. On n'entre qu'une seule instruction ici (elle est indentée sous le bloc `case` et donc de deux niveaux par rapport au `match`).
5. On a aussi deux autres branches décrites par le mot-clé `case`.
6. Si aucune branche n'est trouvée, on exécute celle par défaut (`case _:`). Elle est facultative, comme l'est le `else`.

Tout comme pour le `if`, Python va comparer chaque `case` à la valeur du `match`, choisir une branche, l'exécuter et passer à la suite. À ce stade, on ne voit guère de différence entre `if` et `match`.

Pourtant, ce sont des concepts assez distincts, même à notre niveau. `if` permet de tester une condition alors que `match` compare deux valeurs. À première vue, `match` est donc inférieur. Cependant `match` permet bien d'autres choses. Pour l'heure, sachez que cette syntaxe existe ; nous verrons où l'utiliser par la suite.

De nouveaux opérateurs

Les opérateurs de comparaison

Les conditions nécessitent de nouveaux opérateurs, dits **opérateurs de comparaison**. Je vais les présenter très brièvement, vous laissant l'initiative de faire des tests car ils ne sont réellement pas difficiles à comprendre.

Opérateur	Signification littérale
<	Strictement inférieur à
>	Strictement supérieur à
<=	Inférieur ou égal à
>=	Supérieur ou égal à
==	Égal à
!=	Différent de

Attention : l'égalité de deux valeurs est comparée avec l'opérateur « == » et non « = ». Ce dernier est en effet l'opérateur d'affectation et ne doit pas être utilisé dans une condition.

Prédicats et booléens

Avant d'aller plus loin, sachez que les conditions qui se trouvent, par exemple, entre `if` et les deux points sont appelés des **prédicats**. Vous pouvez les tester directement dans l'interpréteur pour comprendre les explications qui vont suivre.

```
>>> a = 0
>>> a == 5
False
>>> a > -8
True
>>> a != 33.19
True
>>>
```

L'interpréteur renvoie tantôt `True` (c'est-à-dire « vrai »), tantôt `False` (c'est-à-dire « faux »).

`True` et `False` sont les deux valeurs possibles d'un type que nous n'avons pas vu jusqu'ici : le type booléen (`bool`).

N'oubliez pas que `True` et `False` sont des valeurs ayant leur première lettre en majuscule. Si vous commencez à écrire `true` sans un 't' majuscule, Python ne va pas comprendre.

Les variables de ce type ne peuvent prendre comme valeur que vrai ou faux et sont pratiques, justement, pour stocker des prédicats, de la façon que nous avons vue ou d'une façon plus détournée.

```
>>> âge = 21
>>> majeur = False
>>> if âge >= 18:
...     majeur = True
...
>>>
```

À la fin de cet exemple, majeur vaut `True`, c'est-à-dire « vrai », si l'âge est supérieur ou égal à 18. Sinon, il continue de valoir `False`. Les booléens ne vous semblent peut-être pas très utiles pour l'instant mais vous verrez qu'ils rendent de grands services !

Les mots-clés `and`, `or` et `not`

Il arrive souvent que nos conditions doivent tester plusieurs prédicats, par exemple quand on cherche à vérifier si une variable quelconque, de type entier, se trouve dans un intervalle précis (c'est-à-dire comprise entre deux nombres). Avec nos méthodes actuelles, le plus simple serait d'écrire :

```
# On fait un test pour savoir si a est comprise dans
↪ l'intervalle allant de 2 à 8 inclus
a = 5
if a >= 2:
    if a <= 8:
```

```
        print("a est dans l'intervalle.")
    else:
        print("a n'est pas dans l'intervalle.")
else:
    print("a n'est pas dans l'intervalle.")
```

Cela fonctionne mais c'est assez lourd, d'autant que, pour être sûr qu'un message soit affiché à chaque fois, il faut fermer chacune des deux conditions à l'aide d'un `else` (la seconde étant imbriquée dans la première). Si vous avez du mal à comprendre cet exemple, prenez le temps de le décortiquer, ligne par ligne.

Il existe cependant le mot-clé `and` (qui signifie « et » en anglais) qui va nous rendre ici un fier service. En effet, on cherche à tester si `a` est à la fois supérieur ou égal à 2 et inférieur ou égal à 8. On peut donc réduire ainsi les conditions imbriquées :

```
if a >= 2 and a <= 8:
    print("a est dans l'intervalle.")
else:
    print("a n'est pas dans l'intervalle.")
```

C'est simple et bien plus compréhensible, avouez-le.

Sur le même mode, il existe le mot-clé `or` qui signifie cette fois « ou ». Nous allons prendre le même exemple, sauf que nous allons évaluer notre condition différemment.

Nous allons chercher à savoir si `a` *n'est pas* dans l'intervalle, c'est-à-dire si la variable est inférieure à 2 ou supérieure à 8. Voici le code :

```
if a < 2 or a > 8:
    print("a n'est pas dans l'intervalle.")
else:
    print("a est dans l'intervalle.")
```

Enfin, il existe le mot-clé `not` qui « inverse » un prédicat. Le prédicat `not a == 5` équivaut donc à `a != 5`.

Je ne vous propose pas d'exemple pour le mot-clé `not` pour l'heure, mais vous le retrouverez par la suite.

Je vous invite à tester des prédicats plus complexes de la même façon que les précédents, en les saisissant directement, sans le `if` ni les deux points, dans l'interpréteur de commande. Vous pouvez utiliser les parenthèses ouvrantes et fermantes pour encadrer des prédicats et les comparer suivant des priorités bien précises (nous verrons ce point plus loin).

Votre premier programme !

À quoi joue-t-on ?

L'heure du premier TP est venue. Comme il s'agit du tout premier, et parce qu'il y a quelques indications que je dois vous donner pour que vous parveniez jusqu'au bout, je vous accompagnerai pas à pas dans sa réalisation.

Avant de commencer

Vous allez dans cette section écrire votre premier programme. Vous allez sûrement tester les syntaxes directement dans l'interpréteur de commandes.

Si vous préférez écrire votre code directement dans un fichier que vous pouvez ensuite exécuter, je vous renvoie au chapitre 38 traitant de ce point, que vous trouverez à la page 447 de ce livre.

Sujet

Le but de notre programme est de déterminer si une année saisie par l'utilisateur est bissextile. Il s'agit d'un sujet très prisé des enseignants en informatique quand il s'agit d'expliquer les conditions. Mille pardons, donc, à ceux qui ont déjà fait cet exercice dans un autre langage, mais je trouve que ce petit programme reprend assez de thèmes abordés dans ce chapitre pour être réellement intéressant.

Une année est dite bissextile si c'est un multiple de 4, sauf si c'est un multiple de 100. Toutefois, elle est considérée comme bissextile si c'est un multiple de 400. Je développe :

- Si une année n'est pas multiple de 4, on s'arrête là, elle n'est pas bissextile.
- Si elle est multiple de 4, on regarde si elle est multiple de 100.
  - Si c'est le cas, on regarde si elle est multiple de 400.
    - Si c'est le cas, l'année est bissextile.
    - Sinon, elle n'est pas bissextile.
  - Sinon, elle est bissextile.

Solution ou résolution

Le problème est posé clairement ; il faut maintenant réfléchir à sa résolution en termes de programmation. C'est une phase de transition assez délicate de prime abord et je vous conseille de schématiser le problème, de prendre des notes sur les différentes étapes, sans pour l'instant penser au code. C'est une phase purement algorithmique ; autrement dit, on réfléchit au programme sans s'attacher au code proprement dit.

Vous aurez besoin, pour réaliser ce petit programme, de quelques indications qui sont réellement spécifiques à Python. Ne lisez donc ceci qu'après avoir cerné et clairement écrit le problème d'une façon plus algorithmique. Cela étant dit, si vous peinez à trouver une solution, ne vous y attardez pas. Cette phase de réflexion est assez difficile au début et, parfois, il suffit d'un peu de pratique et d'explications pour comprendre l'essentiel.

La fonction `input()`

Tout d'abord, j'ai mentionné une année saisie par l'utilisateur. En effet, depuis le début du chapitre, nous testons des variables que nous déclarons nous-mêmes, avec une valeur précise. La condition est donc assez ridicule.

`input()` est une fonction qui va, pour nous, caractériser nos premières interactions avec l'utilisateur : le programme réagira différemment en fonction du nombre saisi par ce dernier.

`input()` accepte un paramètre facultatif : le message à afficher à l'utilisateur. Cette instruction interrompt le programme et attend que l'utilisateur saisisse ce qu'il veut puis appuie sur `Entrée`. À cet instant, la fonction renvoie ce que l'utilisateur a saisi. Il faut donc piéger cette valeur dans une variable.

```
>>> # Test de la fonction input
>>> année = input("Saisissez une année : ")
Saisissez une année : 2009
>>> année
'2009'
>>>
```

Il subsiste un problème : le type de la variable `année` après l'appel à `input()` est... une chaîne de caractères. Vous pouvez vous en rendre compte grâce aux apostrophes qui encadrent la valeur de la variable quand vous l'affichez directement dans l'interpréteur.

C'est bien ennuyeux car nous voulions travailler sur un entier. Pour convertir une variable vers un autre type, il faut utiliser le nom du type comme une fonction (c'est d'ailleurs exactement ce que c'est).

```
>>> type(année)
<class 'str'>
>>> # On veut convertir la variable en un entier, on utilise
>>> # donc la fonction int qui prend en paramètre la variable
>>> # d'origine
>>> année = int(année)
>>> type(année)
<class 'int'>
>>> année
2009
>>>
```

Bon, parfait ! On a donc maintenant l'année sous sa forme entière. Notez que, si vous saisissez des lettres lors de l'appel à `input()`, la conversion renverra une erreur.

L'appel à la fonction `int()` en a peut-être déconcerté certains. On passe en paramètre de cette fonction la variable contenant la chaîne de caractères issue de `input()`, pour tenter de la convertir. La fonction `int()` renvoie la valeur convertie en entier et on la récupère donc dans la même variable. On évite ainsi de travailler sur plusieurs variables, sachant que la première n'a plus aucune utilité à présent qu'on l'a convertie.

Test de multiples

Certains pourraient également se demander comment tester si un nombre a est multiple d'un nombre b. Il suffit, en fait, de tester le reste de la division entière de b par a. Si ce reste est nul, alors a est un multiple de b.

```
>>> 5 % 2 # 5 n'est pas un multiple de 2
1
>>> 8 % 2 # 8 est un multiple de 2
0
>>>
```

À vous de jouer

Je pense vous avoir donné tous les éléments nécessaires pour réussir. À mon avis, le plus difficile est la phase de réflexion qui précède la composition du programme. Si vous avez du mal à réaliser cette opération, passez à la correction et étudiez-la soigneusement. Sinon, on se retrouve à la section suivante.

Bonne chance !

Correction

C'est l'heure de comparer nos méthodes et, avant de vous divulguer le code de ma solution, je vous précise qu'elle est loin d'être la seule possible. Vous pouvez très bien avoir trouvé quelque chose de différent mais qui fonctionne correctement.

Attention... la voiiiciiiiiii...

```
# Programme testant si une année, saisie par l'utilisateur,
# est bissextile ou non

année = input("Saisissez une année : ") # On attend que
 ↪  l'utilisateur saisisse l'année qu'il désire tester
année = int(année) # Risque d'erreur si l'utilisateur n'a pas
 ↪  saisi un nombre
```

```
6  bissextile = False # On crée un booléen qui vaut vrai ou faux
7                     # selon que l'année est bissextile ou non
8
9  if année % 400 == 0:
10     bissextile = True
11 elif année % 100 == 0:
12     bissextile = False
13 elif année % 4 == 0:
14     bissextile = True
15 else:
16     bissextile = False
17
18 if bissextile: # Si l'année est bissextile
19     print("L'année saisie est bissextile.")
20 else:
21     print("L'année saisie n'est pas bissextile.")
```

Je vous rappelle que vous pouvez enregistrer vos codes dans des fichiers afin de les exécuter. Je vous renvoie à la page 447 pour plus d'informations.

Je pense que le code est assez clair, mais il reste à expliciter l'enchaînement des conditions. Vous remarquerez qu'on a inversé le problème. On teste d'abord si l'année est un multiple de 400, ensuite si c'est un multiple de 100 et enfin si c'est un multiple de 4. En effet, le `elif` garantit que, si `annee` est un multiple de 100, ce n'est pas un multiple de 400 (car le cas a été traité juste avant). De cette façon, on s'assure que tous les cas sont gérés. Faites des essais avec plusieurs années pour vous rendre compte si le programme a raison ou pas.

L'utilisation de `bissextile` comme d'un prédicat à part entière vous a peut-être déconcertés. C'est tout à fait possible et logique, puisque c'est un booléen. Il est vrai ou faux et donc on peut le tester simplement. La forme `if bissextile == True:` revient au même.

Un peu d'optimisation

Ce qu'on a fait était bien, mais on peut l'améliorer. D'ailleurs, vous vous rendrez compte que c'est presque toujours le cas. Ici, il s'agit bien entendu de notre condition, que je vais passer au crible afin d'en construire une plus courte et plus logique. On peut parler d'optimisation dans ce cas, même si l'optimisation intègre aussi et surtout les ressources consommées par votre application, en vue de les diminuer et d'accélérer l'application. Cependant, pour une petite application comme celle-ci, on ne s'attardera pas sur l'optimisation du temps d'exécution.

Le premier détail que vous avez pu remarquer, c'est que le `else` de fin est inutile. En effet, la variable `bissextile` vaut par défaut `False` et conserve donc cette valeur si le cas n'est pas traité (ici, quand l'année n'est ni un multiple de 400, ni un multiple de 100, ni un multiple de 4).

Ensuite, il apparaît que nous pouvons faire un grand ménage dans notre condition car les deux seuls cas correspondant à une année bissextile sont « si l'année est un multiple de 400 » ou « si l'année est un multiple de 4 mais pas de 100 ».

Le prédicat correspondant est un peu délicat, il fait appel aux priorités des parenthèses. Il est important que vous le compreniez bien à présent.

```
1 # Programme testant si une année, saisie par l'utilisateur,
  ↪ est bissextile ou non
2
3 année = input("Saisissez une année : ") # On attend que
  ↪ l'utilisateur saisisse l'année qu'il désire tester
4 année = int(année) # Risque d'erreur si l'utilisateur n'a pas
  ↪ saisi un nombre
5
6 if année % 400 == 0 or (année % 4 == 0 and année % 100 != 0):
7     print("L'année saisie est bissextile.")
8 else:
9     print("L'année saisie n'est pas bissextile.")
```

▷ Copier ce code
Code web : 886842

Du coup, on n'a plus besoin de la variable `bissextile`, c'est déjà cela de gagné. Nous sommes passés de 16 lignes de code à seulement 7 (sans compter les commentaires et les sauts de ligne) ce qui n'est pas rien.

Optimiser est utile jusqu'à un certain point, mais ne tombez pas dans l'excès non plus. Il existe une maladie bien connue appelée l'optimisation prématurée, qui consiste à passer trop de temps sur une optimisation mineure. Gardez à l'esprit que l'optimisation peut réduire le nombre de lignes de code, tout en réduisant sa lisibilité. Enfin, le nombre de lignes de code n'est absolument pas un facteur d'optimisation et je fais parfois le choix d'écrire une opération en plusieurs lignes plutôt que d'aller trop vite et tout regrouper dans une ligne difficile à lire. C'est, pour tout dire, un art que vous apprendrez en pratiquant.

En résumé

— Les conditions permettent d'exécuter certaines instructions dans certains cas, d'autres instructions dans un autre cas.
— Les conditions sont marquées par les mot-clés `if` (« si »), `elif` (« sinon si ») et `else` (« sinon »).
— Les mot-clés `if` et `elif` doivent être suivis d'un test (appelé aussi prédicat).
— Les booléens sont des données soit vraies (`True`) soit fausses (`False`).

Chapitre 5

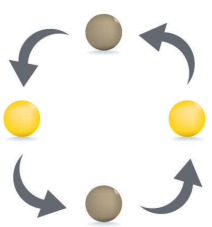
Les boucles

Difficulté :

Les boucles sont un concept nouveau pour vous. Elles vont vous permettre de répéter une certaine opération autant de fois que nécessaire. Le concept risque de vous sembler un peu théorique car les applications pratiques présentées dans ce chapitre ne vous paraîtront probablement pas très intéressantes. Toutefois, il est impératif que cette notion soit comprise avant que vous ne passiez à la suite. Viendra vite le moment où vous aurez du mal à écrire une application sans boucle.

En outre, les boucles servent à parcourir certaines séquences comme les chaînes de caractères pour, par exemple, en extraire chaque caractère.

Alors, on commence ?



En quoi cela consiste-t-il ?

Les boucles constituent un moyen de répéter un certain nombre de fois des instructions de votre programme. Prenons un exemple simple, même s'il est assez peu réjouissant en lui-même : écrire un programme affichant la table de multiplication par 7, de 1 * 7 à 10 * 7.

... bah quoi ?

Bon, ce n'est qu'un exemple, ne faites pas cette tête, et puis je suis sûr que ce sera utile pour certains. Dans un premier temps, vous devriez arriver au programme suivant :

```
1  print(" 1 * 7 =", 1 * 7)
2  print(" 2 * 7 =", 2 * 7)
3  print(" 3 * 7 =", 3 * 7)
4  print(" 4 * 7 =", 4 * 7)
5  print(" 5 * 7 =", 5 * 7)
6  print(" 6 * 7 =", 6 * 7)
7  print(" 7 * 7 =", 7 * 7)
8  print(" 8 * 7 =", 8 * 7)
9  print(" 9 * 7 =", 9 * 7)
10 print("10 * 7 =", 10 * 7)
```

En voici le résultat :

```
1  1 * 7 = 7
2  2 * 7 = 14
3  3 * 7 = 21
4  4 * 7 = 28
5  5 * 7 = 35
6  6 * 7 = 42
7  7 * 7 = 49
8  8 * 7 = 56
9  9 * 7 = 63
10 10 * 7 = 70
```

Je vous rappelle que vous pouvez enregistrer vos codes dans des fichiers. Vous trouverez la marche à suivre à la page 447 de ce livre.

Bon, c'est sûrement la première idée qui vous est venue et cela fonctionne, très bien même. Seulement, vous reconnaîtrez qu'un programme comme cela n'est pas bien utile. Essayons donc le même programme mais, cette fois-ci, en utilisant une variable ; ainsi, si on décide d'afficher la table de multiplication de 6, on n'aura qu'à changer la valeur de la variable ! Pour cet exemple, on utilise une variable `nb` qui contiendra 7. Les instructions seront légèrement différentes mais vous devriez toujours pouvoir écrire ce programme :

```
nb = 7
print(f" 1 * {nb} = {1 * nb}")
print(f" 2 * {nb} = {2 * nb}")
print(f" 3 * {nb} = {3 * nb}")
print(f" 4 * {nb} = {4 * nb}")
print(f" 5 * {nb} = {5 * nb}")
print(f" 6 * {nb} = {6 * nb}")
print(f" 7 * {nb} = {7 * nb}")
print(f" 8 * {nb} = {8 * nb}")
print(f" 9 * {nb} = {9 * nb}")
print(f"10 * {nb} = {10 * nb}")
```

Nous utilisons cette fois les chaînes formatées (avec le préfixe `f`). On aurait très bien pu écrire ce programme en donnant plusieurs paramètres à `print`, mais utiliser les chaînes formatées ici rend le code plus lisible (de mon point de vue). Il faut malgré tout s'habituer à la syntaxe. Pour chaque ligne, regardez ce qu'il y a entre les accolades :

1. `nb` : le nom de la variable `nb` tout simplement. Son contenu (7 dans notre cas) sera toujours inséré.
2. `X * nb` : cette fois, au lieu de demander à Python d'afficher une variable, on lui fait afficher le résultat d'un véritable calcul. Pour chaque ligne, notre X est différent (vous constatez que le code est répétitif).

Si vous êtes un peu perdus, n'hésitez pas à faire des tests ; cette syntaxe est utile et lisible, une fois qu'on y est habitué. N'oubliez pas non plus le `f` avant le guillemet (ou l'apostrophe) ouvrant la chaîne de caractères, c'est une source d'erreurs assez fréquente au début.

Le résultat est le même, vous pouvez le vérifier. Cependant, le code est quand même un peu plus intéressant : on peut changer la table de multiplication à afficher en modifiant la valeur de la variable `nb`.

Néanmoins, ce programme reste assez peu pratique et il accomplit une tâche bien répétitive. Les programmeurs étant très paresseux, ils préfèrent utiliser les boucles.

La boucle `while`

La boucle que je vais présenter se retrouve dans la plupart des autres langages de programmation et porte le même nom. Elle sert à répéter un **bloc d'instructions** tant qu'une condition est vraie (`while` signifie « tant que » en anglais). J'espère que le concept de bloc d'instructions est clair pour vous, sinon je vous renvoie au chapitre précédent.

La syntaxe de `while` est la suivante :

```
while condition:
    # instruction 1
    # instruction 2
    # ...
    # instruction N
```

Vous devriez reconnaître la forme d'un bloc d'instructions, du moins je l'espère.

Quelle condition va-t-on utiliser ?

Eh bien, c'est là le point important. Dans cet exemple, on va créer une variable qui sera incrémentée dans le bloc d'instructions. Tant que cette variable sera inférieure à 10, le bloc s'exécutera pour afficher la table.

Si ce n'est pas clair, regardez le code qui suit ; quelques commentaires suffiront pour le comprendre :

```
1 nb = 7 # On garde la variable contenant le nombre dont on veut
  ↪ la table de multiplication
2 i = 1 # C'est notre variable compteur que nous allons
  ↪ incrémenter dans la boucle
3
4 while i <= 10: # Tant que i est inférieure ou égale à 10
5     print(f"{i} * {nb} = {i * nb}")
6     i += 1 # On incrémente i de 1 à chaque tour de boucle
```

Analysons ce code ligne par ligne :

1. On instancie la variable `nb` qui accueille le nombre sur lequel nous allons travailler (en l'occurence, 7). Vous pouvez bien entendu demander à l'utilisateur de saisir ce nombre ; vous savez à présent comment le programmer.
2. On instancie la variable `i` qui sera notre compteur durant la boucle. `i` est un standard utilisé quand il est question de boucles et de variables s'incrémentant mais il va de soi que vous auriez pu lui donner un autre nom. On l'initialise à 1.
3. Un saut de ligne ne fait jamais de mal !
4. On trouve ici l'instruction `while` qui se décode, comme je l'ai indiqué en commentaire, en « **tant que i est inférieure ou égale à 10** ». N'oubliez pas les deux points à la fin de la ligne.
5. La ligne du `print`, vous devez la reconnaître. Maintenant, la plus grande partie de la ligne affichée est constituée de variables, à part les signes mathématiques.
6. Ici, on incrémente la variable `i` de 1. Si on est dans le premier tour de boucle, `i` passe donc de 1 à 2. Et alors, puisqu'il s'agit de la fin du bloc, on revient à l'instruction `while`. Cette dernière vérifie que la valeur de `i` est toujours inférieure à 10. Si c'est le cas (et ça l'est pour l'instant), on exécute à nouveau le bloc d'instructions. En tout, on exécute ce bloc 10 fois, jusqu'à ce que `i` passe de 10 à 11. Alors, l'instruction `while` vérifie la condition, se rend compte qu'elle est à présent fausse (la valeur de `i` n'est plus inférieure à 10) et s'arrête. S'il y avait du code après le bloc, il serait à présent exécuté.

N'oubliez pas d'incrémenter `i` ! Sinon, vous créez ce qu'on appelle une boucle infinie, puisque la valeur de `i` n'est jamais supérieure à 10 et la condition du `while`, par conséquent, toujours vraie... La boucle s'exécute donc à l'infini, du moins en théorie. Si votre ordinateur se lance dans une boucle infinie à cause de votre programme, vous devrez taper CTRL + C dans la fenêtre de l'interpréteur (sous Windows ou Linux) pour l'interrompre. Python ne le fera pas tout seul car, pour lui, il se passe bel et bien quelque chose. De toute façon, il est incapable de différencier une boucle infinie d'une boucle finie : c'est au programmeur de le faire.

La boucle `for`

Comme je l'ai dit précédemment, on retrouve l'instruction `while` dans la plupart des autres langages. En C++ ou en Java, on retrouve également des instructions `for` mais qui n'ont pas le même sens. Notez que, si vous avez déjà programmé en Perl ou en PHP, vous pouvez retrouver les boucles `for` sous un mot-clé assez proche : `foreach`.

L'instruction `for` travaille sur des séquences. Elle est en fait spécialisée dans le parcours d'une séquence de plusieurs données. Nous n'avons pas vu (et nous ne verrons pas tout de suite) ces séquences assez particulières mais très répandues, même si elles peuvent se révéler complexes. Toutefois, il en existe un type que nous avons rencontré depuis quelque temps déjà : les chaînes de caractères.

Les chaînes de caractères sont des séquences... de caractères ! Vous pouvez parcourir une chaîne de caractères (ce qui est également possible avec `while` mais nous verrons plus tard comment). Pour l'instant, intéressons-nous à `for`.

L'instruction `for` se construit comme suit :

```
for element in sequence:
    # instruction 1
    # instruction 2
    # ...
    # instruction N
```

`element` est une variable créée par le `for`, ce n'est pas à vous de l'instancier. Elle prend successivement chacune des valeurs figurant dans la séquence parcourue.

Ce n'est pas très clair ? Alors, comme d'habitude, tout s'éclaire avec le code !

```
chaîne = "Bonjour les ZER0S"
for lettre in chaîne:
    print(lettre)
```

Cela nous donne le résultat suivant :

```
B
o
n
j
o
u
r

l
e
s

Z
E
R
0
S
```

En fait, la variable `lettre` prend successivement la valeur de chaque lettre contenue dans la chaîne de caractères (d'abord B, puis o, puis n...). On affiche ces valeurs avec `print` et cette fonction revient à la ligne après chaque message, ce qui fait que toutes les lettres sont sur une seule colonne. Littéralement, la ligne 2 signifie **« pour lettre dans chaîne »**. Arrivé à cette ligne, l'interpréteur va créer une variable `lettre` qui contiendra le premier élément de la chaîne (autrement dit, la première lettre). Après l'exécution du bloc, la variable `lettre` contient la seconde lettre, et ainsi de suite tant qu'il y a une lettre dans la chaîne.

Notez bien que, du coup, il est inutile d'incrémenter la variable `lettre` (ce qui serait d'ailleurs assez ridicule vu que ce n'est pas un nombre). Python se charge de l'incrémentation, c'est l'un des grands avantages de l'instruction `for`.

À l'instar des conditions que nous avons vues jusqu'ici, `in` peut être utilisée ailleurs que dans une boucle `for`.

```
chaîne = "Bonjour les ZER0S"
for lettre in chaîne:
    if lettre in "AEIOUYaeiouy": # lettre est une voyelle
        print(lettre)
    else: # lettre est une consonne... ou plus exactement,
    ↪  lettre n'est pas une voyelle
        print("*")
```

Cela donne :

```
*
o
*
*
o
```

```
u
*
*
*
e
*
*
*
E
*
*
*
```

Voilà ! L'interpréteur affiche les lettres si ce sont des voyelles et des astérisques sinon. Notez bien que le 0 n'est pas affiché à la fin, Python ne se doutant nullement qu'il s'agit d'un « o » stylisé.

Retenez bien cette utilisation de `in` dans une condition. On cherche à savoir si un élément quelconque est contenu dans un ensemble donné (ici, si la lettre est contenue dans « AEIOUYaeiouy », c'est-à-dire si `lettre` est une voyelle). On retrouvera plus loin cette fonctionnalité.

En Python, on préfère utiliser une boucle `for` dans la grande majorité des cas. La boucle `while` que nous avons pourtant vue en premier, avec le plus d'exemples concrets, est en réalité assez peu utilisée en Python sauf cas particuliers.

Revenons à notre exemple de table de multiplication : on accomplit la même action dix fois... en incrémentant un nombre. C'est un cas tellement courant qu'il existe une fonction, `range`, qui permet de faire cela avec la boucle `for`, plutôt que la boucle `while`.

```
nb = 7 # On garde la variable contenant le nombre dont on veut la table de multiplication
for i in range(1, 11): # tant que i est entre 1 et 11 non inclus
    print(f"{i} * {nb} = {i * nb}")
```

Le résultat est le même, mais la boucle est plus concise. Cette notation est encouragée en Python. Elle utilise la fonction `range` qui retourne, pour simplifier, une séquence de nombres entiers entre deux bornes (ici, entre 1 et 11 non inclus). Il n'y a à proprement parler pas d'incrémentation : la variable `i` vaut `1` au premier tour de boucle, `2` au second tour de boucle... ainsi de suite jusqu'à `10`.

Si vous ne comprenez pas trop l'intérêt de la fonction `range` à ce stade, pas de panique. Sachez simplement qu'il est bien plus fréquent de trouver une instruction `for` en Python qu'une instruction `while`.

Un petit bonus : les mots-clés `break` et `continue`

Je vais ici vous montrer deux nouveaux mots-clés, `break` et `continue`. Vous ne les utiliserez peut-être pas beaucoup, mais vous devez au moins savoir qu'ils existent… et à quoi ils servent.

Le mot-clé `break`

Le mot-clé **`break`** permet tout simplement d'interrompre une boucle. Il est souvent utilisé dans une forme de boucle que je n'approuve pas trop :

```
while 1: # 1 est toujours vrai -> boucle infinie
    lettre = input("Tapez 'Q' pour quitter : ")
    if lettre == "Q":
        print("Fin de la boucle")
        break
```

La boucle `while` a pour condition `1`, c'est-à-dire une condition qui sera *toujours* vraie. Autrement dit, en regardant la ligne du `while`, on pense à une boucle infinie. En pratique, on demande à l'utilisateur de taper une lettre (un 'Q' pour quitter). Tant que l'utilisateur ne saisit pas cette lettre, le programme lui redemande de taper une lettre. Quand il tape 'Q', le programme affiche `Fin de la boucle` et la boucle s'arrête grâce au mot-clé `break`.

Ce mot-clé sert à arrêter une boucle quelle qu'en soit la condition. Python sort immédiatement de la boucle et exécute le code qui la suit, s'il y en a.

C'est un exemple un peu simpliste mais vous pouvez voir l'idée d'ensemble. Dans ce cas-là et, à mon sens, dans la plupart des cas où `break` est utilisé, on pourrait s'en sortir en précisant une véritable condition à la ligne du `while`. Par exemple, pourquoi ne pas créer un booléen qui sera **vrai** tout au long de la boucle et **faux** quand la boucle doit s'arrêter ? Ou bien tester directement si `lettre != « Q »` dans le `while` ?

Parfois, `break` est véritablement utile et fait gagner du temps. Pour autant, ne l'utilisez pas à outrance ; préférez une boucle avec une condition claire plutôt qu'un bloc d'instructions avec un `break`, qui sera plus dur à appréhender d'un seul coup d'œil.

Le mot-clé `continue`

Le mot-clé `continue` permet de… continuer une boucle, en repartant directement à la ligne du `while` ou du `for`. Un petit exemple s'impose :

```
i = 1
while i < 20: # Tant que i est inférieure à 20
    if i % 3 == 0:
        i += 4 # On ajoute 4 à i
        print("On incrémente i de 4. i vaut maintenant", i)
        continue # On retourne au while sans exécuter les
         ↪  autres lignes
```

```
7       print("La variable i =", i)
8       i += 1 # Dans le cas classique on ajoute juste 1 à i
```

Voici le résultat :

```
1  La variable i = 1
2  La variable i = 2
3  On incrémente i de 4. i vaut maintenant 7
4  La variable i = 7
5  La variable i = 8
6  On incrémente i de 4. i vaut maintenant 13
7  La variable i = 13
8  La variable i = 14
9  On incrémente i de 4. i vaut maintenant 19
10 La variable i = 19
```

Comme vous le voyez, tous les trois tours de boucle, `i` s'incrémente de 4. Arrivé au mot-clé `continue`, Python n'exécute pas la fin du bloc mais revient au début de la boucle en testant à nouveau la condition du `while`. Autrement dit, quand Python arrive à la ligne 6, il saute à la 2 sans exécuter les lignes 7 et 8. Au nouveau tour de boucle, Python reprend l'exécution normale (`continue` n'ignore la fin du bloc que pour le tour courant).

Mon exemple ne démontre pas de manière éclatante l'utilité de `continue`. Les rares fois où j'utilise ce mot-clé, c'est par exemple pour supprimer des éléments d'une liste, mais nous n'avons pas encore vu les listes. L'essentiel, pour l'instant, c'est que vous vous souveniez de ces deux mots-clés et que vous sachiez ce qu'ils font, si vous les rencontrez au détour d'une instruction.

En résumé

— Une boucle sert à répéter une portion de code en fonction d'un prédicat.
— On peut créer une boucle grâce au mot-clé `while` suivi d'un prédicat.
— On peut parcourir une séquence grâce à la syntaxe `for element in sequence:`.

Chapitre 6

Pas à pas vers la modularité (1/2)

Difficulté :

En programmation, on est souvent amené à utiliser plusieurs fois des groupes d'instructions dans un but très précis. Attention, je ne parle pas ici de boucles. Simplement, vous pourrez vous rendre compte que la plupart de nos tests pourront être regroupés dans des blocs plus vastes, fonctions ou modules. Je vais détailler tranquillement ces deux concepts.

Les fonctions regroupent plusieurs instructions dans un bloc qui sera appelé grâce à un nom. D'ailleurs, vous avez déjà vu des fonctions : `print` et `input` en font partie par exemple.

Les modules regroupent plusieurs fonctions selon le même principe. Toutes les fonctions mathématiques, par exemple, peuvent être placées dans un module dédié.

Les fonctions : à vous de jouer

Nous avons utilisé de nombreuses fonctions depuis le début de ce cours. On citera pour mémoire `print`, `type` et `input`, sans compter quelques autres. Cependant, vous devez bien vous rendre compte qu'il en existe un nombre incalculable déjà construites en Python. Toutefois, vous vous apercevrez aussi que, très souvent, un programmeur crée ses propres fonctions. C'est le premier pas que vous ferez, dans ce chapitre, vers la **modularité**. Ce terme un peu barbare signifie que nous allons nous habituer à regrouper dans des fonctions des parties de notre code que nous serons amenés à réutiliser. Au prochain chapitre, nous apprendrons à regrouper nos fonctions ayant un rapport entre elles dans un fichier, pour constituer un module, mais n'anticipons pas.

La création de fonctions

Nous allons, pour illustrer cet exemple, reprendre le code de la table de multiplication, que nous avons vu au chapitre précédent et qui, décidément, n'en finit pas de vous poursuivre.

Nous allons emprisonner notre code calculant la table de multiplication par 7 dans une fonction que nous appellerons `table_par_7`.

On crée une fonction selon le schéma suivant :

```
1 def nom_de_la_fonction(paramètre1, paramètre2, paramètre3,
  ↪ paramètreN):
2     # Bloc d'instructions
```

Les blocs d'instructions nous courent après aussi, quel enfer ! Si l'on décortique la ligne de définition de la fonction, on trouve dans l'ordre :

- `def`, mot-clé qui est l'abréviation de « define » (définir, en anglais) et qui constitue le prélude à toute construction de fonction.
- Le nom de la fonction, qui se nomme exactement comme une variable (nous verrons par la suite que ce n'est pas par hasard). N'utilisez pas un nom de variable déjà instanciée pour nommer une fonction.
- La liste des paramètres qui seront fournis lors d'un appel à la fonction. Les paramètres sont séparés par des virgules et la liste est encadrée par des parenthèses (les espaces sont optionnels mais améliorent la lisibilité).
- Les deux points, encore et toujours, qui clôturent la ligne.

Les parenthèses sont obligatoires, quand bien même votre fonction n'attendrait aucun paramètre.

Le code pour mettre notre table de multiplication par 7 dans une fonction serait donc :

```
1 def table_par_7():
2     nb = 7
```

```
    for i in range(1, 11): # tant que i est entre 1 et 11 non
    ↪ inclus
        print(f"{i} * {nb} = {i * nb}")
```

Quand vous exécutez ce code à l'écran, il ne se passe rien. Une fois que vous avez retrouvé les trois chevrons, essayez d'appeler la fonction :

```
>>> table_par_7()
1 * 7 = 7
2 * 7 = 14
3 * 7 = 21
4 * 7 = 28
5 * 7 = 35
6 * 7 = 42
7 * 7 = 49
8 * 7 = 56
9 * 7 = 63
10 * 7 = 70
>>>
```

Bien, c'est, euh, exactement ce qu'on avait réussi à faire au chapitre précédent et l'intérêt ne saute pas encore aux yeux. L'avantage est que l'on peut appeler facilement la fonction et réafficher toute la table sans avoir besoin de tout réécrire !

Et si on saisissait des paramètres pour afficher la table de 5 ou de 8... ?

Oui, ce serait déjà bien plus utile. Je ne pense pas que vous ayez trop de mal à trouver le code de la fonction :

```
def table(nb):
    for i in range(1, 11): # tant que i est entre 1 et 11 non
    ↪ inclus
        print(f"{i} * {nb} = {i * nb}")
```

Et là, vous pouvez passer en argument différents nombres, `table(8)` pour afficher la table de multiplication par 8 par exemple.

On peut aussi envisager de passer en paramètre le nombre de valeurs à afficher dans la table.

```
def table(nb, maximum):
    for i in range(1, maximum + 1): # tant que i est entre 1
    ↪ et maximum inclus
        print(f"{i} * {nb} = {i * nb}")
```

Si vous tapez à présent `table(11, 20)`, l'interpréteur vous affichera la table de 11, de 1*11 à 20*11. Magique non ?

Dans le cas où l'on utilise plusieurs paramètres sans les nommer, comme ici, il faut respecter l'ordre d'appel des paramètres, cela va de soi. Si vous commencez à mettre le nombre d'affichages en premier paramètre alors que, dans la définition, c'était le second, vous risquez d'avoir quelques surprises. Il est possible d'appeler les paramètres dans le désordre mais il faut, dans ce cas, préciser leur nom : nous verrons cela plus loin.

Si vous fournissez en second paramètre un nombre négatif, vous n'allez voir aucune table s'afficher à l'écran. C'est peu souhaitable (mieux vaut indiquer clairement les erreurs qui surviennent). En Python, on préférera mettre un commentaire en tête de fonction ou une `docstring`, comme on le verra ultérieurement, pour indiquer que `maximum` doit être positif, plutôt que de faire des vérifications qui au final feront perdre du temps. Une des phrases reflétant la philosophie du langage et qui peut s'appliquer à ce type de situation est « *we're all consenting adults here* » [1] (sous-entendu, quelques avertissements en commentaires sont plus efficaces qu'une restriction au niveau du code). On aura l'occasion de retrouver cette phrase plus loin, surtout quand on parlera des objets.

Valeurs par défaut des paramètres

Il est également possible de préciser une valeur par défaut pour les paramètres de la fonction. Vous pouvez par exemple indiquer que le nombre maximum d'affichages doit être de 10 par défaut (c'est-à-dire si l'utilisateur de votre fonction ne le précise pas). Cela se fait le plus simplement du monde :

```
1  def table(nb, maximum=10):
2      """Fonction affichant la table de multiplication par nb.
3
4      Arguments :
5          nb (int) : le nombre dont la table de multiplication
   ↪   est à afficher.
6          maximum (int, optionnel) : afficher la table de 1 à
   ↪   maximum.
7
8      """
9      for i in range(1, maximum + 1): # tant que i est entre 1
   ↪      et maximum inclus
10         print(f"{i} * {nb} = {i * nb}")
```

Il suffit d'ajouter `=10` après `maximum`. À présent, vous pouvez appeler la fonction de deux façons : soit en précisant le numéro de la table et le nombre maximum d'affichages, soit en ne précisant que le numéro de la table (`table(7)`). Dans ce dernier cas, `maximum` vaudra 10 par défaut.

J'en ai profité pour ajouter quelques lignes d'explications que vous aurez sans doute remarquées. Nous avons placé une chaîne de caractères, sans la capturer dans une

1. « Nous sommes entre adultes consentants ».

variable, juste en-dessous de la définition de la fonction. Cette chaîne est ce qu'on appelle une `docstring` que l'on pourrait traduire par une chaîne d'aide. Si vous tapez `help(table)`, c'est ce message que vous verrez apparaître. Documenter vos fonctions est également une bonne habitude à prendre. Comme vous le voyez, on indente cette chaîne et on la met entre triple guillemets. Si la chaîne figure sur une seule ligne, on pourra mettre les trois guillemets fermants sur la même ligne ; sinon, on préférera sauter une ligne avant de fermer cette chaîne, pour des raisons de lisibilité. Tout le texte d'aide est indenté au même niveau que le code de la fonction.

Enfin, sachez que l'on peut appeler des paramètres par leur nom. Cela est utile pour une fonction comptant un certain nombre de paramètres qui ont tous une valeur par défaut. Vous pouvez aussi utiliser cette méthode sur une fonction sans paramètre par défaut, mais c'est moins courant.

Prenons un exemple de définition de fonction :

```
1 def fonc(a=1, b=2, c=3, d=4, e=5):
2     print(f"a = {a} b = {b} c = {c} d = {d} e = {e}")
```

Simple, n'est-ce pas ? Eh bien, vous avez de nombreuses façons d'appeler cette fonction. En voici quelques exemples :

Instruction	Résultat
`fonc()`	a = 1 b = 2 c = 3 d = 4 e = 5
`fonc(4)`	a = 4 b = 2 c = 3 d = 4 e = 5
`fonc(b=8, d=5)`	a = 1 b = 8 c = 3 d = 5 e = 5
`fonc(b=35, c=48, a=4, e=9)`	a = 4 b = 35 c = 48 d = 4 e = 9

Je ne pense pas que des explications supplémentaires s'imposent. Si vous voulez changer la valeur d'un paramètre, vous tapez son nom, suivi d'un signe égal puis d'une valeur (qui peut être une variable bien entendu). Peu importent les paramètres que vous précisez (comme vous le voyez dans cet exemple où tous les paramètres ont une valeur par défaut, vous pouvez appeler la fonction sans paramètre), peu importe l'ordre d'appel des paramètres.

Signature d'une fonction

On entend par « signature de fonction » les éléments qui permettent au langage d'identifier ladite fonction. En C++, par exemple, la signature d'une fonction est constituée de son nom et du type de chacun de ses paramètres. Cela veut dire que l'on peut trouver plusieurs fonctions portant le même nom mais dont les paramètres diffèrent. Au moment de l'appel de fonction, le compilateur recherche celle qui correspond à cette signature.

En Python, comme vous avez pu le voir, on ne précise pas les types des paramètres. Dans ce langage, la signature d'une fonction est tout simplement son nom. Cela signifie que vous ne devez pas définir deux fonctions du même nom (si vous le faites, l'ancienne définition est écrasée par la nouvelle).

```
1  def exemple():
2      print("Un exemple de fonction sans paramètre")
3
4  exemple()
5  Un exemple de fonction sans paramètre
6
7  def exemple(): # On redéfinit la fonction exemple
8      print("Un autre exemple de fonction sans paramètre")
9
10 exemple()
11 Un autre exemple de fonction sans paramètre
```

À la ligne 1, on définit la fonction `exemple`. On l'appelle une première fois à la ligne 4. On la redéfinit à la ligne 7. L'ancienne définition est écrasée et l'ancienne fonction ne pourra plus être appelée.

Retenez simplement que, comme pour les variables, un nom de fonction ne renvoie que vers une fonction unique, on ne peut surcharger de fonctions en Python.

Depuis Python 3.5, il est possible de spécifier les types d'arguments d'une fonction. La syntaxe est assez claire, mais elle n'est pas présentée ici pour éviter de surcharger un chapitre déjà assez long. Notez que, même si des annotations de type sont utilisées, Python ne fait aucune vérification quant au type utilisé lors de l'appel/ ; rien ne change à ce niveau, c'est simplement plus clair pour l'utilisateur de la fonction.

L'instruction `return`

Nous n'avons pas encore fait le tour des possibilités de la fonction. Et d'ailleurs, même à la fin de ce chapitre, il nous restera quelques petites fonctionnalités à voir. Si vous vous souvenez bien, il existe des fonctions comme `print` qui ne renvoient rien (attention, « renvoyer » et « afficher » sont deux choses différentes) et des fonctions telles que `input` ou `type` qui renvoient une valeur. Vous pouvez capturer cette valeur dans une variable (exemple `variable2 = type(variable1)`). En effet, les fonctions travaillent en général sur des données et renvoient le résultat obtenu, suite à un calcul par exemple.

Prenons un exemple simple : une fonction chargée de mettre au carré une valeur passée en argument. Je vous signale au passage que Python en est parfaitement capable sans avoir à coder une nouvelle fonction, mais c'est pour l'exemple.

```
1  def carré(valeur):
2      return valeur * valeur
```

L'instruction `return` signifie qu'on va **renvoyer** la valeur, pour la récupérer ensuite et la stocker dans une variable par exemple. Cette instruction arrête le déroulement de la fonction ; le code situé après le `return` ne s'exécutera pas.

```
1  variable = carré(5)
```

`variable` contiendra, après exécution de cette instruction, 5 au carré, c'est-à-dire 25.

Sachez que l'on peut renvoyer plusieurs valeurs que l'on sépare par des virgules et que l'on peut capturer dans des variables également séparées par des virgules, mais je m'attarderai plus loin sur cette particularité. Retenez simplement la définition d'une fonction, les paramètres, les valeurs par défaut, l'instruction `return` et ce sera déjà bien.

Les annotations de type

Ai-je dit qu'on ne précisait pas les types des arguments de nos fonctions en Python ? Oui, j'ai dit ça. C'est plutôt vrai dans l'ensemble. Toutefois, depuis Python 3.5, on peut préciser le type des arguments et le type de retour des fonctions.

À quoi cela sert-il, si c'est optionnel ?

La réponse la plus courte : en programmation, beaucoup de choses optionnelles sont plutôt utiles. Voyez les commentaires ! Dans ce cas précis, les annotations de type ont été imaginées comme purement optionnelles. Aujourd'hui, des bibliothèques comme Pydantic ou Mypy les utilisent. Il n'est pas nécessaire de savoir s'en servir, mais c'est une fonctionnalité intéressante qui semble gagner progressivement du terrain parmi les développeurs.

L'idée est assez simple : pour chaque paramètre de la fonction et pour la ou les valeur(s) de retour, on va indiquer le type de l'information correspondante. Par type, ici, j'entends `int`, `float`, `str` ou autre.

Côté syntaxe, après le nom du paramètre, on trouve le signe deux points, un espace (pour la lisibilité) et le type de ce paramètre. Revenons à notre fonction `table` pour l'exemple. Je ne garde que le premier paramètre, `nb`, pour éviter de créer la confusion :

```
1 def table(nb: int):
```

Ici, on précise que le paramètre `nb` est de type `int` (entier).

Qu'est-ce que ça change, concrètement ?

Éh bien... rien du tout ! Enfin presque rien.

```
1 >>> table(7)
2 1 * 7 = 7
3 2 * 7 = 14
4 3 * 7 = 21
5 4 * 7 = 28
6 5 * 7 = 35
```

```
6 * 7 = 42
7 * 7 = 49
8 * 7 = 56
9 * 7 = 63
10 * 7 = 70
>>>
```

Rien de bien extraordinaire. On peut appeler notre fonction comme avant. Et si on essaye de lui passer autre chose...

```
>>> table("non !")
1 * non ! = non !
2 * non ! = non !non !
3 * non ! = non !non !non !
4 * non ! = non !non !non !non !
5 * non ! = non !non !non !non !non !
6 * non ! = non !non !non !non !non !non !
7 * non ! = non !non !non !non !non !non !non !
8 * non ! = non !non !non !non !non !non !non !non !
9 * non ! = non !non !non !non !non !non !non !non !non !
10 * non ! = non !non !non !non !non !non !non !non !non !non
 ↪  !
>>>
```

Déception. Python semble indifférent à la beauté du code. Je croyais qu'il m'aimait bien.

Pourtant ce résultat est magnifique ! Il est... inattendu, certes, mais le code fonctionne ! Et... il ne devrait sans doute pas.

Pourquoi ce résultat ? Vous l'avez sans doute deviné, c'est une fonctionnalité du signe de multiplication, quand il multiplie un nombre par une chaîne de caractères. C'est très pratique dans certaines situations, mais ce n'est pas ce qui nous intéresse ici.

Python n'a évidemment pas empêché d'exécuter notre fonction : on lui avait dit que notre paramètre `nb` était un entier. On lui a donné une chaîne de caractères. Il l'accepte. Pourquoi ?

Python accepte les annotations de type. Cela ne veut pas dire qu'il y fait particulièrement attention. Trop de vérifications altéreraient ses performances... et Python s'imagine, avec raison, que, pour la plupart, les programmeurs accepteront les annotations de type et n'essayeront pas de faire du code d'appel avec des types invalides. En attendant, ces annotations de type ont bel et bien un intérêt, comme nous le verrons plus loin : bien qu'elles n'empêchent pas d'appeler des fonctions avec des types non supportés, elles permettent à d'autres outils (comme les IDE) de vérifier la cohérence de votre code avant de le publier. Et ce n'est pas un mince avantage.

En revanche, pour un programmeur, il est très utile de savoir à quel type appartient un paramètre. Voici notre fonction `table` avec deux paramètres, dont un facultatif :

```
def table(nb: int, maximum: int = 10):
```

Quand on précise un paramètre avec une valeur par défaut, on indique d'abord son type (nom de la variable, le signe deux points, un espace, le nom du type) suivi de la valeur par défaut. Par convention, on évite de coller le signe égal au nom du type, on met des espaces autour du signe égal ce qui évite de penser que la valeur par défaut (10, ici) s'applique au type (`int` ici). Ce n'est qu'une convention, mais elle se retrouve assez souvent.

Maintenant voyons notre fonction `carré` :

```
1 def carré(valeur):
2     return valeur * valeur
```

On pourrait commencer par ajouter une annotation de type sur le paramètre `valeur`.

```
1 def carré(valeur: int):
2     return valeur * valeur
```

Mais comment indiquer le type de retour ? Il existe une syntaxe facultative qui consiste à ajouter, entre la parenthèse fermante et le signe deux-points, une flèche (constituée d'un tiret et du signe supérieur) suivie du type de retour :

```
1 def carré(valeur: int) -> int:
2     return valeur * valeur
```

Cela paraît assez technique de prime abord, mais avec l'habitude, c'est très lisible. On sait que la fonction `carré` prend un entier en paramètre et retourne un entier. Même sans lire la documentation (que je vous conseille de toujours écrire), un programmeur peut en un coup d'œil comprendre les arguments nécessaires pour utiliser votre fonction.

Néanmoins... il y a un léger problème avec notre fonction `carré`. Ne peut-elle prendre que des entiers ?

```
1 >>> carré(2.5)
2 6.25
3 >>>
```

On peut aussi lui passer des nombres flottants. C'est une option attendue. Depuis Python 3.10, on peut préciser plusieurs types possibles en les séparant par une barre verticale :

```
1 def carré(valeur: int | float) -> int | float:
2     return valeur * valeur
```

Notre fonction indique maintenant qu'elle prend un paramètre qui peut être soit un entier, soit un nombre flottant. Elle retourne soit un entier, soit un nombre flottant.

Il faut un peu de temps pour s'habituer à cette syntaxe. Quand j'ai vu mes premières annotations de type, je me suis demandé si l'exemple était bien en Python et pas dans un autre langage. Pourtant elles sont bien plus lisibles qu'à leur sortie. Je ne vais pas écrire des annotations de type à chaque fonction... mais j'en préciserai pour que vous puissiez vous habituer à cette syntaxe. Si cette dernière vous déconcerte trop, n'oubliez pas qu'elle est facultative, comme un commentaire sur une variable.

Les fonctions `lambda`

Nous venons de voir comment créer une fonction grâce au mot-clé `def`. Python nous propose un autre moyen de créer des fonctions ; des fonctions extrêmement courtes car limitées à une seule instruction.

Pourquoi une autre façon de créer des fonctions ? La première suffit, non ?

Disons que ce n'est pas tout à fait la même chose, comme vous allez le voir. Les fonctions lambda sont en général utilisées dans un certain contexte, pour lequel définir une fonction à l'aide de `def` serait plus long et moins pratique.

Syntaxe

Avant tout, voyons la syntaxe d'une définition de fonction `lambda`. Nous allons utiliser le mot-clé `lambda` comme ceci : `lambda arg1, arg2,... : instruction de retour`.

Je pense qu'un exemple vous semblera plus clair. On veut créer une fonction qui prend un paramètre et renvoie ce paramètre au carré.

```
>>> lambda x: x * x
<function <lambda> at 0x00BA1B70>
>>>
```

D'abord, on a le mot-clé `lambda` suivi de la liste des arguments, séparés par des virgules. Ici, il n'y en a qu'un seul ; c'est `x`. Ensuite figure un nouveau signe deux-points et l'instruction de la `lambda` ; c'est son résultat qui sera renvoyé par la fonction. Dans notre exemple, on renvoie donc `x * x`.

Comment fait-on pour appeler notre `lambda` ?

On a bien créé une fonction `lambda` mais on ne dispose ici d'aucun moyen pour l'appeler. Vous pouvez tout simplement stocker votre fonction `lambda` nouvellement définie dans une variable, par une simple affectation :

```
>>> f = lambda x: x * x
>>> f(5)
25
>>> f(-18)
324
>>>
```

Un autre exemple : si vous voulez créer une fonction `lambda` prenant deux paramètres et renvoyant la somme de ces deux paramètres, la syntaxe sera la suivante :

```
lambda x, y: x + y
```

Utilisation

À notre niveau, les fonctions `lambda` sont plus une curiosité que véritablement utiles. Je vous les présente maintenant parce que le contexte s'y prête et que vous pourriez en rencontrer certaines sans comprendre ce que c'est.

Il vous faudra cependant attendre un peu pour que je vous montre une réelle application des `lambda`. En attendant, n'oubliez pas ce mot-clé et la syntaxe qui va avec... on passe à la suite !

À la découverte des modules

Jusqu'ici, nous avons travaillé avec les fonctions de Python chargées au lancement de l'interpréteur. Il y en a déjà un certain nombre et nous pourrions continuer et finir cette première partie sans utiliser de module Python... ou presque. Pourtant, il faut bien qu'à un moment, je vous montre cette possibilité des plus intéressantes !

Les modules, qu'est-ce que c'est ?

Un module est grossièrement un bout de code que l'on a enfermé dans un fichier. On emprisonne ainsi des fonctions et des variables ayant toutes un rapport entre elles. Ainsi, si l'on veut travailler avec les fonctionnalités contenues dans le module, il n'y a qu'à **importer** celui-ci et utiliser ensuite toutes les fonctions et variables prévues.

Il existe un grand nombre de modules disponibles avec Python sans qu'il soit nécessaire d'installer des bibliothèques supplémentaires. Pour cette partie, nous prendrons l'exemple du module `math` qui contient, comme son nom l'indique, des fonctions mathématiques. Inutile de vous inquiéter, nous n'allons pas nous attarder sur le module lui-même pour coder une calculatrice scientifique ; nous verrons surtout les différentes méthodes d'importation.

La méthode `import`

Lorsque vous ouvrez l'interpréteur Python, les fonctionnalités du module `math` ne sont pas incluses. Il s'agit en effet d'un module ; il vous appartient de l'importer si vous vous dites « tiens, mon programme risque d'avoir besoin de fonctions mathématiques ». Nous allons voir une première syntaxe d'importation.

```
>>> import math
>>>
```

La syntaxe est facile à retenir : le mot-clé `import`, qui signifie « importer » en anglais, suivi du nom du module, ici `math`.

Après l'exécution de cette instruction, rien ne se passe... en apparence. En réalité, Python vient d'importer le module `math`. Toutes les fonctions mathématiques qu'il contient sont maintenant accessibles. Pour appeler l'une d'entre elles, il faut taper le nom du module suivi d'un point « . » puis du nom de la fonction. C'est la même syntaxe pour appeler des variables du module. Voyons un exemple :

```
>>> math.sqrt(16)
4.0
>>>
```

Comme vous le voyez, la fonction `sqrt` du module `math` renvoie la racine carrée du nombre passé en paramètre.

Comment suis-je censé savoir quelles fonctions existent et ce que fait `math.sqrt` dans ce cas précis ?

J'aurais dû vous montrer cette fonction bien plus tôt car, oui, c'est une fonction qui va nous donner la solution. Il s'agit de `help`, qui prend en argument la fonction ou le module sur lequel vous demandez de l'aide. L'aide est fournie en anglais mais c'est de l'anglais technique, c'est-à-dire une forme que vous devrez maîtriser pour programmer, si ce n'est pas déjà le cas. Une grande majorité de la documentation est en anglais, bien que vous puissiez maintenant en trouver une bonne part en français.

```
>>> help(math)
Help on built-in module math:

NAME
  math

DESCRIPTION
    This module provides access to the mathematical functions
    defined by the C standard.

FUNCTIONS
    acos(x, /)
        Return the arc cosine (measured in radians) of x.

        The result is between 0 and pi.

    acosh(x, /)
        Return the inverse hyperbolic cosine of x.

    asin(x, /)
        Return the arc sine (measured in radians) of x.
```

```
22
23          The result is between -pi/2 and pi/2.
24
25      asinh(x, /)
26          Return the inverse hyperbolic sine of x.
27
28      atan(x, /)
29          Return the arc tangent (measured in radians) of x.
```

Si vous parlez un minimum l'anglais, vous avez accès à une description exhaustive des fonctions du module `math`. Vous voyez en haut de la page le nom du module, le fichier qui l'héberge, puis la description du module. Ensuite se trouve une liste des fonctions, chacune étant accompagnée d'une courte description.

Tapez `Q` pour revenir à la fenêtre d'interpréteur, `Espace` pour avancer d'une page, `Entrée` pour avancer d'une ligne. Vous pouvez également passer un nom de fonction en paramètre de la fonction `help`.

```
1  >>> help(math.sqrt)
2  Help on built-in function sqrt in module math:
3
4  sqrt(x, /)
5      Return the square root of x.
6  >>>
```

Ne mettez pas les parenthèses habituelles après le nom de la fonction. C'est en réalité la référence de la fonction que vous envoyez à `help`. Si vous ajoutez les parenthèses après le nom, vous devrez préciser une valeur. Dans ce cas, c'est la valeur renvoyée par `math.sqrt` qui sera analysée, soit un nombre (entier ou flottant).

Nous reviendrons plus tard sur le concept des références des fonctions. Si vous avez compris pourquoi il ne fallait pas mettre de parenthèses après le nom de la fonction dans `help`, tant mieux. Sinon, ce n'est pas grave, nous y reviendrons en temps voulu.

Depuis Python 3.7, la documentation officielle est disponible et maintenue à jour dans différentes langues. La documentation officielle et en français du module `math` se trouve ici : docs.python.org/fr/3/library/math.html .

Utiliser un espace de noms spécifique

En vérité, quand vous tapez `import math`, cela crée un espace de noms dénommé « math », contenant les variables et fonctions du module `math`. Quand vous tapez `math.sqrt(25)`, vous précisez à Python que vous souhaitez exécuter la fonction `sqrt` contenue dans l'espace de noms `math`. Cela signifie que vous pouvez avoir, dans

l'espace de noms principal, une autre fonction `sqrt` que vous avez définie vous-mêmes. Il n'y aura pas de conflit entre, d'une part, la fonction que vous avez créée et que vous appellerez grâce à l'instruction `sqrt` et, d'autre part, la fonction `sqrt` du module `math` que vous appellerez grâce à l'instruction `math.sqrt`.

Concrètement, qu'est-ce qu'un espace de noms ?

Il s'agit de regrouper certaines fonctions et variables sous un préfixe spécifique. Prenons un exemple concret :

```
1 import math
2 a = 5
3 b = 33.2
```

Dans l'espace de noms principal, celui qui ne nécessite pas de préfixe et que vous utilisez depuis le début de ce cours, on trouve :

- La variable `a`.
- La variable `b`.
- Le module `math`, qui se trouve dans un espace de noms s'appelant `math` également. Dans cet espace de noms, on trouve :
  - la fonction `sqrt` ;
  - la variable `pi` ;
  - et bien d'autres fonctions et variables...

C'est aussi l'intérêt des modules : des variables et fonctions sont stockées à part, bien à l'abri dans un espace de noms, sans risque de conflit avec vos propres variables et fonctions. Dans certains cas toutefois, vous voudrez peut-être changer le nom de l'espace de noms dans lequel sera stocké le module importé.

```
1 import math as mathematiques
2 mathematiques.sqrt(25)
```

Qu'est-ce qu'on a fait là ?

On a simplement importé le module `math` en spécifiant à Python de l'héberger dans l'espace de noms dénommé « mathematiques » au lieu de `math`. Cela permet de mieux contrôler les espaces de noms des modules que vous importerez. Dans la plupart des cas, vous n'utiliserez pas cette fonctionnalité mais, au moins, vous savez qu'elle existe. Quand on se penchera sur les *packages*, vous vous souviendrez probablement de cette possibilité.

Une autre méthode d'importation : `from ... import ...`

Il existe une autre méthode d'importation qui ne fonctionne pas tout à fait de la même façon. En fonction du résultat attendu, j'utilise indifféremment l'une ou l'autre de

ces méthodes. Reprenons notre exemple du module `math`. Admettons que nous ayons uniquement besoin, dans notre programme, de la fonction renvoyant la valeur absolue d'une variable. Dans ce cas, nous n'allons importer que la fonction, au lieu d'importer tout le module.

```
>>> from math import fabs
>>> fabs(-5)
5.0
>>> fabs(2)
2.0
>>>
```

Pour ceux qui n'ont pas encore étudié les valeurs absolues, il s'agit tout simplement de l'opposé de la variable si elle est négative et de la variable elle-même si elle est positive. Une valeur absolue est ainsi toujours positive.

Vous aurez remarqué qu'on ne met plus le préfixe `math.` devant le nom de la fonction. En effet, nous l'avons importée avec la méthode `from` : celle-ci charge la fonction depuis le module indiqué et la place dans l'interpréteur au même plan que les fonctions existantes, comme `print`. Si vous avez compris les explications sur les espaces de noms, vous voyez que `print` et `fabs` sont dans le même espace de noms (principal).

Vous pouvez appeler toutes les variables et fonctions d'un module en tapant « * » à la place du nom de la fonction à importer.

```
>>> from math import *
>>> sqrt(4)
2.0
>>> fabs(5)
5.0
```

À la ligne 1 de notre programme, l'interpréteur a parcouru toutes les fonctions et variables du module `math` et les a importées directement dans l'espace de noms principal sans les emprisonner dans l'espace de noms `math`.

Sachez que la syntaxe `from ... import *` n'est pas précisément bien vue des développeurs Python. Elle a plusieurs inconvénients, dont celui de tout mettre dans l'espace de nom principal et écraser sans distinction. Mieux vaut n'importer que ce dont on a besoin.

Bilan

Quelle méthode faut-il utiliser ?

Vaste question ! Je dirais que c'est à vous de voir. La seconde méthode a l'avantage inestimable d'économiser la saisie systématique du nom du module en préfixe de chaque

fonction. L'inconvénient de cette méthode apparaît si l'on utilise plusieurs modules de cette manière : si par hasard il existe dans deux modules différents deux fonctions portant le même nom, l'interpréteur ne conservera que la dernière appelée [2]. Conclusion... c'est à vous de voir en fonction de vos besoins !

En résumé

— Une fonction est une portion de code contenant des instructions, que l'on va pouvoir réutiliser facilement.
— Découper son programme en fonctions permet une meilleure organisation.
— Les fonctions peuvent recevoir des informations en entrée et renvoyer une information grâce au mot-clé `return`.
— Les fonctions se définissent de la façon suivante : `def nom_fonction(parametre1, parametre2, parametreN):`

2. Je vous rappelle qu'il ne peut y avoir deux variables ou fonctions portant le même nom.

Chapitre 7

Pas à pas vers la modularité (2/2)

Difficulté :

Nous allons commencer par voir comment mettre nos programmes en boîte... ou plutôt en fichier. Je vais faire d'une pierre deux coups : d'abord, c'est agréable d'avoir son code dans un fichier modifiable à souhait, surtout qu'on commence à savoir écrire des programmes assez sympathiques (même si vous n'en avez peut-être pas l'impression). Ensuite, c'est un prélude nécessaire à la création de modules.

Comme vous allez le voir, nos programmes Python peuvent être mis dans des fichiers pour être exécutés ultérieurement. De ce fait, vous avez déjà pratiquement toutes les clés pour créer un programme Python exécutable. Le même mécanisme est utilisé pour la création de modules. Les modules sont eux aussi des fichiers contenant du code Python.

Enfin, nous verrons à la fin de ce chapitre comment créer des *packages* pour regrouper nos modules ayant un rapport entre eux.

C'est parti !

Mettre notre code en boîte

Fini, l'interpréteur ?

Je le répète encore, l'interpréteur est véritablement très pratique pour un grand nombre de raisons. Et la meilleure d'entre elles est qu'il propose une manière interactive d'écrire un programme, qui permet de tester le résultat de chaque instruction. Toutefois, l'interpréteur a aussi un défaut : le code que vous saisissez est effacé à la fermeture de la fenêtre. Or, nous commençons à être capables de rédiger des programmes relativement complexes. Dans ces conditions, devoir réécrire le code entier de son programme à chaque fois qu'on ouvre l'interpréteur de commandes est assez lourd.

La solution ? Conserver notre code dans un fichier que nous exécuterons à volonté, comme un véritable programme !

Comme je l'ai dit au début de ce chapitre, il est grand temps que je vous présente cette possibilité. On ne dit pas adieu à l'interpréteur de commandes pour autant, juste au revoir pour cette fois… on le retrouvera bien assez tôt, la possibilité de tester le code à la volée étant vraiment un atout pour apprendre le langage.

Emprisonnons notre programme dans un fichier

Pour cette démonstration, je reprendrai le code optimisé du programme calculant si une année est bissextile. C'est un petit programme dont l'utilité est certes discutable, mais il suffit pour un premier essai.

Je vous rappelle le code ici pour que nous travaillions tous sur les mêmes lignes (votre version fonctionnera également sans problème dans un fichier, si elle tournait sous l'interpréteur de commandes).

```
1  # Programme testant si une année, saisie par l'utilisateur,
   ↪  est bissextile ou non
2
3  année = input("Saisissez une année : ") # On attend que
   ↪  l'utilisateur fournisse l'année qu'il désire tester
4  année = int(année) # Risque d'erreur si l'utilisateur n'a pas
   ↪  saisi un nombre
5
6  if année % 400 == 0 or (année % 4 == 0 and année % 100 != 0):
7      print("L'année saisie est bissextile.")
8  else:
9      print("L'année saisie n'est pas bissextile.")
```

▷ Copier ce code
Code web : 886842

C'est à votre tour de travailler maintenant. Je vais vous donner des pistes, mais je ne vais pas me mettre à votre place ; chacun prend ses habitudes en fonction de ses préférences.

Ouvrez un éditeur basique : sous Windows, le bloc-notes est candidat, Wordpad ou Word sont exclus ; sous Linux, vous pouvez utiliser Gedit, Vim ou Emacs. Insérez le code dans ce fichier et enregistrez-le avec l'extension `.py` (exemple `bissextile.py`), comme à la figure 7.1. Cela indiquera au système d'exploitation qu'il doit utiliser Python pour exécuter ce programme [1].

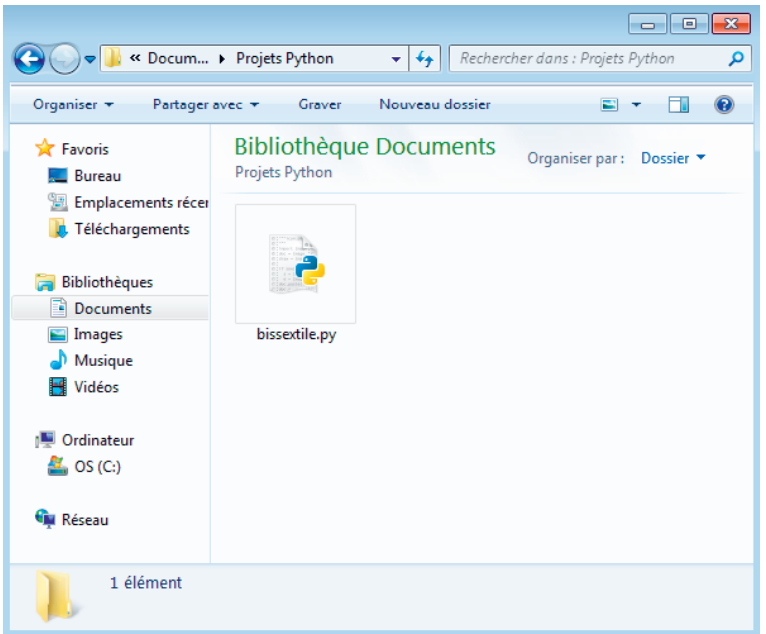


FIGURE 7.1 – Enregistrer un fichier Python sous Windows

Sous Linux

Vous devrez ajouter dans votre fichier une ligne, tout au début, spécifiant le chemin de l'interpréteur Python (si vous avez déjà rédigé des scripts, en Bash par exemple, cette méthode ne vous surprendra pas). La première ligne de votre programme sera :

```
1 #!chemin
```

Remplacez alors le terme `chemin` par le chemin donnant accès à l'interpréteur, par exemple : `/usr/local/bin/python3.10`. Vous devrez changer le droit d'exécution du fichier avant de l'exécuter comme un script.

Sous Windows

Les accents dans les programmes sont autorisés. Depuis la version 3, Python part du principe que nos fichiers sont tous encodés en UTF-8. Sous Linux, il y a de fortes chances

1. Cela est nécessaire sous Windows uniquement.

pour que ce soit vrai. Sous Windows, c'est rarement le cas ; il va donc falloir indiquer à Python que nous utilisons un autre encodage pour nos fichiers. Le temps manque pour une explication détaillée, alors je me contenterai de cette simplification : si vous êtes sous Windows et utilisez le bloc-notes classique, recopiez la ligne de code suivante tout en haut de chacun de vos programmes écrits en Python avant de l'exécuter. Si vous êtes sous Windows et utilisez un éditeur différent (comme Notepad++), vous aurez la possibilité d'écrire le code directement en UTF-8. Je vous le conseille. Dans ce cas, la ligne ci-après n'est pas nécessaire. Sinon, placez-la en haut de votre fichier :

```
# -*-coding:latin-1 -*
```

Si vous rencontrez des problèmes d'accents, de nombreuses ressources existent sur Internet qui vous expliqueront autant le problème soulevé par Python, que la solution qui marchera le mieux pour vous.

Enfin, toujours sous Windows, avant d'exécuter notre code, il faut y ajouter deux autres lignes (ne blâmez pas Python trop vite). Les deux lignes suivantes (l'une presque tout en haut du programme, l'autre tout en bas) sont nécessaires car Windows risque de refermer votre programme trop vite pour que vous en observiez le résultat. Elles forcent Python à se mettre en pause tant que l'utilisateur (vous, en l'occurrence) n'appuie sur aucune touche. Cette étape n'est pas nécessaire sous Linux :

```
import os # On importe le module os qui dispose de variables
          # et de fonctions utiles pour dialoguer avec votre
          # système d'exploitation

# ... placez votre programme ici AVANT les lignes suivantes

# On met le programme en pause pour éviter qu'il ne se referme
 ↪  (Windows)
os.system("pause")
```

En résumé, sous Windows, voici le contenu complet du fichier que vous devriez obtenir. La première ligne est nécessaire si vous utilisez le bloc-notes. Les seconde et dernière lignes sont nécessaires sous Windows :

```
# -*-coding:latin-1 -*
import os

# Programme testant si une année, saisie par l'utilisateur,
 ↪  est bissextile ou non

année = input("Saisissez une année : ") # On attend que
 ↪  l'utilisateur fournisse l'année qu'il désire tester
année = int(année) # Risque d'erreur si l'utilisateur n'a pas
 ↪  saisi un nombre

if année % 400 == 0 or (année % 4 == 0 and année % 100 != 0):
    print("L'année saisie est bissextile.")
else:
```

```
12      print("L'année saisie n'est pas bissextile.")
13
14  os.system("pause")
```

Ouf ! Maintenant, dans votre explorateur de fichiers, double-cliquez simplement sur le fichier (`bissextile.py` dans notre cas). Le programme devrait se lancer, vous demandant une année. Entrez-la en utilisant votre clavier. Puis le programme devrait vous dire si l'année est bissextile ou non et se mettre en pause. Vous devrez appuyer sur une touche du clavier, ce qui refermera votre programme.

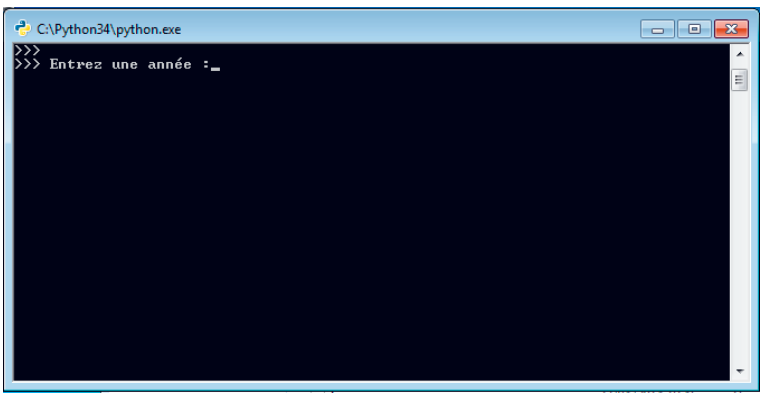


FIGURE 7.2 – Notre programme ne se ferme plus !

Quand vous exécutez ce script, que ce soit sous Windows ou Linux, vous faites toujours appel à l'interpréteur Python ! Votre programme n'est pas compilé mais chaque ligne d'instruction est exécutée à la volée par l'interpréteur, le même qui exécutait vos premiers programmes. La grande différence ici est que Python exécute votre programme depuis le fichier et que donc, si vous souhaitez changer le code, il faudra modifier le fichier.

Sachez qu'il existe des éditeurs spécialisés pour Python, notamment Idle qui est installé en même temps que Python. Ouvrez-le avec un clic droit sur votre fichier `.py` et regardez comment il fonctionne ; ce n'est pas bien compliqué et vous pouvez même exécuter votre programme depuis ce logiciel.

Je viens pour conquérir le monde... et créer mes propres modules

Mes modules à moi

Bon, le plus dur est derrière nous... ça va ? Rassurez-vous, nous n'allons rien faire de compliqué dans cette dernière section.

Commencez par vous créer un espace de test pour les petits programmes Python que nous allons être amenés à créer, un joli dossier à l'écart de vos photos et musiques. Nous allons y créer deux fichiers `.py` :

— `multipli.py`, qui contiendra la fonction `table` que nous avons codée au chapitre précédent ;
— `test.py`, qui contiendra le test d'exécution de notre module.

Vous devriez vous en tirer sans problème. N'oubliez pas de spécifier la ligne précisant l'encodage en tête de vos deux fichiers si vous utilisez Windows et le bloc-notes. Maintenant, voyons le code du fichier `multipli.py`.

```
"""Module multipli contenant la fonction table."""

def table(nb, maximum=10):
    """Fonction affichant la table de multiplication par nb.

    Arguments :
        nb (int) : le nombre dont la table de multiplication est à afficher.
        maximum (int, optionnel) : afficher la table de 1 à maximum.

    """
    for i in range(1, maximum + 1): # tant que i est entre 1 et maximum inclus
        print(f"{i} * {nb} = {i * nb}")
```

On se contente de définir une seule fonction, `table`, qui affiche la table de multiplication choisie. Rien n'est nouveau jusqu'ici. Si vous vous souvenez des `docstrings`, dont nous avons parlé au chapitre précédent, vous voyez que nous en avons inséré une nouvelle ici, non pas pour commenter une fonction mais bien un module entier. C'est une bonne habitude à prendre quand nos projets deviennent importants.

Voici le code du fichier `test.py` ; n'oubliez pas la ligne précisant votre encodage, en tête du fichier, si vous êtes sous Windows et rédigez avec le bloc-notes.

```
import os # Windows uniquement
from multipli import *

# test de la fonction table
table(3, 20)
os.system("pause") # Sous Windows uniquement
```

En le lançant directement, voilà ce qu'on obtient :

```
1 * 3 = 3
2 * 3 = 6
3 * 3 = 9
4 * 3 = 12
5 * 3 = 15
```

```
6 * 3 = 18
7 * 3 = 21
8 * 3 = 24
9 * 3 = 27
10 * 3 = 30
11 * 3 = 33
12 * 3 = 36
13 * 3 = 39
14 * 3 = 42
15 * 3 = 45
16 * 3 = 48
17 * 3 = 51
18 * 3 = 54
19 * 3 = 57
20 * 3 = 60
Appuyez sur une touche pour continuer...
```

Nous avons vu comment créer un module ; il suffit de le mettre dans un fichier. On peut alors l'importer depuis un autre fichier *contenu dans le même répertoire* en précisant son nom (sans l'extension `.py`). Notre code, encore une fois, n'est pas très utile mais vous pouvez le modifier pour le rendre plus intéressant ; vous en avez parfaitement les compétences à présent.

Au moment d'importer votre module, Python va lire (ou créer s'il n'existe pas) un fichier `.pyc`. À partir de la version 3.2, ce fichier se trouve dans un dossier `__pycache__`.

Ce fichier est généré par Python et contient le code compilé (ou presque) de votre module. Il ne s'agit pas réellement de langage machine mais d'un format que Python décode un peu plus vite que le code que vous écrivez. Python se charge lui-même de générer ce fichier et vous n'avez pas vraiment besoin de vous en soucier quand vous codez ; simplement, ne soyez pas surpris. Vous pouvez laisser ce dossier `__pycache__` tranquille ou bien le supprimer, Python ne se fâchera pas.

Faire un test dans le module-même

Dans l'exemple que nous venons de voir, nous avons créé deux fichiers, le premier contenant un module, le second le testant. Cependant, vous pourriez exécuter votre module comme un programme à lui tout seul, qui se testerait lui-même. Voyons voir cela.

Reprenons le code du module `multipli`. Il définit une seule fonction, `table`, qu'il serait bon de tester. Oui mais... si nous ajoutons une ligne juste en dessous, par exemple `table(8)`, elle sera exécutée lors de l'importation et, donc, dans le programme appelant le module. Quand vous écrirez `import multipli`, vous verrez la table de multiplication par 8 s'afficher... hum, il y a mieux.

Heureusement, il y a un moyen très rapide de séparer les éléments du code qui doivent être exécutés lorsqu'on lance le module directement en tant que programme ou lorsqu'on cherche à l'importer.

```
"""Module multipli contenant la fonction table."""

import os # Seulement sous Windows

def table(nb: int, maximum: int = 10):
    """Fonction affichant la table de multiplication par nb.

    Arguments :
        nb (int) : le nombre dont la table de multiplication est à afficher.
        maximum (int, optionnel) : afficher la table de 1 à maximum.

    """
    for i in range(1, maximum + 1): # tant que i est entre 1 et maximum inclus
        print(f"{i} * {nb} = {i * nb}")

# test de la fonction table
if __name__ == "__main__":
    table(8)
    os.system("pause") # Sous Windows uniquement
```

N'oubliez pas la ligne indiquant l'encodage si nécessaire !

Voilà. À présent, si vous double-cliquez directement sur le fichier `multipli.py`, vous voyez la table de multiplication par 8. En revanche, si vous l'importez, le code de test ne s'exécutera pas. Tout repose en fait sur la variable `__name__`, qui existe dès le lancement de l'interpréteur. Si elle vaut `__main__`, cela veut dire que le fichier appelé est le fichier exécuté. Autrement dit, si `__name__` vaut `__main__`, vous pouvez mettre un code qui sera exécuté si le fichier est lancé directement comme un exécutable.

Prenez le temps de comprendre ce mécanisme et faites des tests si nécessaire ; cela vous sera utile par la suite.

Les packages

Les modules sont un des moyens de regrouper plusieurs fonctions (et, comme on le verra plus tard, certaines classes également). On peut aller encore au-delà en regroupant des modules dans ce qu'on va appeler des *packages*.

En théorie

Comme je l'ai dit, un *package* sert à regrouper plusieurs modules. Cela permet de ranger plus proprement vos modules, classes et fonctions dans des emplacements séparés. Si vous voulez y accéder, vous allez devoir fournir un chemin vers le module que vous visez. De ce fait, les risques de conflits de noms sont moins importants et surtout, tout est bien plus ordonné.

Par exemple, imaginons que vous installiez un jour une bibliothèque tierce pour écrire une interface graphique. En s'installant, la bibliothèque ne va pas créer ses dizaines (voire ses centaines) de modules au même endroit. Ce serait un peu désordonné... surtout quand on pense qu'on peut ranger tout cela d'une façon plus claire : d'un côté, on peut avoir les différents objets graphiques de la fenêtre, de l'autre les différents évènements (clavier, souris), ailleurs encore les effets graphiques...

Dans ce cas, on va sûrement se retrouver face à un *package* portant le nom de la bibliothèque. Il contiendra probablement d'autres *packages* : `evenements`, `objets` et `effets`. Dans chacun d'eux, on trouvera soit d'autres *packages*, soit des modules contenant eux-mêmes des fonctions.

Ouf ! Cela nous fait une hiérarchie assez complexe, non ? D'un autre côté, c'est tout l'intérêt. Concrètement, pour utiliser cette bibliothèque, on n'est pas obligé de connaître tous ses *packages*, modules et fonctions (heureusement d'ailleurs !), mais juste ceux dont on a réellement besoin.

En pratique

En pratique, les *packages* sont... des répertoires ! Dedans peuvent se trouver d'autres répertoires (d'autres *packages*) ou des fichiers (des modules).

Exemple de hiérarchie

Pour notre bibliothèque imaginaire, la hiérarchie des répertoires et fichiers ressemblerait à ce qui suit :

- Un répertoire du nom de la bibliothèque contenant :
  - un répertoire `evenements` contenant :
    - un module `clavier` ;
    - un module `souris` ;
    - ...
  - un répertoire `effets` contenant différents effets graphiques ;

— un répertoire `objets` contenant les différents objets graphiques de notre fenêtre (boutons, zones de texte, barres de menus).

Importer des packages

Si vous voulez utiliser, dans votre programme, la bibliothèque fictive que nous venons de voir, vous avez plusieurs moyens qui tournent tous autour des mots-clés `from` et `import` :

```
1 import nom_bibliotheque
```

Cette ligne importe le *package* contenant la bibliothèque. Pour accéder aux sous-*packages*, vous utiliserez un point « . » afin de modéliser le chemin menant au module ou à la fonction qui vous intéresse :

```
1 nom_bibliotheque.evenements # Pointe vers le
  ↪ sous-\emph{package} evenements
2 nom_bibliotheque.evenements.clavier # Pointe vers le module
  ↪ clavier
```

Si vous ne voulez importer qu'un seul module (ou une seule fonction) d'un *package*, vous utiliserez une syntaxe similaire, assez intuitive :

```
1 from nom_bibliotheque.objets import bouton
```

En fonction des besoins, vous pouvez décider d'importer tout un *package*, un sous-*package*, un sous-sous-*package*... ou bien juste un module ou même une seule fonction.

Créer ses propres packages

Si vous voulez créer vos propres *packages*, commencez par définir, dans le même dossier que votre programme Python, un répertoire portant le nom du *package*.

Python n'a pas de problème avec des noms de variables ou de fonctions contenant des accents, comme nous l'avons vu. Cependant, nommer les modules ou *packages* avec des accents va poser des problèmes que Python n'est pas toujours capable de résoudre. C'est en partie la faute du système d'exploitation. Il vous est conseillé de garder vos noms de module ou *package* sans accent.

Dans ce répertoire, vous pouvez :

— ranger vos modules, vos fichiers à l'extension `.py` ;
— créer des sous-*packages* de la même façon, en créant autant de sous-répertoires.

Ne mettez pas d'espaces dans vos noms de *packages* et évitez aussi les caractères spéciaux. Quand vous les utilisez dans vos programmes, ces noms sont traités comme des noms de variables et ils doivent donc obéir aux mêmes règles de nommage.

Le fichier d'initialisation

En Python, vous trouverez souvent le fichier d'initialisation `__init__.py` dans un répertoire destiné à devenir un *package*. Ce fichier est optionnel depuis la version 3.3 de Python. Je ne vais pas rentrer dans le détail ici (vous avez déjà beaucoup de choses à retenir), mais sachez que ce code d'initialisation est appelé quand vous importez votre *package*.

Un dernier exemple

Voici un dernier exemple, que vous pouvez cette fois coder en même temps que moi pour vous assurer que cela fonctionne.

Dans votre répertoire de code, là où vous mettez vos exemples Python, créez un fichier que vous appelerez `test_package.py`.

Créez dans le même répertoire un dossier `package`. Dedans, créez un fichier `fonctions.py` dans lequel vous recopierez votre fonction `table`.

Dans votre fichier `test_package.py`, si vous voulez importer votre fonction `table`, vous avez plusieurs solutions :

```
1 from package.fonctions import table
2 table(5) # Appel de la fonction table
3
4 # Ou ...
5 import package.fonctions
6 package.fonctions.table(5) # Appel de la fonction table
```

Voilà. Il reste bien des choses à dire sur les *packages* mais je crois que vous avez vu l'essentiel. Cette petite explication révélera son importance quand vous aurez à construire des programmes assez volumineux. Évitez de tout mettre dans un seul module sans chercher à hiérarchiser ; profitez de cette possibilité offerte par Python.

En résumé

— On peut écrire les programmes Python dans des fichiers portant l'extension `.py`.
— On peut créer des fichiers contenant des **modules** pour séparer le code.
— On peut créer des répertoires contenant des **packages** pour hiérarchiser un programme.

Chapitre 8

Les exceptions

Difficulté :

Dans ce chapitre, nous aborderons le dernier concept que je considère comme indispensable avant d'attaquer la partie sur la Programmation Orientée Objet : j'ai nommé « les exceptions ».

Comme vous allez le voir, il s'agit des erreurs que peut rencontrer Python en exécutant votre programme. Elles sont faciles à intercepter et c'est même, dans certains cas, indispensable.

Cependant, il ne faut pas tout intercepter non plus : si Python envoie une erreur, c'est qu'il y a une raison. Si vous ignorez une erreur, vous risquez d'obtenir des résultats très étranges dans votre programme.

À quoi cela sert-il ?

Nous avons déjà été confrontés à des erreurs dans nos programmes ; certaines que j'ai volontairement provoquées, d'autres que vous avez dû rencontrer si vous avez testé un minimum des instructions dans l'interpréteur. Quand Python rencontre une erreur dans votre code, il **lève une exception**.

```
>>> # Exemple classique : test d'une division par zéro
... variable = 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Attardons-nous sur la dernière ligne. Nous y trouvons deux informations :

— `ZeroDivisionError` : le type de l'exception ;
— `division by zero` : le message qu'envoie Python pour vous aider à comprendre l'erreur qui vient de se produire.

Python lève donc des exceptions quand il trouve une erreur, soit dans le code (une erreur de syntaxe, par exemple), soit dans l'opération que vous lui demandez de faire.

Notez qu'à l'instar des variables, on trouve différents types d'exceptions que Python va utiliser en fonction de la situation. Le type `ValueError`, notamment, sera levé par Python face à diverses erreurs de « valeurs ». Dans ce cas, c'est donc le message qui vous indique plus clairement le problème. Nous verrons dans la prochaine partie, consacrée à la Programmation Orientée Objet, ce que sont réellement ces types d'exceptions.

Bon, c'est bien joli d'avoir cette exception. On voit le fichier et la ligne à laquelle s'est produite l'erreur (très pratique quand on commence à travailler sur un projet) et on a une indication sur le problème qui suffit en général à le régler. Toutefois, Python permet quelque chose de bien plus pratique.

Admettons que certaines erreurs puissent être provoquées par l'utilisateur. Par exemple, on demande à ce dernier de saisir un entier au clavier et il tape une chaîne de caractères... problème. Nous avons déjà rencontré cette situation : souvenez-vous du programme `bissextile`.

```
année = input() # On demande à l'utilisateur de saisir l'année
année = int(année) # On veut convertir l'année en un entier
```

Je vous avais dit que si l'utilisateur fournissait ici une valeur impossible à convertir en entier (une lettre par exemple), le programme plantait. En fait, il lève une exception et Python arrête l'exécution. Si vous testez le programme en double-cliquant directement dans l'explorateur, il va se fermer tout de suite (en fait, il affiche bel et bien l'erreur mais se referme aussitôt).

Dans ce cas, et dans d'autres cas similaires, Python permet de tester un extrait de code. S'il ne renvoie aucune erreur, Python continue. Sinon, on peut lui demander d'exécuter une autre action (par exemple, redemander à l'utilisateur de saisir l'année). C'est ce que nous allons voir ici.

Forme minimale du bloc `try`

On va parler ici du bloc `try`. Nous allons en effet écrire les instructions que nous souhaitons tester dans un premier bloc et celles à exécuter en cas d'erreur dans un autre bloc :

```
try:
    # Bloc à essayer
except:
    # Bloc qui sera exécuté en cas d'erreur
```

Dans l'ordre, nous trouvons :

- Le mot-clé `try` suivi des deux points « : » (*try* signifie « essayer » en anglais).
- Le bloc d'instructions à essayer.
- Le mot-clé `except` suivi, une fois encore, des deux points « : ». Il se trouve au même niveau d'indentation que le `try`.
- Le bloc d'instructions qui sera exécuté si une erreur est trouvée dans le premier bloc.

Reprenons notre test de conversion en enfermant dans un bloc `try` l'instruction susceptible de lever une exception.

```
année = input()
try: # On essaye de convertir l'année en entier
    année = int(année)
except:
    print("Erreur lors de la conversion de l'année.")
```

Vous pouvez tester ce code en précisant plusieurs valeurs différentes pour la variable `année`, comme « 2010 » ou « année2010 ».

Dans le titre de cette section, j'ai parlé de *forme minimale* et ce n'est pas pour rien. D'abord, il va de soi que vous ne pouvez intégrer cette solution directement dans votre code. En effet, si l'utilisateur saisit une année impossible à convertir, le système affiche certes une erreur mais finit par planter (puisque l'année, au final, n'a pas été convertie). Une des solutions envisageables est d'attribuer une valeur par défaut ou de redemander à l'utilisateur de saisir l'année.

Ensuite et surtout, cette méthode est assez grossière. Elle essaye une instruction et intercepte *n'importe quelle* exception liée à cette instruction. Ici, c'est acceptable car nous n'avons pas énormément d'erreurs possibles, mais c'est une mauvaise habitude à prendre. Voici une manière plus élégante et moins dangereuse.

Forme plus complète

Nous allons apprendre à compléter notre bloc `try`. Comme je l'ai indiqué plus haut, la forme minimale est à éviter pour plusieurs raisons.

D'abord, elle ne différencie pas les exceptions potentiellement levées dans le bloc `try`. Ensuite, Python risque de lever des exceptions ne signifiant pas nécessairement qu'il y a eu une erreur.

Exécuter le bloc `except` pour un type d'exception précis

Dans l'exemple que nous avons vu plus haut, on ne pense qu'à un type d'exceptions susceptible d'être levé : le type `ValueError`, qui trahirait une erreur de conversion. Voyons un autre exemple :

```
try:
    résultat = numérateur / dénominateur
except:
    print("Une erreur est survenue... laquelle ?")
```

Ici, plusieurs erreurs sont susceptibles d'intervenir, chacune levant une exception différente.

- `NameError` : l'une des variables `numérateur` ou `dénominateur` n'a pas été définie (elle n'existe pas). Si vous essayez dans l'interpréteur l'instruction `print(numérateur)` alors que vous n'avez pas défini la variable `numérateur`, vous provoquerez la même erreur.
- `TypeError` : l'une des variables `numérateur` ou `dénominateur` ne peut diviser ou être divisée (les chaînes de caractères ne peuvent être divisées, ni diviser d'autres types, par exemple). Cette exception est levée car vous utilisez l'opérateur de division « / » sur des types qui ne savent pas quoi en faire.
- `ZeroDivisionError` : encore elle ! Si `dénominateur` vaut 0, cette exception sera levée.

Cette énumération n'est pas une liste exhaustive de toutes les exceptions susceptibles d'être levées à l'exécution de ce code. Elle est surtout là pour vous montrer que plusieurs erreurs risquent de se produire sur une instruction (c'est encore plus flagrant sur un bloc constitué de plusieurs instructions) et que la forme minimale intercepte toutes ces erreurs sans les distinguer, ce qui est problématique dans certains cas.

Tout se joue sur la ligne du `except`. Entre ce mot-clé et les deux points, vous allez préciser le type de l'exception que vous souhaitez traiter.

```
try:
    résultat = numérateur / dénominateur
except NameError:
    print("La variable numérateur ou dénominateur n'a pas été
    ↪  définie.")
```

Ce code ne traite que le cas où une exception `NameError` est levée. On intercepte les autres types d'exceptions en créant d'autres blocs `except` à la suite :

```
try:
    résultat = numérateur / dénominateur
except NameError:
    print("La variable numérateur ou dénominateur n'a pas été
    ↪  définie.")
except TypeError:
    print("La variable numérateur ou dénominateur est d'un
    ↪  type incompatible avec la division.")
```

```
except ZeroDivisionError:
    print("La variable dénominateur est égale à 0.")
```

C'est mieux non ?

Allez un petit dernier !

On peut capturer l'exception et afficher son message grâce au mot-clé `as` que vous avez déjà vu dans un autre contexte (si si, rappelez-vous l'importation de modules).

```
try:
    # Bloc de test
except type_de_l_exception as exception_retournée:
    print("Voici l'erreur :", exception_retournée)
```

Dans ce cas, une variable `exception_retournée` est créée par Python si une exception du type précisé est levée dans le bloc `try`.

Je vous conseille de *toujours* préciser un type d'exceptions après `except` (sans nécessairement capturer l'exception dans une variable, bien entendu). D'abord, vous ne devez pas utiliser `try` comme une méthode miracle pour tester n'importe quel bout de code. Il est important que vous gardiez le maximum de contrôle sur votre code. Cela signifie que, si une erreur se produit, vous devez être capable de l'anticiper. En pratique, vous n'irez pas jusqu'à tester si une variable quelconque existe bel et bien ; il faut faire un minimum confiance à son code. Cependant, si vous êtes en face d'une division et si le dénominateur risque d'être nul, placez la division dans un bloc `try` et précisez, après le `except`, le type de l'exception qui risque de se produire (`ZeroDivisionError` dans cet exemple).

Si vous adoptez la forme minimale (à savoir `except` sans préciser un type), toutes les exceptions seront traitées de la même façon. Et, même si exception est synonyme d'erreur la plupart du temps, ce n'est pas toujours le cas. Par exemple, Python lève une exception quand vous voulez fermer votre programme avec le raccourci CTRL + C. Ici, vous ne voyez peut-être pas le problème ; pourtant, si votre bloc `try` est dans une boucle, vous ne pourrez pas arrêter votre programme avec CTRL + C, puisque l'exception sera traitée par votre `except`.

Je vous conseille donc de toujours préciser un type d'exception possible après votre `except`. Vous pouvez bien entendu faire des tests dans l'interpréteur de commandes Python pour reproduire l'exception que vous voulez traiter et ainsi connaître son type.

Les mots-clés `else` et `finally`

Grâce à ces deux mots-clés, nous allons construire un bloc `try` plus complet.

Le mot-clé `else`

Vous avez déjà vu ce mot-clé et j'espère que vous vous le rappelez. Dans un bloc `try`, `else` va permettre d'exécuter une action si aucune erreur ne survient dans le bloc. Voici un petit exemple :

```
try:
    résultat = numérateur / dénominateur
except NameError:
    print("La variable numérateur ou dénominateur n'a pas été
    ↪ définie.")
except TypeError:
    print("La variable numérateur ou dénominateur est d'un
    ↪ type incompatible avec la division.")
except ZeroDivisionError:
    print("La variable dénominateur est égale à 0.")
else:
    print(f"Le résultat obtenu est {résultat}")
```

Dans les faits, on utilise assez peu `else`. Beaucoup de codeurs préfèrent écrire la ligne contenant le `print` directement dans le bloc `try`. Pour ma part, je trouve que c'est important de distinguer entre le bloc `try` et ce qui s'effectue ensuite. La ligne du `print` ne produira vraisemblablement aucune erreur ; il est inutile de la placer dans le bloc `try`.

Le mot-clé `finally`

`finally` permet d'exécuter du code après un bloc `try`, *quelle que soit le résultat de l'exécution dudit bloc.* La syntaxe est des plus simples :

```
try:
    # Test d'instruction(s)
except TypeDInstruction:
    # Traitement en cas d'erreur
finally:
    # Instruction(s) exécutée(s) qu'il y ait eu des erreurs ou
    ↪ non
```

Est-ce que cela ne revient pas au même si on met du code juste après le bloc ?

Pas tout à fait. Le bloc `finally` est exécuté dans tous les cas de figure. Quand bien même Python trouverait une instruction `return` dans votre bloc `except` par exemple, il exécutera le bloc `finally`.

Un petit bonus : le mot-clé `pass`

Il arrive, dans certains cas, que l'on souhaite tester un bloc d'instructions… mais ne rien faire en cas d'erreur. Toutefois, un bloc `try` ne peut être seul.

```
>>> try:
...     1/0
```

```
...
  File "<stdin>", line 3
    ^
SyntaxError: expected 'except' or 'finally' block
```

Il existe un mot-clé utilisable dans ce cas. Son nom est `pass` et sa syntaxe est très simple d'utilisation :

```
try:
    # Test d'instruction(s)
except type_de_l_exception: # Rien ne doit se passer en cas d'erreur
    pass
```

Je ne vous encourage pas particulièrement à utiliser ce mot-clé mais il existe et vous le savez à présent.

`pass` n'est pas un mot-clé propre aux exceptions : on peut également le trouver dans des conditions ou dans des fonctions que l'on souhaite laisser vides.

Voilà, nous avons vu l'essentiel. Il nous reste à faire un petit point sur les assertions et à voir comment lever une exception (ce sera très rapide).

Les assertions

Les assertions sont un moyen simple de s'assurer, avant de continuer, qu'une condition est respectée. Cette syntaxe est en théorie utilisée par les développeurs d'une fonctionnalité, pas par les utilisateurs.

Voyons comment cela fonctionne : nous allons pour l'occasion découvrir un nouveau mot-clé (encore un), `assert`. Sa syntaxe est la suivante :

```
assert test
```

Si le test renvoie `True`, l'exécution se poursuit normalement. Sinon, une exception `AssertionError` est levée.

Voyons un exemple :

```
>>> var = 5
>>> assert var == 5
>>> assert var == 8
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>>
```

Comme vous le voyez, la ligne 2 s'exécute sans problème et ne lève aucune exception. On teste en effet si `var == 5`. C'est le cas, le test est donc vrai, aucune exception n'est levée.

À la ligne suivante, cependant, l'assertion est `var == 8`. Cette fois, le test est faux et une exception du type `AssertionError` est levée.

À quoi cela sert-il, concrètement ?

`assert` soulève des questions intéressantes. En théorie, on pourrait l'utiliser, par exemple, dans notre programme pour vérifier que l'année entrée par l'utilisateur est non nulle. En pratique cependant, `assert` ne devrait pas être utilisée pour vérifier des entrées utilisateurs. C'est plus une limitation philosophique (nous pourrions très bien le faire). Python a attribué `assert` comme mot-clé de `debug` (c'est-à-dire qu'il est utile pour les développeurs d'une fonctionnalité, les utilisateurs ne devraient jamais voir cette erreur).

En résumé, nous pourrions éventuellement l'utiliser dans notre fonction qui affiche les tables de multiplication. Voyez plutôt :

```
"""Module multipli contenant la fonction table."""

def table(nb, maximum=10):
    """Fonction affichant la table de multiplication par nb.

    Arguments :
        nb (int) : le nombre dont la table de multiplication est à afficher.
        maximum (int, optionnel) : afficher la table de 1 à maximum.

    """
    assert maximum > 0
    for i in range(1, maximum + 1): # tant que i est entre 1 et maximum inclus
        print(f"{i} * {nb} = {i * nb}")
```

Ici, une exception `AssertionError` sera automatiquement levée si le développeur appelant la fonction précise un `maximum` négatif ou nul. La raison, nous l'avons vue : si le maximum est négatif ou nul, la fonction ne fait rien. C'est une information utile pour les programmeurs qui lisent notre code plutôt que notre magnifique documentation (il faut de tout pour faire un monde).

En revanche, au moment où un utilisateur non programmeur se sert de cette fonction (dans un programme que nous avons nous-mêmes écrit par exemple), l'erreur ne devrait jamais s'afficher. `AssertionError` ne devrait jamais être utilisée pour valider les entrées utilisateur (bien que vous puissiez voir du code dans ce sens au cours de vos vagabondages sur Internet).

Lever une exception

Hmmm... je vois d'ici les mines sceptiques (non non, ne vous cachez pas !). Vous vous demandez probablement pourquoi vous feriez le travail de Python en levant des exceptions. Après tout, le vôtre, c'est en théorie d'éviter que votre programme plante.

Parfois, cependant, il sera utile de lever des exceptions. Je vous ai dit que le mot-clé `assert` ne devrait pas être utilisé pour valider les entrées utilisateur, mais c'est un excellent moment pour lever et intercepter une autre exception... voyons d'abord la syntaxe.

On utilise un nouveau mot-clé pour lever une exception... le mot-clé `raise`.

```
raise TypeDeLException
# ... ou bien ...
raise TypeDeLException("message à afficher")
```

Prenons un petit exemple, toujours autour de notre programme `bissextile`. Nous allons lever une exception de type `ValueError` si l'utilisateur saisit une année nulle.

```
année = input() # L'utilisateur saisit l'année
try:
    année = int(année) # On tente de convertir l'année
    if année == 0:
        raise ValueError
except ValueError:
    print("La valeur saisie est invalide (l'année est
    ↪  peut-être nulle).")
```

Le code est relativement court et lisible. Pourtant, il y a quelques petites choses intéressantes à noter :

- `ValueError` est levée si l'année entrée par l'utilisateur ne peut être convertie en entier.
- `ValueError` est également levée si l'année entrée par l'utilisateur est nulle.
- Ces deux erreurs sont traitées de façon similaire, dans le même bloc `except`, ce qui permet de regrouper les messages ou le traitement de l'erreur.

En d'autres termes, notre bloc `except` est appelé dans les deux cas où l'entrée utilisateur n'est pas correcte. C'est économique et assez agréable. Certains développeurs en Python trouvent cette syntaxe quelque peu dérangeante ; il faut avouer que c'est surtout une question de goût ici.

Il reste des choses à découvrir sur les exceptions, mais on en a assez fait pour ce chapitre et cette partie. Je ne vous demande pas de connaître toutes les exceptions que Python est amené à utiliser (certaines d'entre elles pourront d'ailleurs n'exister que dans certains modules). En revanche, vous devez être capables de savoir, grâce à l'interpréteur de commandes, quelles exceptions risquent d'être levées par Python dans une situation donnée.

En résumé

— On peut intercepter les erreurs (ou exceptions) levées par notre code grâce aux blocs `try` `except`.
— La syntaxe d'une assertion est `assert test:`.
— Les assertions lèvent une exception `AssertionError` si le test échoue.
— On peut lever une exception grâce au mot-clé `raise` suivi du type de l'exception.

Chapitre 9

TP : tous au ZCasino

Difficulté :

L'heure de vérité a sonné ! C'est dans ce premier TP que je vais faire montre de ma cruauté sans limite en vous lâchant dans la nature... ou presque. Ce n'est pas tout à fait votre premier TP, dans le sens où le programme du chapitre 4, sur les conditions, constituait votre première expérience en la matière. Cependant, à ce moment-là, nous n'avions pas écrit un programme très... récréatif.

Cette fois, nous allons nous atteler au développement d'un petit jeu de casino. Vous trouverez le détail de l'énoncé plus bas, ainsi que quelques conseils pour la réalisation de ce TP.

Si, durant le codage, vous sentez que certaines connaissances vous manquent, revenez en arrière ; prenez tout votre temps, on n'est pas pressé !

Notre sujet

Dans ce chapitre, nous allons coder un petit programme que nous appellerons ZCasino. Il s'agira d'un petit jeu de roulette très simplifié dans lequel vous pourrez miser une certaine somme et gagner ou perdre de l'argent (telle est la fortune, au casino !). Quand vous n'avez plus d'argent, vous avez perdu.

Notre règle du jeu

Bon, la roulette, c'est très sympathique comme jeu, mais un peu trop compliqué pour un premier TP. Alors, on va simplifier les règles et je vous présente tout de suite ce que l'on obtient :

— Le joueur mise sur un numéro compris entre 0 et 49 (50 en tout). En choisissant son numéro, il y dépose la somme qu'il souhaite miser.
— La roulette est constituée de 50 cases allant naturellement de 0 à 49. Les numéros pairs sont de couleur noire, les numéros impairs sont de couleur rouge. Le croupier lance la roulette, lâche la bille et quand la roulette s'arrête, relève le numéro de la case dans laquelle la bille s'est arrêtée. Dans notre programme, nous ne reprendrons pas tous ces détails « matériels » mais ces explications sont aussi à l'intention de ceux qui ont eu la chance d'éviter les salles de casino jusqu'ici. Le numéro sur lequel s'est arrêtée la bille est, naturellement, le numéro gagnant.
— Si le numéro gagnant est celui sur lequel le joueur a misé (probabilité de 1/50, plutôt faible), le gain est de 3 fois la somme misée.
— Sinon, le croupier regarde si le numéro misé par le joueur est de la même couleur que le numéro gagnant (s'ils sont tous les deux pairs ou tous les deux impairs). Si c'est le cas, le gain est de 50 % de la somme misée. Si ce n'est pas le cas, le joueur perd sa mise.

Dans les deux scénarios gagnants vus ci-dessus (le numéro misé et le numéro gagnant sont identiques ou ont la même couleur), le croupier remet au joueur la somme initialement misée avant d'y ajouter ses gains. Cela veut dire que, dans ces deux scénarios, le joueur récupère de l'argent. Il n'y a que dans le troisième cas qu'il perd la somme misée.

Organisons notre projet

Pour ce projet, nous n'allons pas écrire de module. Nous allons utiliser ceux de Python, qui sont bien suffisants pour l'instant, notamment celui permettant de générer de l'aléatoire, que je vais présenter plus bas. En attendant, ne vous privez quand même pas de créer un répertoire et d'y ranger le fichier `ZCasino.py`, tout va s'y jouer.

Vous êtes capables d'écrire le programme `ZCasino` tel qu'expliqué dans la première partie sans difficulté… sauf pour générer des nombres aléatoires. Python a dédié tout un module à la génération d'éléments pseudo-aléatoires, le module `random`.

Le module `random`

Dans ce module, nous allons nous intéresser particulièrement à la fonction `randint` qui s'utilise en précisant deux paramètres (`randint(1, 6)` : renvoie un nombre aléatoire compris entre 1 et 6 (compris), ce qui est utile, par exemple, pour reproduire une expérience avec un dé à six faces).

Pour tirer un nombre aléatoire compris entre 0 et 49 et simuler ainsi l'expérience du jeu de la roulette, nous allons donc utiliser l'instruction `randint(0, 49)`.

N'hésitez pas à réaliser des tests dans l'interpréteur de commandes et essayez plusieurs fois la fonction `randint`. Je vous rappelle qu'elle se trouve dans le module `random` ; n'oubliez pas de l'importer.

Prenez également le réflexe de lire la documentation officielle : vous n'avez plus trop d'excuses pour l'éviter, car elle est traduite et maintenue à jour. Vous trouverez la documentation du module `random` à l'adresse suivante : `https://docs.python.org/fr/3/library/random.html`

Arrondir un nombre

Vous l'avez peut-être bien noté, dans l'explication des règles je spécifiais que si le joueur misait sur la bonne couleur, il obtenait 50% de sa mise. Oui mais... c'est quand même mieux de travailler avec des entiers. Si le joueur mise 3 €, par exemple, on lui rend 1,5 €. C'est encore acceptable mais, si cela se poursuit, on risque d'arriver à des nombres flottants avec beaucoup de chiffres après la virgule. Alors autant arrondir à l'entier supérieur. Ainsi, si le joueur mise 3 €, on lui rend 2 €. Pour cela, on va utiliser une fonction du module `math` nommée `ceil`. Je vous laisse regarder ce qu'elle fait, il n'y a rien de compliqué.

À vous de jouer

Voilà, vous avez toutes les clés en main pour coder ce programme. Prenez le temps qu'il faut pour y arriver. Ne vous ruez pas sur la correction ; le but du TP est que vous appreniez à coder vous-mêmes un programme... et celui-ci n'est pas très difficile. Si vous avez du mal, morcelez le programme, ne codez pas tout d'un coup. Et n'hésitez pas à passer par l'interpréteur pour tester des fonctionnalités : c'est réellement une chance qui vous est donnée, ne la laissez pas passer.

À vous de jouer !

Correction !

C'est l'heure de comparer nos versions. Et, une fois encore, il est très peu probable que vous ayez un code identique au mien. Donc si le vôtre fonctionne, je dirais que c'est

l'essentiel. Si vous vous heurtez à des difficultés insurmontables, la correction est là pour vous aider.

ATTENTION... voici... la solution !

```
# Ce fichier abrite le code du ZCasino, un jeu de roulette adapté

import os
from random import randint
from math import ceil

# Déclaration des variables de départ
argent = 1000 # On a 1000 euros au début du jeu
continuer_partie = True # Booléen qui est vrai tant qu'on doit
                        # continuer la partie

print(f"Vous vous installez à la table de roulette avec {argent} euro(s).")

while continuer_partie: # Tant qu'on doit continuer la partie
    # on demande à l'utilisateur de saisir le nombre sur
    # lequel il va miser
    nombre_misé = -1
    while nombre_misé < 0:
        nombre_misé = input("Tapez le nombre sur lequel vous voulez miser (entre 0 et 49) : ")
        # On convertit le nombre misé
        try:
            nombre_misé = int(nombre_misé)
            if nombre_misé < 0 or nombre_misé > 49:
                raise ValueError
        except ValueError:
            print("Vous n'avez pas saisi de nombre valide.")
            nombre_misé = -1

    # À présent, on sélectionne la somme à miser sur le nombre
    mise = -1
    while mise <= 0 or mise > argent:
        mise = input("Tapez le montant de votre mise : ")
        # On convertit la mise
        try:
            mise = int(mise)
            if mise <= 0 or mise > argent:
```

```
                raise ValueError
            except ValueError:
                print("Vous n'avez pas saisi de nombre valide.")
                mise = -1

    # Le nombre misé et la mise ont été sélectionnés par
    # l'utilisateur, on fait tourner la roulette
    numéro_gagnant = randint(0, 49)
    print(f"La roulette tourne... ... et s'arrête sur le
    ↪ numéro {numéro_gagnant}.")

    # On établit le gain du joueur
    if numéro_gagnant == nombre_misé:
        print(f"Félicitations ! Vous obtenez {mise * 3}
        ↪ euro(s) !")
        argent += mise * 3
    elif numéro_gagnant % 2 == nombre_misé % 2: # ils sont de
    ↪ la même couleur
        mise = ceil(mise * 0.5)
        print(f"Vous avez misé sur la bonne couleur. Vous
        ↪ obtenez {mise}
        euro(s).")
        argent += mise
    else:
        print("Désolé l'ami, ce n'est pas pour cette fois.
        ↪ Vous perdez votre mise.")
        argent -= mise

    # On interrompt la partie si le joueur est ruiné
    if argent <= 0:
        print("Vous êtes ruiné ! C'est la fin de la partie.")
        continuer_partie = False
    else:
        # On affiche l'argent du joueur
        print(f"Vous avez à présent {argent} euro(s).")
        match input("Souhaitez-vous quitter le casino (o/n) ?
        ↪ "):
            # l'utilisateur entre 'o' ou 'O'
            case "O" | "o":
                print("Vous quittez le casino avec vos
                ↪ gains.")
                continuer_partie = False

# On met en pause le système (Windows)
os.system("pause")
```

Pour accéder en ligne au code source de la solution, utilisez le code web suivant :

▷ Copier ce code
Code web : 366476.

Encore une fois, n'oubliez pas la ligne spécifiant l'encodage si vous voulez éviter les surprises.

Une petite chose qui pourrait vous surprendre est la construction des boucles pour tester si le joueur a saisi une valeur correcte (quand on demande à l'utilisateur de taper un nombre entre 0 et 49 par exemple, il faut s'assurer qu'il l'a bien fait). C'est assez simple en vérité : on attend que le joueur saisisse un nombre. Si son choix n'est pas valide, on lui demande de recommencer. J'en ai profité pour utiliser le concept des exceptions afin de vérifier que l'utilisateur saisit bien un nombre. Comme vous l'avez vu, si ce n'est pas le cas, on affiche un message d'erreur. La valeur de la variable qui contient le nombre est remise à `-1` (ce qui indique à la boucle que nous n'avons toujours pas obtenu de l'utilisateur une valeur valide). De cette façon, si l'utilisateur fournit une donnée inconvertible, le jeu ne plante pas et lui demande tout simplement de recommencer. L'inconvénient de cette méthode est que le message reste le même si le joueur entre une donnée non valide (qui n'est pas un nombre) ou un nombre invalide. On aurait pu écrire ces boucles différemment pour tenir compte de ces deux possibilités.

La boucle principale fonctionne autour d'un booléen. On utilise une variable nommée `continuer_partie` qui vaut « vrai » tant qu'on doit continuer la partie. Une fois que la partie doit s'interrompre, elle passe à « faux ». Notre boucle globale, qui gère le déroulement de la partie, travaille sur ce booléen ; par conséquent, dès qu'il passe à la valeur « faux », la boucle s'interrompt et le programme se met en pause. Tout le reste, vous devriez le comprendre sans aide, les commentaires sont là pour vous expliquer. Si vous avez des doutes, vous pouvez tester les lignes d'instructions problématiques dans votre interpréteur de commandes Python : encore une fois, n'oubliez pas cet outil.

Et maintenant ?

Prenez bien le temps de lire ma version et surtout de modifier la vôtre, si vous êtes arrivés à une version qui fonctionne bien ou qui fonctionne presque. Ne mettez pas ce projet à la corbeille sous prétexte que nous avons fini de le coder et qu'il marche. On peut toujours améliorer un projet et celui-ci ne fait évidemment pas exception. Vous trouverez probablement de nouveaux concepts, dans la suite de ce livre, qui seront applicables dans le programme de `ZCasino`.

Deuxième partie

La programmation orientée objet côté utilisateur

Chapitre 10

Notre premier objet : la chaîne de caractères

Difficulté :

Les objets... vaste sujet ! Avant d'en créer, nous allons d'abord voir de quoi il s'agit. Nous allons commencer avec les chaînes de caractères, un type que vous pensez bien connaître.

Dans ce chapitre, vous allez découvrir petit à petit le mécanisme qui se cache derrière la notion d'objet. Ces derniers font partie des notions incontournables en Python, étant donné que tout ce que nous avons utilisé jusqu'ici... est un objet !

Vous avez dit objet ?

La première question qui risque de vous empêcher de dormir si je n'y réponds pas tout de suite, c'est :

Mais qu'est-ce qu'un objet ?

Eh bien, j'ai lu beaucoup de définitions très différentes et je ne leur ai pas trouvé de point commun. Nous allons donc partir d'une définition incomplète mais qui suffira pour l'instant : **un objet est une structure de données, comme les variables, qui peut contenir elle-même d'autres variables et fonctions.** On étoffera plus loin cette définition.

Je ne comprends rien. Passe encore qu'une variable en contienne d'autres, après tout les chaînes de caractères contiennent bien des caractères, mais qu'une variable contienne des fonctions... À quoi cela rime-t-il ?

Je pourrais passer des heures à expliquer la théorie du concept que vous n'en seriez pas beaucoup plus avancés. J'ai choisi de vous montrer les objets par l'exemple et donc, vous allez très rapidement voir ce que tout cela signifie. Toutefois, vous allez devoir me faire confiance, au début, sur l'utilité de la méthode objet.

Avant d'attaquer, une petite précision. J'ai dit qu'un objet était un peu comme une variable... en fait, pour être exact, il faut dire qu'une variable est un objet. Toutes les variables avec lesquelles nous avons travaillé jusqu'ici sont des objets. Les fonctions que nous avons vues sont également des objets. **En Python, tout est objet** : gardez cela à l'esprit.

Les méthodes de la classe `str`

J'en vois qui grimacent rien qu'en lisant le titre. Vous n'avez pourtant aucune raison de vous inquiéter ! On va y aller tout doucement.

Posons un problème : comment peut-on passer une chaîne de caractères en minuscules ? Si vous n'avez lu jusqu'à présent que les premiers chapitres, vous ne pourrez pas faire cet exercice ; j'ai volontairement évité de trop aborder les chaînes de caractères jusqu'ici. Admettons que vous arriviez à coder une fonction prenant en paramètre la chaîne en question. Vous aurez un code qui ressemble à ceci :

```
>>> chaîne = "NE CRIE PAS SI FORT !"
>>> mettre_en_minuscule(chaîne)
'ne crie pas si fort !'
```

Sachez que, dans les anciennes versions de Python, il y avait un module spécialisé dans le traitement des chaînes de caractères. On importait ce module et on pouvait appeler

la fonction passant une chaîne en minuscules. Ce module existe d'ailleurs encore et reste utilisé pour certains traitements spécifiques. Cependant, on va découvrir ici une autre façon de faire. Regardez attentivement :

```
>>> chaîne = "NE CRIE PAS SI FORT !"
>>> chaîne.lower() # Mettre la chaîne en minuscule
'ne crie pas si fort !'
```

La fonction `lower` est une nouveauté pour vous. Vous devez reconnaître le point « . » qui symbolisait déjà, dans le chapitre sur les modules, une relation d'appartenance (`a.b` signifiait que `b` était contenu dans `a`). Ici, il possède la même signification : `lower` est une fonction de la variable `chaîne`.

La fonction `lower` est propre aux chaînes de caractères. Toutes les chaînes peuvent y faire appel. Si vous tapez `type(chaîne)` dans l'interpréteur, vous obtenez `<class 'str'>`. Nous avons dit qu'une variable est issue d'un type de donnée. Je vais à présent reformuler : un **objet** est issu d'une **classe**. La **classe** est une forme de type de donnée, sauf qu'elle permet de définir des fonctions et variables propres au type. C'est pour cela que, dans toutes les chaînes de caractères, on peut appeler la fonction `lower`. C'est tout simplement parce que la fonction `lower` a été définie dans la classe `str`. Les fonctions définies dans une classe sont appelées des **méthodes**.

Récapitulons. Nous avons découvert :

- Les **objets**, que j'ai présentés comme des variables, pouvant contenir d'autres variables ou fonctions (que l'on appelle **méthodes**). On appelle une méthode d'un objet grâce à `objet.methode()`.
- Les **classes**, que j'ai présentées comme des types de données. Une classe est un modèle qui servira à construire un objet ; c'est dans la classe qu'on va définir les méthodes propres à l'objet.

Voici le mécanisme qui vous permet d'appeler la méthode `lower` d'une chaîne :

1. Les développeurs de Python ont créé la classe `str` qui est utilisée pour créer des chaînes de caractères. Dans cette classe, ils ont défini plusieurs méthodes, comme `lower`, utilisables par n'importe quel objet construit sur cette classe.
2. Quand vous écrivez `chaîne = "NE CRIE PAS SI FORT !"`, Python reconnaît qu'il doit créer une chaîne de caractères. Il va donc créer un objet d'après la classe (le modèle) qui a été définie à l'étape précédente.
3. Vous pouvez ensuite appeler toutes les méthodes de la classe `str` depuis l'objet `chaîne` que vous venez de créer.

Ouf ! Cela fait beaucoup de choses nouvelles, du vocabulaire et des concepts un peu particuliers.

Vous ne voyez peut-être pas encore tout l'intérêt d'avoir des méthodes définies dans une certaine classe. Tout d'abord, cela sépare les diverses fonctionnalités (on ne peut pas passer en minuscules un nombre entier, cela n'a aucun sens). Ensuite, c'est plus intuitif, une fois passé le choc de la première rencontre.

Bon, on parle, on parle, mais on ne code pas beaucoup !

Mettre en forme une chaîne

Non, vous n'allez pas apprendre à mettre une chaîne en gras, souligné, avec une police Verdana de 15px... Nous ne sommes encore que dans une console. Nous venons de présenter `lower`, il existe d'autres méthodes. Avant tout, voyons un contexte d'utilisation.

Certains d'entre vous se demandent peut-être l'intérêt de passer des chaînes en minuscules... alors voici un petit exemple.

```
chaîne = str() # Crée une chaîne vide
# On aurait obtenu le même résultat en tapant chaîne = ""

while chaîne.lower() != "q":
    print("Tapez 'Q' pour quitter...")
    chaîne = input()

print("Merci !")
```

Vous devez comprendre rapidement ce programme. Dans une boucle, on demande à l'utilisateur de taper la lettre « q » pour quitter. Tant que l'utilisateur saisit une autre lettre, la boucle continue de s'exécuter. Dès que l'utilisateur appuie sur la touche Q de son clavier, la boucle s'arrête et le programme affiche « Merci ! ». Cela devrait vous rappeler quelque chose... direction le TP de la partie 1 pour ceux qui ont la mémoire courte.

La petite nouveauté réside dans le test de la boucle : `chaîne.lower() != "q"`. On prend la chaîne saisie par l'utilisateur, on la passe en minuscules et on regarde si elle est différente de « q ». Cela veut dire que l'utilisateur peut taper « q » en majuscule ou en minuscule ; dans les deux cas, la boucle s'arrêtera.

Notez que `chaîne.lower()` renvoie la chaîne en minuscules mais ne modifie pas la chaîne. C'est très important ; nous verrons pourquoi dans le prochain chapitre.

Notez aussi que nous avons appelé la fonction `str` pour créer une chaîne vide. Je ne vais pas trop compliquer les choses, mais sachez qu'appeler ainsi un type en tant que fonction permet de créer un objet de la classe. Ici, `str()` crée un objet *chaîne de caractères*. Nous avons vu dans la première partie le mot-clé `int()`, qui crée aussi un entier [1].

Parlant du TP, vous vous souvenez peut-être d'une ligne un peu étrange que j'ai écrite sans trop la détailler (en partie parce qu'elle semble relativement intuitive) :

```
# ...
# On affiche l'argent du joueur
print(f"Vous avez à présent {argent} euro(s).")
match input("Souhaitez-vous quitter le casino (o/n) ? "):
    # l'utilisateur entre 'o' ou 'O'
    case "O" | "o":
        print("Vous quittez le casino avec vos gains.")
        continuer_partie = False
```

1. Si nécessaire depuis un autre type, ce qui permet de convertir une chaîne en entier.

Ici, c'est le `case` qui nous intéresse :

```
case "O" | "o":
```

Il signifie qu'on continue le programme si l'utilisateur entre `o` en minuscule ou majuscules. On pourrait réécrire cette partie à présent :

```
print(f"Vous avez à présent {argent} euro(s).")
match input("Souhaitez-vous quitter le casino (o/n) ? ").lower():
    # l'utilisateur entre 'o' ou 'O'
    case "o":
        print("Vous quittez le casino avec vos gains.")
        continuer_partie = False
```

On appelle la méthode `lower()` sur le retour de la fonction `input()` (qui est une chaîne de caractères). Notre `match` travaille donc uniquement avec une chaîne de caractères en minuscules.

Bon, voyons d'autres méthodes. Je vous invite à tester mes exemples (ils sont commentés, mais on retient mieux en essayant par soi-même).

```
>>> minuscules = "une chaîne en minuscules"
>>> minuscules.upper() # Mettre en majuscules
'UNE CHAINE EN MINUSCULES'
>>> minuscules.capitalize() # La première lettre en majuscule
'Une chaîne en minuscules'
>>> espaces = "   une  chaîne avec  des espaces   "
>>> espaces.strip() # On retire les espaces au début et à la fin de la chaîne
'une  chaîne avec  des espaces'
>>> titre = "introduction"
>>> titre.upper().center(20)
'    INTRODUCTION    '
>>>
```

La dernière instruction mérite quelques explications.

On appelle d'abord la méthode `upper` de l'objet `titre`. Elle renvoie en majuscules la chaîne de caractères contenue dans l'objet.

On appelle ensuite la méthode `center`, qui centre une chaîne. On lui passe en paramètre la taille de la chaîne que l'on souhaite obtenir. La méthode va ajouter alternativement un espace au début et à la fin de la chaîne, jusqu'à obtenir la longueur demandée. Dans cet exemple, `titre` contient la chaîne `'introduction'`, qui (en minuscules ou en majuscules) mesure 12 caractères. On demande à `center` de centrer cette chaîne dans un espace de 20 caractères ; la méthode va donc placer 4 espaces avant le titre et 4 après, pour obtenir 20 caractères en tout.

Bon, mais maintenant, sur quel objet `center` travaille-t-elle ? Sur `titre` ? Non. Sur la chaîne renvoyée par `titre.upper()`, c'est-à-dire le titre en majuscules. C'est pourquoi on peut « chaîner » ces deux méthodes : `upper`, comme la plupart des

méthodes de chaînes, travaille sur une chaîne et en renvoie une autre... qui elle aussi va posséder les méthodes propres à une chaîne de caractères. Faites quelques tests, avec `titre.upper()` et `titre.center(20)`, en passant par une seconde variable si nécessaire, pour vous rendre compte du mécanisme.

Je n'ai cité ici que quelques méthodes, mais il en existe bien d'autres. Leur liste est disponible dans l'aide, en tapant `help(str)` dans l'interpréteur. Vous pouvez également consulter l'aide en ligne (et en français) à l'adresse suivante : `https://docs.python.org/fr/3/library/stdtypes.html#str`

Formater et afficher une chaîne

Attends, on a appris à faire cela depuis cinq bons chapitres ! On ne va pas tout réapprendre quand même ?

Heureusement que non ! Nous allons reconsidérer ce que nous savons à travers le modèle objet. Et vous allez vous rendre compte que, la plupart du temps, nous n'avons fait qu'effleurer les fonctionnalités du langage.

Je ne vais pas revenir sur ce que j'ai dit : pour afficher une chaîne, on passe par la fonction `print`.

```
1 chaîne = "Bonjour tout le monde !"
2 print(chaîne)
```

Nous avons également vu les chaînes formatées (`f-string`). Souvenez-vous, il s'agit des chaînes préfixées par `f` :

```
1 >>> prénom = "Paul"
2 >>> f"Je m'appelle {prénom}."
3 "Je m'appelle Paul."
```

Nous allons appliquer une nouvelle méthode de formatage des chaînes. Elle va également nous aider à mieux comprendre ce qui se passe lorsqu'on utilise les chaînes formatées (`f-string`). Regardez le code suivant :

```
1 >>> prénom = "Paul"
2 >>> nom = "Dupont"
3 >>> âge = 21
4 >>> print("Je m'appelle {0} {1} et j'ai {2}
  ↪  ans.".format(prénom, nom, âge))
5 Je m'appelle Paul Dupont et j'ai 21 ans.
```

Mais ! Qu'est-ce que c'est que ça ?

Question légitime. Voyons un peu.

Première syntaxe de la méthode `format`

Nous avons utilisé une méthode de la classe `str` pour formater notre chaîne. De gauche à droite, nous avons :

- une chaîne de caractères qui ne présente rien de particulier, sauf ces accolades entourant des nombres, d'abord `0`, puis `1`, puis `2` ;
- nous appelons la méthode `format` de cette chaîne en lui passant en paramètres les variables à afficher, dans un ordre bien précis ;
- quand Python exécute cette méthode, il remplace dans notre chaîne `{0}` par la première variable passée à la méthode `format` (soit le prénom), `{1}` par la deuxième variable… et ainsi de suite.

En programmation, on compte à partir de 0 et non pas de 1. Gardez cette information en tête car elle est valable, non seulement pour tous les types composés de plusieurs éléments, mais aussi pour les intervalles (dans les boucles d'itération par exemple).

Bien, mais on aurait pu faire exactement la même chose en utilisant la syntaxe des chaînes formatées avec `f""`, non ?

Absolument. Nous allons revenir sur la différence entre les deux. La méthode `format` est encore largement utilisée et elle a quelques avantages. Avant tout, il s'agit d'une bonne opportunité de découvrir une nouvelle méthode et de revenir un peu sur le formatage de chaînes de caractères en Python.

Nous avons utilisé la méthode `.format` avec `print`, mais bien entendu, elle sert aussi pour créer de nouvelles chaînes de caractères, sans nécessairement les afficher :

```
>>> nouvelle_chaîne = "Je m'appelle {0} {1} et j'ai {2}
↪  ans.".format(prénom, nom, âge)
>>>
```

Dans cet exemple, nous avons appelé les variables dans l'ordre où nous les placions dans `format`, mais ce n'est pas une obligation. Considérez cet exemple :

```
>>> prénom = "Paul"
>>> nom = "Dupont"
>>> âge = 21
>>> print(
...   "Je m'appelle {0} {1} ({3} {0} pour l'administration)
↪  et j'ai {2} ans.".format(
...   prénom, nom, âge, nom.upper()))
Je m'appelle Paul Dupont (DUPONT Paul pour l'administration)
↪  et j'ai 21 ans.
```

Si vous avez du mal à comprendre l'exemple, relisez l'instruction en remplaçant vous-mêmes les nombres entre accolades par les variables.

Dans la plupart des cas, on ne précise pas le numéro de la variable entre accolades.

```
>>> date = "Dimanche 24 juillet 2011"
>>> heure = "17:00"
>>> print("Cela s'est produit le {}, à {}.".format(date,
 ↪  heure))
Cela s'est produit le Dimanche 24 juillet 2011, à 17:00.
>>>
```

Naturellement, cela ne fonctionne que si vous donnez les variables dans le bon ordre dans `format`.

Cette syntaxe suffit la plupart du temps mais elle n'est pas forcément intuitive quand on insère beaucoup de variables : on doit retenir leur position dans l'appel à `format` pour comprendre laquelle est affichée à tel endroit. Il existe une autre syntaxe.

Seconde syntaxe de la méthode `format`

On peut également nommer les variables que l'on va afficher, c'est souvent plus intuitif que d'utiliser leur indice. Voici un nouvel exemple :

```
# formatage d'une adresse
adresse = """
  {no_rue}, {nom_rue}
  {code_postal} {nom_ville} ({pays})""".format(
      no_rue=5, nom_rue="rue des Postes",
      code_postal=75003, nom_ville="Paris", pays="France")
print(adresse)
```

Cela affichera :

```
  5, rue des Postes
  75003 Paris (France)
```

Je pense que vous voyez assez précisément en quoi consiste cette deuxième syntaxe de `format`. Au lieu de donner des nombres entre accolades, on spécifie des noms de variables qui doivent correspondre à ceux fournis comme mots-clés dans la méthode `format`. Je ne m'attarderai pas davantage sur ce point, je pense qu'il est assez clair comme cela.

La méthode `format` ou les chaînes formatées avec le préfixe `f` ?

Vous l'avez sans doute remarqué, la seconde syntaxe de la méthode `format` est très proche de la formulation avec le préfixe `f`. Historiquement, la première méthode est apparue bien avant la seconde, qui n'a été intégrée que depuis Python 3.6.

Vous pourriez assez facilement confondre les deux. Il est important de faire la distinction et de savoir utiliser l'une ou l'autre méthode en fonction des cas. Tout d'abord, voici un récapitulatif en code :

```
>>> prénom = "Paul"
>>> # Méthode .format avec nombres (implicite)
... "Je m'appelle {}.".format(prénom)
"Je m'appelle Paul."
>>> # Méthode .format avec arguments nommés
... "Je m'appelle {p}.".format(p=prénom)
"Je m'appelle Paul."
>>> # Méthode avec f-string
... f"Je m'appelle {prénom}."
"Je m'appelle Paul."
>>>
```

Voici ce qu'il faut retenir de ces trois méthodes :

- La troisième méthode (`f-string`) est sans contredit la plus rapide à écrire et la plus facile à lire.
- La seconde méthode est la plus longue à écrire et la plus répétitive.
- La syntaxe des `f-string` est plus étendue ; elle permet de faire des calculs et d'appeler des méthodes d'objets affichés (voir le code ci-après), alors que `format` l'interdira.
- La méthode avec `f-string` interprète le contenu de la chaîne sans délai ; la méthode avec `format` peut s'utiliser dans un contexte plus large.

Les deux derniers points méritent quelques explications :

```
>>> x = 5
>>> f"x * 5 = {x * 5}"
'x * 5 = 25'
>>> "x * 5 = {x * 5}".format(x=x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'x * 5'
>>>
```

La méthode `format` n'autorise pas les calculs ou expressions complexes entre accolades. La méthode avec `f-string` n'a pas cette limitation. Elle paraît donc plus puissante à première vue et elle est assez lisible, une fois qu'on est habitué à la syntaxe.

Malgré tout, `format` a encore son utilité. La méthode avec `f-string` interprète directement les variables définies. La méthode `format` interprète ce qu'on lui passe : elle est donc plus sûre (quand il s'agit de formater du texte inconnu aux utilisateurs) et permet de différer le formatage. C'est un cas particulier assez complexe que nous ne verrons pas ici.

En résumé : à mon humble avis, utiliser `f"..."` est le plus simple et efficace dans une très grande majorité des cas. Gardez simplement à l'esprit que cette méthode n'est pas la seule, que `format` est toujours utilisée pour des raisons en partie historiques et qu'un jour vous pourriez en avoir besoin vous-mêmes.

Insistons une fois de plus, `f"..."` n'est pas identique à `"..."`. Elles créent toutes deux des chaînes de caractères (classe `str`) mais la première demande à Python d'interpréter la chaîne pour en extraire les expressions entre accolades. La seconde ne le fait pas.

La concaténation de chaînes

Nous allons glisser très rapidement sur le concept de concaténation, assez intuitif d'ailleurs. On cherche à regrouper deux chaînes en une, en mettant la seconde à la suite de la première. Cela se fait le plus simplement du monde :

```
>>> prénom = "Paul"
>>> message = "Bonjour"
>>> chaîne_complete = message + prénom # On utilise le
 ↪  symbole '+' pour concaténer deux chaînes
... print(chaîne_complete) # Résultat :
BonjourPaul
>>> # Pas encore parfait, il manque un espace
... # Qu'à cela ne tienne !
... chaîne_complete = message + " " + prénom
>>> print(chaîne_complete) # Résultat :
Bonjour Paul
>>>
```

C'est assez clair, je pense. Le signe « + » utilisé pour ajouter des nombres sert ici pour **concaténer** deux chaînes. Essayons à présent de concaténer des chaînes et des nombres :

```
>>> âge = 21
>>> message = "J'ai " + âge + " ans."
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>>
```

Python se fâche tout rouge, alors que d'autres langages auraient accepté cette syntaxe sans sourciller.

Au début de la première partie, nous avons dit que Python était un langage à **typage dynamique**, ce qui signifie qu'il identifie lui-même les types de données et que les variables peuvent changer de type au cours du programme. Néanmoins, Python est aussi un langage **fortement typé**, ce qui veut dire que les types de données ne sont pas là juste pour faire joli. Ainsi, on veut ici ajouter une chaîne à un entier et à une autre chaîne. Python ne comprend pas : est-ce que les chaînes contiennent des nombres qu'il doit convertir pour les ajouter à l'entier ou est-ce que l'entier doit être converti en chaîne puis concaténé avec les autres chaînes ? Il ne le sait pas. Il ne le fera pas tout seul. Pourtant, il se révèle de bonne volonté puisqu'il suffit de lui demander de convertir l'entier pour pouvoir le concaténer aux autres chaînes.

```
>>> âge = 21
>>> message = "J'ai " + str(âge) + " ans."
>>> print(message)
J'ai 21 ans.
>>>
```

On appelle `str` pour convertir un objet en une chaîne de caractères, comme nous avons appelé `int` pour convertir un objet en entier. C'est le même mécanisme, sauf que convertir un entier en chaîne de caractères ne lèvera vraissemblablement aucune exception.

Le typage fort de Python est important, il est un fondement de sa philosophie. J'ai tendance à considérer qu'un langage faiblement typé crée des erreurs qui sont plus difficiles à repérer alors qu'ici, il nous suffit de convertir explicitement le type pour que Python sache ce qu'il doit faire.

Parcours et sélection de chaînes

Nous avons vu très rapidement dans la première partie un moyen de parcourir des chaînes. Nous allons en voir ici un second qui fonctionne par indice.

Parcours par indice

Vous devez vous en souvenir : j'ai dit qu'une chaîne de caractères était une séquence constituée... de caractères. En fait, une chaîne de caractères est elle-même constituée de chaînes de caractères, chacune d'elles n'étant composée que d'un seul caractère.

Accéder aux caractères d'une chaîne

Nous allons apprendre à accéder aux lettres constituant une chaîne. Par exemple, nous souhaitons sélectionner la première lettre d'une chaîne.

```
>>> chaîne = "Salut les ZER0S !"
>>> chaîne[0] # Première lettre de la chaîne
'S'
>>> chaîne[2] # Troisième lettre de la chaîne
'l'
>>> chaîne[-1] # Dernière lettre de la chaîne
'!'
>>>
```

On précise entre crochets `[]` l'indice (la position) du caractère auquel on souhaite accéder.

Rappelez-vous, on commence à compter à partir de 0. La première lettre est donc à l'indice 0, la deuxième à l'indice 1, la troisième à l'indice 2... On peut accéder aux lettres en partant de la fin à l'aide d'un indice négatif. Quand vous tapez `chaîne[-1]`, vous accédez ainsi à la dernière lettre de la chaîne (enfin, au dernier caractère, qui n'est pas une lettre ici).

On obtient la longueur de la chaîne (le nombre de caractères qu'elle contient) grâce à la fonction `len`.

```
>>> chaîne = "Salut"
>>> len(chaîne)
5
>>>
```

Pourquoi ne pas avoir défini cette fonction comme une méthode de la classe `str` ? Pourquoi ne pourrait-on pas faire `chaîne.len()` ?

En fait c'est un peu le cas, mais nous le verrons bien plus loin. `str` n'est qu'un exemple parmi d'autres de séquences (on en découvrira d'autres dans les prochains chapitres) et donc les développeurs de Python ont préféré créer une fonction qui travaillerait sur les séquences au sens large, plutôt qu'une méthode pour chacune de ces classes.

Méthode de parcours par `while`

Vous en savez assez pour parcourir une chaîne grâce à la boucle `while`. Notez que, dans la plupart des cas, on préférera parcourir une séquence avec `for`. Il est néanmoins bon de savoir procéder de différentes manières, cela vous sera utile parfois.

Voici le code auquel vous pourriez arriver :

```
chaîne = "Salut"
i = 0 # On appelle l'indice 'i' par convention
while i < len(chaîne):
    print(chaîne[i]) # On affiche le caractère à chaque tour
    ↪  de boucle
    i += 1
```

N'oubliez pas d'incrémenter `i`, sinon vous aurez quelques surprises.

Si vous essayez d'accéder à un indice qui n'existe pas (par exemple 25 alors que votre chaîne ne fait que 20 caractères de longueur), Python lèvera une exception de type `IndexError`.

Enfin, une dernière petite chose : vous ne pouvez changer les lettres de la chaîne en utilisant les indices.

```
>>> mot = "lac"
>>> mot[0] = "b" # On veut remplacer 'l' par 'b'
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

Python n'est pas content. Il ne veut pas que vous utilisiez les indices pour modifier des caractères de la chaîne. Pour ce faire, il va falloir utiliser la sélection.

Sélection de chaînes

Nous allons voir comment sélectionner une partie de la chaîne. Si je souhaite, par exemple, sélectionner les deux premières lettres de la chaîne, je procéderai comme dans l'exemple ci-dessous.

```
>>> présentation = "salut"
>>> présentation[0:2] # On sélectionne les deux premières
 ↪  lettres
'sa'
>>> présentation[2:len(présentation)] # On sélectionne la
 ↪  chaîne sauf les deux premières lettres
'lut'
>>>
```

La sélection consiste donc à extraire une partie de la chaîne. Cette opération renvoie le morceau de la chaîne sélectionné, sans modifier la chaîne d'origine.

Sachez que l'on peut sélectionner du début de la chaîne jusqu'à un indice, ou d'un indice jusqu'à la fin de la chaîne, sans préciser autant d'informations que dans nos exemples. Python comprend très bien si on sous-entend certaines informations.

```
>>> présentation[:2] # Du début jusqu'à la troisième lettre
 ↪  non comprise
'sa'
>>> présentation[2:] # De la troisième lettre (comprise) à la
 ↪  fin
'lut'
>>>
```

Maintenant, nous pouvons reprendre notre exemple de tout à l'heure pour constituer une nouvelle chaîne, en remplaçant une lettre par une autre :

```
>>> mot = "lac"
>>> mot = "b" + mot[1:]
>>> print(mot)
bac
>>>
```

Voilà !

Cela reste assez peu intuitif, non ?

Pour remplacer des lettres, cela paraît un peu lourd en effet. Et d'ailleurs, on s'en sert assez rarement pour cela. Pour rechercher/remplacer, nous avons à notre disposition les méthodes `count`, `find` et `replace`, à savoir « compter », « rechercher » et « remplacer ».

En résumé

— Les variables utilisées jusqu'ici sont en réalité des objets.
— Les types de données utilisés jusqu'ici sont en fait des classes. Chaque objet est modelé sur une classe.
— Chaque classe définit certaines fonctions, appelées méthodes, qui seront accessibles depuis l'objet grâce à `objet.methode(arguments)`.
— On peut directement accéder à un caractère d'une chaîne grâce au code suivant : `chaîne[position_dans_la_chaîne]`.
— Il est tout à fait possible de sélectionner une partie de la chaîne grâce au code suivant : `chaîne[indice_debut:indice_fin]`.

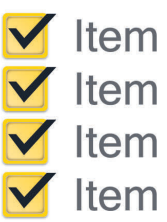
Chapitre 11

Les listes et tuples (1/2)

Difficulté :

J'aurai réussi à vous faire connaître et, j'espère, aimer le langage Python sans vous apprendre les listes. Allons ! Cette époque est révolue. Maintenant que nous commençons à étudier l'objet sous toutes ses formes, je ne vais pas pouvoir garder le secret plus longtemps : il existe des listes en Python. Pour ceux qui ne voient même pas de quoi je parle, vous allez vite vous rendre compte qu'avec les dictionnaires (que nous verrons plus loin), c'est un type, ou plutôt une classe, dont on aura du mal à se passer.

Les listes sont des séquences. En fait, leur nom est plutôt explicite, puisque ce sont des objets capables de regrouper d'autres objets de n'importe quel type. Une liste peut contenir plusieurs nombres entiers (1, 2, 50, 2000 ou plus, peu importe), des flottants, des chaînes de caractères... ou un mélange de ces différents types.



Créons et éditons nos premières listes

D'abord, qu'est-ce qu'une liste ?

En Python, les listes sont des objets qui en contiennent d'autres. Ce sont donc des séquences, comme les chaînes de caractères, mais qui rassemblent n'importe quels objets.

Création de listes

On a deux moyens de créer des listes. Si je vous dis que la classe d'une liste s'appelle, assez logiquement, `list`, vous devriez déjà voir une manière de procéder.

Non ? ...

Vous allez vous habituer à cette syntaxe :

```
>>> ma_liste = list() # On crée une liste vide
>>> type(ma_liste)
<class 'list'>
>>> ma_liste
[]
>>>
```

Là encore, on utilise le nom de la classe comme une fonction pour **instancier** un objet de cette classe.

Quand vous affichez la liste, vous constatez qu'elle est vide. Entre les crochets (qui sont les délimiteurs des listes en Python), il n'y a rien. On peut également utiliser ces crochets pour créer une liste.

```
>>> ma_liste = [] # On crée une liste vide
>>>
```

Cela revient au même, vous pouvez le vérifier. Toutefois, il est également possible d'indiquer les éléments directement à la création de la liste.

```
>>> ma_liste = [1, 2, 3, 4, 5] # Une liste avec cinq objets
>>> print(ma_liste)
[1, 2, 3, 4, 5]
>>>
```

La liste que nous venons de créer compte cinq objets de type `int`. Ils sont classés par ordre croissant. Néanmoins, rien de tout cela n'est obligatoire.

— Le nombre d'éléments n'est pas limité.
— Tous les types d'objet sont autorisés.
— L'ordre de citation des objets est quelconque. Toutefois, la structure d'une liste fait que chaque objet *a sa place* et que l'ordre compte.

```
>>> ma_liste = [1, 3.5, "une chaîne", []]
>>>
```

Nous avons créé ici une liste contenant quatre objets de types différents : un entier, un flottant, une chaîne de caractères et... une autre liste.

Voyons à présent comment accéder aux éléments d'une liste :

```
>>> ma_liste = ['c', 'f', 'm']
>>> ma_liste[0] # On accède au premier élément de la liste
'c'
>>> ma_liste[2] # Troisième élément
'm'
>>> ma_liste[1] = 'Z' # On remplace 'f' par 'Z'
>>> ma_liste
['c', 'Z', 'm']
>>>
```

Comme vous le constatez, on accède aux éléments d'une liste de la même façon qu'aux caractères d'une chaîne : on indique entre crochets l'indice de l'élément qui nous intéresse.

Contrairement à la classe `str`, `list` vous autorise à remplacer un élément par un autre. Les listes sont en effet des types dits **mutables**.

Insérer des objets dans une liste

On dispose de plusieurs méthodes, définies dans la classe `list`, pour ajouter des éléments dans une liste.

Ajouter un élément à la fin de la liste

On utilise la méthode `append` pour ajouter un élément à la fin d'une liste.

```
>>> ma_liste = [1, 2, 3]
>>> ma_liste.append(56) # On ajoute 56 à la fin de la liste
>>> ma_liste
[1, 2, 3, 56]
>>>
```

C'est assez simple non ? On passe en paramètre de la méthode `append` l'objet que l'on souhaite ajouter à la fin de la liste.

La méthode `append`, comme beaucoup de méthodes de listes, travaille directement sur l'objet et ne renvoie donc rien !

Ceci est extrêmement important. Dans le chapitre précédent, nous avons vu qu'aucune des méthodes de chaînes ne modifie l'objet d'origine, mais qu'elles renvoient toutes un nouvel objet, qui est la chaîne modifiée. Ici, c'est le contraire : les méthodes de listes ne renvoient rien mais modifient l'objet d'origine.

```
>>> chaîne1 = "une petite phrase"
>>> chaîne2 = chaîne1.upper() # On met en majuscules chaîne1
>>> chaîne1                   # On affiche la chaîne
 ↪  d'origine
'une petite phrase'
>>> # Elle n'a pas été modifiée par la méthode upper
>>> chaîne2                   # On affiche chaîne2
'UNE PETITE PHRASE'
>>> # C'est chaîne2 qui contient les majuscules
... # Voyons pour les listes à présent
... liste1 = [1, 5.5, 18]
>>> liste2 = liste1.append(-15) # On ajoute -15 à liste1
>>> liste1                      # On affiche liste1
[1, 5.5, 18, -15]
>>> # Cette fois, l'appel de la méthode a modifié l'objet
 ↪  d'origine (liste1)
... # Voyons ce que contient liste2
... liste2
>>> # Rien ? Vérifions avec print
... print(liste2)
None
>>>
```

Je vais expliquer les dernières lignes. Tout d'abord, il faut que vous fassiez bien la différence entre les méthodes de chaînes, où l'objet d'origine n'est jamais modifié et qui renvoient un nouvel objet, et les méthodes de listes, qui ne renvoient rien mais modifient l'objet d'origine.

J'ai dit que les méthodes de listes ne renvoient rien. On va pourtant essayer de « capturer » la valeur de retour dans `liste2`. Quand on essaye d'afficher la valeur de `liste2` par saisie directe, on n'obtient rien. Il faut l'afficher avec `print` pour savoir ce qu'elle contient : `None`. C'est l'objet vide de Python. En réalité, quand une fonction ne renvoie rien, elle renvoie `None`. Vous retrouverez peut-être cette valeur de temps à autre, ne soyez donc pas surpris.

Insérer un élément dans la liste

Nous allons passer assez rapidement sur cette seconde méthode. On peut, très simplement, insérer un objet dans une liste, à l'endroit voulu, avec la méthode `insert`.

```
>>> ma_liste = ['a', 'b', 'd', 'e']
>>> ma_liste.insert(2, 'c') # On insère 'c' à l'indice 2
```

```
>>> print(ma_liste)
['a', 'b', 'c', 'd', 'e']
```

Quand on demande d'insérer `c` à l'indice 2, la méthode va décaler les objets d'indice supérieur ou égal à 2. `c` va donc s'intercaler entre `b` et `d`.

Concaténation de listes

On peut également agrandir des listes en les concaténant avec d'autres.

```
>>> ma_liste1 = [3, 4, 5]
>>> ma_liste2 = [8, 9, 10]
>>> ma_liste1.extend(ma_liste2) # On insère ma_liste2 à la
 ↪  fin de ma_liste1
>>> print(ma_liste1)
[3, 4, 5, 8, 9, 10]
>>> ma_liste1 = [3, 4, 5]
>>> ma_liste1 + ma_liste2
[3, 4, 5, 8, 9, 10]
>>> ma_liste1 += ma_liste2 # Identique à extend
>>> print(ma_liste1)
[3, 4, 5, 8, 9, 10]
>>>
```

Voici les différentes façons de concaténer des listes. Vous remarquez l'opérateur `+` qui concatène deux listes entre elles et renvoie le résultat. On peut utiliser `+=` assez logiquement pour étendre une liste. Cette façon de faire revient au même qu'utiliser la méthode `extend`.

Suppression d'éléments d'une liste

Nous allons découvrir rapidement comment supprimer des éléments d'une liste, avant d'apprendre à les parcourir. Vous constaterez vite que c'est assez simple. Voyons deux méthodes pour supprimer des éléments d'une liste :

— le mot-clé `del` ;
— la méthode `remove`.

Le mot-clé `del`

C'est un des mots-clés de Python, que j'aurais pu vous montrer plus tôt. Toutefois, les applications de `del` me semblaient assez peu pratiques avant d'aborder les listes.

`del` (abréviation de *delete*) signifie « supprimer » en anglais. Son utilisation est des plus simple : `del variable_a_supprimer`. Voyons un exemple.

```
>>> variable = 34
>>> variable
```

```
34
>>> del variable
>>> variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'variable' is not defined
>>>
```

Comme vous le voyez, après l'utilisation de `del`, la variable n'existe plus. Python l'efface tout simplement. Cependant, `del` sert également pour supprimer des éléments d'une séquence, comme une liste.

```
>>> ma_liste = [-5, -2, 1, 4, 7, 10]
>>> del ma_liste[0] # On supprime le premier élément de la
 ↪  liste
>>> ma_liste
[-2, 1, 4, 7, 10]
>>> del ma_liste[2] # On supprime le troisième élément de la
 ↪  liste
>>> ma_liste
[-2, 1, 7, 10]
>>>
```

La méthode `remove`

On peut aussi supprimer des éléments de la liste grâce à la méthode `remove` qui prend en paramètre, non pas l'indice de l'élément à supprimer, mais l'élément lui-même.

```
>>> ma_liste = [31, 32, 33, 34, 35]
>>> ma_liste.remove(32)
>>> ma_liste
[31, 33, 34, 35]
>>>
```

La méthode `remove` parcourt la liste et en retire l'élément que vous lui passez en paramètre. C'est une autre façon de procéder et vous appliquerez `del` ou `remove` en fonction de la situation.

La méthode `remove` ne retire que la première occurrence de la valeur trouvée dans la liste !

Notez au passage que le mot-clé `del` n'est pas une méthode de liste. Il s'agit d'une fonctionnalité de Python qu'on retrouve dans la plupart des objets conteneurs, tels que les listes que nous venons de voir, ou les dictionnaires que nous verrons plus tard. D'ailleurs, `del` sert plus généralement à supprimer non seulement des éléments d'une séquence mais aussi, comme nous l'avons vu, des variables.

Le parcours de listes

Vous avez déjà dû vous faire une idée des méthodes pour parcourir une liste. Je vais passer brièvement dessus ; vous ne verrez rien de nouveau ni, je l'espère, de très surprenant.

```
>>> ma_liste = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> i = 0 # Notre indice pour la boucle while
>>> while i < len(ma_liste):
...     print(ma_liste[i])
...     i += 1 # On incrémente i, ne pas oublier !
...
a
b
c
d
e
f
g
h
>>> # Cette méthode est cependant préférable
... for elt in ma_liste: # elt va prendre les valeurs
 ↪  successives des éléments de ma_liste
...     print(elt)
...
a
b
c
d
e
f
g
h
>>>
```

Il s'agit des mêmes méthodes de parcours que celles présentées au chapitre précédent. Nous allons cependant aller un peu plus loin.

La fonction `enumerate`

Les deux méthodes précédentes possèdent toutes deux des inconvénients :

— La boucle `while` est plus longue à écrire, moins intuitive et elle est perméable aux boucles infinies, si l'on oublie d'incrémenter la variable servant de compteur.
— La boucle `for` se contente de parcourir la liste en capturant les éléments dans une variable, sans qu'on sache où ils sont dans la liste.

C'est vrai dans le cas que nous venons de voir. Certains codeurs vont combiner les deux méthodes pour plus de flexibilité mais, très souvent, le code obtenu est moins lisible. Heureusement, les développeurs de Python ont pensé à nous.

```
>>> ma_liste = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> for i, elt in enumerate(ma_liste):
...     print(f"À l'indice {i} se trouve {elt}.")
...
À l'indice 0 se trouve a.
À l'indice 1 se trouve b.
À l'indice 2 se trouve c.
À l'indice 3 se trouve d.
À l'indice 4 se trouve e.
À l'indice 5 se trouve f.
À l'indice 6 se trouve g.
À l'indice 7 se trouve h.
>>>
```

Pas de panique !

Nous avons ici une boucle `for` un peu surprenante. Entre `for` et `in`, nous trouvons deux variables, séparées par une virgule.

En fait, `enumerate` prend en paramètre une liste et renvoie un objet qui peut être associé à une liste contenant deux valeurs par élément : l'indice et l'élément de la liste parcouru.

Ce n'est sans doute pas encore très clair. Essayons d'afficher cela un peu mieux :

```
>>> for elt in enumerate(ma_liste):
...     print(elt)
...
(0, 'a')
(1, 'b')
(2, 'c')
(3, 'd')
(4, 'e')
(5, 'f')
(6, 'g')
(7, 'h')
>>>
```

Quand on parcourt chaque élément de l'objet renvoyé par `enumerate`, on voit des **tuples** qui contiennent deux éléments : d'abord l'indice, puis l'objet se trouvant à cet indice dans la liste passée en argument à la fonction `enumerate`.

Les tuples sont des séquences, assez semblables aux listes, sauf qu'on ne peut modifier un tuple après qu'il a été créé. Cela signifie qu'on définit le contenu d'un tuple (les objets qu'il doit contenir) lors de sa création, mais qu'on ne peut en ajouter ou en retirer par la suite.

Si les parenthèses vous déconcertent trop, imaginez des crochets à la place ; dans cet exemple, cela revient au même.

Avec `enumerate`, on capture l'indice et l'élément dans deux variables distinctes. Voyons un autre exemple pour comprendre ce mécanisme :

```
>>> autre_liste = [
...     [1, 'a'],
...     [4, 'd'],
...     [7, 'g'],
...     [26, 'z'],
... ] # J'ai étalé la liste sur plusieurs lignes
>>> for nb, lettre in autre_liste:
...     print(f"La lettre {lettre} est la {nb}e de
 ↪  l'alphabet.")
...
La lettre a est la 1e de l'alphabet.
La lettre d est la 4e de l'alphabet.
La lettre g est la 7e de l'alphabet.
La lettre z est la 26e de l'alphabet.
>>>
```

J'espère que c'est assez clair dans votre esprit. Dans le cas contraire, décomposez ces exemples ; le déclic devrait se faire.

On écrit ici la définition de la liste sur plusieurs lignes pour des raisons de lisibilité. On n'est pas obligé de mettre des barres obliques inverses « \ » en fin de ligne car, tant que Python ne trouve pas de crochet fermant la liste, il continue d'attendre sans interpréter la ligne. Vous pouvez d'ailleurs le constater avec les points qui remplacent les chevrons au début de la ligne, tant que la liste n'a pas été refermée.

Quand on travaille sur une liste que l'on parcourt en même temps, on peut se retrouver face à des erreurs assez étranges, qui paraissent souvent incompréhensibles au début.

Par exemple, on peut être confronté à des exceptions `IndexError` si on tente de supprimer certains éléments d'une liste en la parcourant.

Nous verrons au prochain chapitre comment programmer cela proprement ; pour l'heure, il suffit de vous méfier d'un parcours qui modifie une liste, surtout sa structure. D'une façon générale, évitez de parcourir une liste dont la taille évolue en même temps.

Les listes et annotations de type

Vous souvenez-vous des annotations de type ? Quand nous avons appris à créer des fonctions, nous avons passé un moment sur les annotations de type : cette notation facultative qui permet d'indiquer le type d'un paramètre de la fonction.

```
def fonction(param1: str, param2: int) -> float:
```

Pour des fonctions qui ont besoin de listes, comment faire ? Disons que nous avons créé une fonction, `somme`, qui ajoute les éléments d'une liste et retourne la somme obtenue.

Votre premier essai serait sans doute assez semblable à ce code :

```
def somme(liste):
    somme = 0
    for elt in liste:
        somme += elt
    return somme
```

Comment indiquer que notre paramètre `liste` comprend une liste ? Vous l'aurez sans doute deviné :

```
def somme(liste: list):
```

Est-ce que notre fonction accepte une liste contenant n'importe quel argument ? Elle fonctionne bien si la liste comprend des entiers ou des nombres flottants. En revanche, si elle comprend des chaînes de caractères, par exemple, il y a problème.

Il nous faudrait donc pouvoir décrire le type des éléments attendus dans la liste. C'est heureusement possible :

```
def somme(liste: list[int]):
```

On place ici entre crochets le type des éléments attendus. On indique que notre liste doit contenir des entiers. Hum... et des flottants ?

```
def somme(liste: list[int | float]):
```

N'oublions pas le type de retour :

```
def somme(liste: list[int | float]) -> int | float:
```

Ouf ! Cela nous donne une ligne plus longue. Là encore, avec de l'habitude, elle paraît assez claire et on sait en un coup d'œil ce que notre liste peut contenir.

Les tuples

Nous avons brièvement cité les **tuples** un peu plus haut, grâce à la fonction `enumerate`. Les tuples sont des listes immuables, qu'on ne peut modifier. En fait, vous allez constater que nous utilisons depuis longtemps des tuples sans nous en rendre compte.

Un tuple se définit comme une liste, sauf qu'on utilise comme délimiteur des parenthèses au lieu des crochets.

```
tuple_vide = ()
tuple_non_vide = (1,)
tuple_non_vide = (1, 3, 5)
```

À la différence des listes, les tuples, une fois créés, ne sont pas modifiables : il est impossible d'y ajouter ou d'en retirer des objets.

Une petite subtilité ici : si on veut créer un tuple contenant un unique élément, on doit quand même ajouter une virgule après celui-ci. Sinon, Python va automatiquement supprimer les parenthèses et on se retrouvera avec une variable lambda et non un tuple contenant cette variable.

À quoi cela sert-il ?

Il est assez rare que l'on travaille directement sur des tuples. Cela vous paraît peut-être encore assez abstrait, mais il est parfois utile de travailler sur des données sans avoir le droit de les modifier.

En attendant, voyons plutôt les cas où nous avons utilisé des tuples sans le savoir.

Affectation multiple

Tous les cas que nous allons voir sont des affectations multiples. Vous vous souvenez ?

```
>>> a, b = 3, 4
>>> a
3
>>> b
4
>>>
```

On a également utilisé cette syntaxe pour permuter deux variables. Eh bien, elle passe par des tuples qui ne sont pas déclarés explicitement. Vous pourriez écrire ce qui suit :

```
>>> (a, b) = (3, 4)
>>>
```

Quand Python trouve plusieurs variables ou valeurs séparées par des virgules et sans délimiteur, il les range dans des tuples. Dans le premier exemple, les parenthèses sont sous-entendues et Python comprend ce qu'il doit faire.

Une fonction renvoyant plusieurs valeurs

Nous ne l'avons pas vu jusqu'ici, mais une fonction peut renvoyer deux valeurs ou même plus :

```
def decomposer(entier, divise_par):
    """Cette fonction retourne la partie entière et le reste
    ↪  de entier / divise_par"""

    p_e = entier // divise_par
    reste = entier % divise_par
    return p_e, reste
```

Dans cet exemple, il est ensuite possible de capturer la partie entière et le reste dans deux variables, au retour de la fonction :

```
>>> partie_entiere, reste = decomposer(20, 3)
>>> partie_entiere
6
>>> reste
2
>>>
```

Là encore, on passe par des tuples sans que ce soit indiqué explicitement à Python. Si vous écrivez `retour = decomposer(20, 3)`, vous capturez un tuple contenant deux éléments : la partie entière et le reste de 20 divisé par 3.

Nous verrons plus loin d'autres exemples de tuples et d'autres utilisations. Je pense que cela suffit pour cette fois.

En résumé

- Une liste est une séquence mutable contenant plusieurs autres objets.
- Une liste se construit ainsi : `liste = [element1, element2, elementN]`.
- On peut insérer des éléments dans une liste à l'aide des méthodes `append`, `insert` et `extend`.
- On supprime des éléments d'une liste grâce au mot-clé `del` ou à la méthode `remove`.
- Un tuple est une séquence contenant des objets. À la différence de la liste, le tuple n'est pas modifiable une fois créé.

Chapitre 12

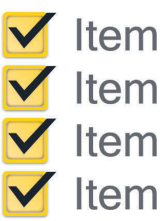
Les listes et tuples (2/2)

Difficulté : 

Les listes sont très utilisées en Python. Elles sont liées à de nombreuses fonctionnalités, dont certaines plutôt complexes. Aussi ai-je préféré scinder l'approche des listes en deux chapitres. Vous allez voir dans celui-ci quelques fonctionnalités qui ne s'appliquent qu'aux listes et aux tuples et qui vous seront extrêmement utiles. Je vous conseille donc, avant tout, d'être bien à l'aise avec les listes et leur création, parcours, édition, suppression...

D'autre part, comme pour la plupart des sujets abordés, je ne peux faire un tour d'horizon exhaustif de toutes les fonctionnalités de chaque objet présenté. Je vous invite donc à lire la documentation, en tapant `help(list)`, pour accéder à une liste exhaustive des méthodes, ou en allant sur la page suivante : `https://docs.python.org/fr/3/tutorial/datastructures.html`

C'est parti !



Entre chaînes et listes

Nous allons étudier un moyen de transformer des chaînes en listes et réciproquement.

Il est assez surprenant, de prime abord, qu'une conversion soit possible entre ces deux types qui sont tout de même assez différents. En fait, comme on va le voir, il ne s'agit pas réellement d'une conversion. Il va être difficile de démontrer l'utilité de cette fonctionnalité tout de suite ; mieux valent quelques exemples.

Des chaînes aux listes

Pour « convertir » une chaîne en liste, on va utiliser une méthode nommée `split` (« éclater » en anglais). Elle prend un paramètre qui est une autre chaîne, souvent d'un seul caractère, définissant comment on va découper notre chaîne initiale.

C'est un peu compliqué et cela paraît très tordu… mais regardez plutôt :

```
>>> ma_chaîne = "Bonjour à tous"
>>> ma_chaîne.split(" ")
['Bonjour', 'à', 'tous']
>>>
```

On passe en paramètre de la méthode `split` une chaîne contenant un unique espace. La méthode renvoie une liste contenant les trois mots de notre petite phrase. Chaque mot se trouve dans une case de la liste.

C'est assez simple en fait : quand on appelle la méthode `split`, celle-ci découpe la chaîne en fonction du paramètre donné. Ici, la première case de la liste va donc du début de la chaîne au premier espace (non inclus), la deuxième case va du premier espace au second, et ainsi de suite jusqu'à la fin de la chaîne.

Sachez que `split` possède un paramètre par défaut, un code qui représente les espaces, les tabulations et les sauts de ligne. Donc, si vous écrivez `ma_chaîne.split()`, cela revient ici au même.

Des listes aux chaînes

Voyons l'inverse à présent, c'est-à-dire une liste contenant plusieurs chaînes de caractères que l'on souhaite fusionner en une seule. On utilise la méthode de chaîne `join` (« joindre » en anglais). Sa syntaxe est un peu surprenante :

```
>>> ma_liste = ['Bonjour', 'à', 'tous']
>>> " ".join(ma_liste)
'Bonjour à tous'
>>>
```

En paramètre de la méthode `join`, on passe la liste des chaînes que l'on souhaite « ressouder ». La méthode va travailler sur l'objet qui l'appelle, ici une chaîne de

caractères contenant un unique espace. Elle va insérer cette chaîne entre chaque paire de chaînes de la liste, ce qui au final nous donne la chaîne de départ, « Bonjour à tous ».

N'aurait-il pas été plus simple ou plus logique de créer une méthode de liste prenant en paramètre la chaîne faisant la jonction ?

Ce choix est en effet contesté mais, pour ma part, je ne trancherai pas. Le fait est que c'est cette méthode qui a été choisie et, avec un peu d'habitude, on arrive à bien lire le résultat obtenu. D'ailleurs, nous allons voir comment appliquer concrètement ces deux méthodes.

Une application pratique

Admettons que nous ayons un réel dont nous souhaitons afficher la partie entière et uniquement les trois premiers chiffres de la partie flottante. Autrement dit, si on a un réel tel que « 3.999999999999998 », on souhaite obtenir « 3.999 » comme résultat. D'ailleurs, ce serait plus joli si on remplaçait le point décimal par la virgule, à laquelle nous sommes plus habitués.

Là encore, je vous invite à coder ce petit exercice par vous-mêmes. On part du principe que la valeur de retour de la fonction chargée de la pseudo-conversion est une chaîne de caractères. Voici quelques exemples d'utilisation de la fonction que vous devriez coder :

```
>>> afficher_flottant(3.99999999999998)
'3,999'
>>> afficher_flottant(1.5)
'1,5'
>>>
```

Voici la correction que je vous propose :

```
def afficher_flottant(flottant: float) -> str:
    """Fonction prenant en paramètre un flottant et renvoyant
    ↪ une chaîne de caractères représentant la troncature de
    ↪ ce nombre. La partie flottante doit avoir une longueur
    ↪ maximum de 3 caractères.

    De plus, on va remplacer le point décimal par la virgule.

    """
    if not isinstance(flottant, float):
        raise TypeError("Le paramètre attendu doit être un
        ↪ flottant")
    flottant = str(flottant)
    partie_entiere, partie_flottante = flottant.split(".")
    # La partie entière n'est pas à modifier
```

```
    # Seule la partie flottante doit être tronquée
    return ",".join([partie_entiere, partie_flottante[:3]])
```

Note : la méthode `isinstance` ici nous permet de vérifier que la variable `flottant` est bien de type `float`.

En s'assurant que le type passé en paramètre est bien un flottant, on garantit qu'il n'y aura pas d'erreur lors du fractionnement de la chaîne. On est sûr qu'il y aura forcément une partie entière et une partie flottante séparées par un point, même si la partie flottante n'est constituée que d'un 0. Si vous n'y êtes pas arrivés par vous-mêmes, étudiez bien cette solution ; elle n'est pas forcément évidente au premier coup d'œil. On fait intervenir un certain nombre de mécanismes que vous avez découverts il y a peu ; tâchez de bien les comprendre.

Les listes et paramètres de fonctions

Nous allons droit vers une fonctionnalité des plus intéressantes, qui fait une partie de la puissance de Python. Nous allons étudier un cas assez particulier avant de généraliser : les fonctions dont le nombre de paramètres est inconnu.

Notez malgré tout que ce point est assez délicat. Si vous n'arrivez pas bien à le comprendre, laissez cette section de côté, cela ne vous pénalisera pas.

Les fonctions dont on ne connaît pas à l'avance le nombre de paramètres

Vous devriez tout de suite penser à la fonction `print` : on lui passe une liste de paramètres qu'elle va afficher, dans l'ordre où ils sont placés, séparés par un espace (ou tout autre délimiteur choisi).

Vous n'allez peut-être pas trouver d'applications de cette fonctionnalité dans l'immédiat mais, tôt ou tard, cela arrivera. La syntaxe est tellement simple que c'en est déconcertant :

```
def fonction(*parametres):
```

On place une étoile `*` devant le nom du paramètre qui accueillera la liste des arguments. Voyons plus précisément comment cela se présente :

```
>>> def fonction_inconnue(*parametres):
...     """Test d'une fonction pouvant être appelée
...     avec un nombre variable de paramètres"""
...     print(f"J'ai reçu : {parametres}.")
...
>>> fonction_inconnue() # On appelle la fonction sans
 ↪  paramètre
J'ai reçu : ().
>>> fonction_inconnue(33)
J'ai reçu : (33,).
>>> fonction_inconnue('a', 'e', 'f')
J'ai reçu : ('a', 'e', 'f').
>>> var = 3.5
```

```
>>> fonction_inconnue(var, [4], "...")
J'ai reçu : (3.5, [4], '...').
>>>
```

Je pense que cela suffit. Comme vous le constatez, on peut appeler la `fonction_inconnue` avec un nombre indéterminé de paramètres, allant de 0 à l'infini (enfin, théoriquement). Le fait de préciser une étoile `*` devant le nom du paramètre fait que Python va placer tous les paramètres de la fonction dans un **tuple**, que l'on peut ensuite traiter comme on le souhaite.

Et les paramètres nommés dans l'histoire ? Comment sont-ils insérés dans le tuple ?

Ils ne le sont pas. Si vous tapez `fonction_inconnue(couleur="rouge")`, vous provoquez une erreur : `fonction_inconnue() got an unexpected keyword argument 'couleur'`. Nous verrons au prochain chapitre comment capturer ces paramètres nommés. Vous pouvez bien entendu définir une fonction avec plusieurs paramètres qui doivent être fournis quoi qu'il arrive, suivis d'une liste de paramètres variables :

```
def fonction_inconnue(nom, prénom, *commentaires):
```

Dans cet exemple de définition de fonction, vous devez impérativement préciser un nom et un prénom. Ensuite, vous mettez ce que vous voulez en commentaire, aucun paramètre, un, deux...

Si on définit une liste variable de paramètres, elle doit se trouver après la liste des paramètres standard.

Au fond, cela est évident. Vous ne pouvez avoir une définition de fonction comme `def fonction_inconnue(*parametres, nom, prénom)`. En revanche, si vous souhaitez transmettre des paramètres nommés, il faut les citer après cette liste. Les paramètres nommés sont un peu une exception puisqu'ils ne figureront de toute façon pas dans le tuple obtenu. Voyons par exemple la définition de la fonction `print` :

```
print(value, ..., sep=' ', end='\n', file=sys.stdout,
↪   flush=False)
```

Ne nous occupons pas des derniers paramètre pour l'heure.

D'où viennent ces points de suspension dans les paramètres ?

En fait, il s'agit d'un affichage un peu plus agréable. Si on veut réellement avoir la définition en code Python, on retombera plutôt sur :

```
def print(*values, sep=' ', end='\n', file=sys.stdout,
↪   flush=False):
```

Petit exercice : écrivez une fonction `afficher` identique à `print`, c'est-à-dire prenant un nombre indéterminé de paramètres, les affichant en les séparant à l'aide du paramètre nommé `sep` et terminant l'affichage par la variable `fin`. Notre fonction `afficher` ne comptera pas de paramètre `file`. En outre, elle devra passer par `print` pour afficher (on ne connaît pas encore d'autres façons de procéder). La seule contrainte est que l'appel à `print` ne doit compter qu'un seul paramètre non nommé. Autrement dit, avant l'appel à `print`, la chaîne devra avoir été déjà formatée, prête à l'affichage.

Pour que ce soit plus clair, voici la définition de la fonction, ainsi que la `docstring` que j'ai écrite :

```
def afficher(*parametres, sep=' ', fin='\n'):
    """Fonction chargée de reproduire le comportement de
    ↪  print.

    Elle doit finir par faire appel à print pour afficher
    le résultat. Mais les paramètres devront déjà avoir été
    formatés. On doit passer à print une unique chaîne,
    en lui spécifiant de ne rien mettre à la fin :

    print(chaîne, end='')

    """
    # Les paramètres sont sous la forme d'un tuple
    # Or on a besoin de les convertir
    # Mais on ne peut pas modifier un tuple
    # On a plusieurs possibilités ; ici je choisis de
    # convertir le tuple en liste
    parametres = list(parametres)
    # On va commencer par convertir toutes les valeurs en
    # chaîne sinon on va avoir quelques problèmes lors du join
    for i, parametre in enumerate(parametres):
        parametres[i] = str(parametre)
    # La liste des paramètres ne contient plus que des chaînes
    # de caractères. À présent on va constituer la chaîne
    # finale
    chaîne = sep.join(parametres)
    # On ajoute le paramètre fin à la fin de la chaîne
    chaîne += fin
    # On affiche l'ensemble
    print(chaîne, end='')
```

J'espère que ce n'était pas trop difficile et que, si vous avez commis des erreurs, vous avez pu les comprendre.

Ce n'est pas du tout grave si vous avez réussi à coder cette fonction d'une manière différente. **En programmation, il n'y a pas qu'une solution ; il y a des solutions.**

Transformer une liste en paramètres de fonction

C'est peut-être un peu moins fréquent, mais vous devez connaître ce mécanisme puisqu'il complète parfaitement le premier. Si vous avez un tuple ou une liste contenant des paramètres qui doivent être passés à une fonction, vous pouvez très simplement les transformer en paramètres lors de l'appel. Le seul problème c'est que, côté démonstration, je me vois un peu limité.

```
>>> liste_des_paramètres = [1, 4, 9, 16, 25, 36]
>>> print(*liste_des_paramètres)
1 4 9 16 25 36
>>>
```

Ce n'est pas bien spectaculaire et pourtant c'est une fonctionnalité très puissante du langage. Là, on a une liste contenant des valeurs et on la transforme en une liste de paramètres de la fonction `print`. Donc, au lieu d'afficher la liste proprement dite, on affiche tous les nombres, séparés par des espaces. C'est exactement comme si vous aviez écrit `print(1, 4, 9, 16, 25, 36)`.

Quel en est l'intérêt ? Cela ne change pas grand-chose et il est rare que l'on capture les paramètres d'une fonction dans une liste, non ?

Oui je vous l'accorde. Ici l'intérêt ne saute pas aux yeux. Pourtant, vous pourrez tomber sur des applications où les fonctions sont utilisées sans savoir quels paramètres elles attendent réellement. Si on ne connaît pas la fonction que l'on appelle, c'est très pratique. Là encore, vous découvrirez cela dans les chapitres suivants ou dans certains projets. Essayez de garder à l'esprit ce mécanisme de transformation.

On utilise une étoile * dans les deux cas. Si c'est dans une définition de fonction, cela signifie que les paramètres fournis non attendus lors de l'appel seront capturés dans la variable, sous la forme d'un tuple. Si c'est dans un appel de fonction, au contraire, cela signifie que la variable sera décomposée en plusieurs paramètres envoyés à la fonction.

J'espère que vous êtes encore en forme, on attaque deux points que je considère comme les plus durs de ce chapitre, mais aussi les plus intéressants. Gardez les yeux ouverts !

Match et les listes

Vous vous en souvenez, on a vu `match` et `case`, deux mots-clés sans doute exotiques qui ne vous ont probablement pas semblé très intéressants. Cependant, je vous l'ai promis : ces mots-clés sont très utiles et bien plus puissants qu'une condition. Éh bien voilà, l'heure de voir toute la puissance de `match` a sonné !

Nous avons beaucoup travaillé sur des listes contenant des informations du même type. Comme vous l'avez vu, ce n'est pas du tout une obligation. Cette fonctionnalité peut être extrêmement utile pour représenter des informations plus détaillées.

Admettons que vous travailliez sur un agenda électronique, une application qui permette de stocker des rendez-vous, tâches et anniversaires. Nous pourrions conserver ces trois informations dans des listes. Toutefois... un rappel pour un anniversaire n'a pas beaucoup de caractéristiques, alors qu'un rendez-vous commence à une heure particulière et pourrait se terminer à une heure précise. Une tâche est plutôt semblable à un anniversaire, sauf qu'on ne fait généralement pas la même chose tous les ans. Voici quelques exemples :

```
anniversaire = ["anniversaire", "Étienne"]
tâche = ["tâche", "ranger la cuisine"]
rendez_vous = ["rendez-vous", "10:00", "10:30", "Réunion
↪  professionnelle"]
```

Dans cet exemple, on a trois listes différentes. Les deux premières contiennent deux éléments, mais la troisième en contient quatre. Jusqu'ici, pas de difficulté.

Notez cependant que le premier élément de la liste est toujours une chaîne de caractères, décrivant le contenu de la liste : la chaîne peut contenir `anniversaire`, `tâche` ou `rendez-vous`. C'est un moyen de conserver les différentes activités dans notre agenda (nous verrons bien d'autres moyens en créant nos propres classes).

Admettons que l'on veuille garder toutes ces informations dans une liste (disons, une liste des choses à faire pour aujourd'hui) :

```
aujourd_hui = [anniversaire, tâche, rendez_vous]
```

Là encore, rien d'extraordinaire. On crée une liste qui va contenir des listes, chacune décrivant une action à faire aujourd'hui.

Maintenant, comment faire si l'on veut parcourir la liste et afficher des détails particuliers si l'élément de la liste est un anniversaire, une tâche ou un rendez-vous ?

Vous pourriez essayer, vous en savez assez, mais nous allons ici voir quelque chose de bien plus facile à écrire au final.

Oui, `match` peut nous rendre ce service. En fait il devient vraiment intéressant dans ce genre de contexte, car son rôle principal est de faire correspondre et extraire des informations. Il devient très utile pour des séquences. Voyez le code pour lire un anniversaire :

```
>>> match anniversaire: # On travaille sur la variable
↪  anniversaire pour l'exemple
...     case ["anniversaire", personne]:
...         # La variable personne contient le second élément
↪  de la liste.
...         print(f"Pensez à souhaiter un joyeux anniversaire
↪  à {personne} !")
...
Pensez à souhaiter un joyeux anniversaire à Étienne !
>>>
```

Voici... notre première correspondance et extraction ! Prenons le temps d'examiner ce code :

— D'abord, on travaille sur la variable `anniversaire`. On sait que cette variable contient une liste de deux éléments : le premier est la chaîne de caractères "anniversaire", le second est le nom de la personne.
— Notre premier (et unique) `case` est intéressant : entre crochets, on précise une chaîne de caractères, "anniversaire", puis un nom... `personne` ? D'où sort ce `personne` ?
— Quand Python arrive sur ce `case`, il va créer une variable `personne` ici qui pourra être utilisée dans la branche et contiendra le second élément de la liste.

La syntaxe est relativement légère mais peut sembler un peu magique. Il est important de comprendre l'enchaînement des étapes ici. Quand Python arrive à la ligne du `case`, il sait qu'il doit comparer la valeur de ce dernier et ce qu'on lui a donné en `match` (c'est-à-dire la liste `anniversaire`). La valeur du `case` est aussi une liste. Elle contient deux éléments.

— Python commence par s'assurer que la variable comparée (`anniversaire`) et le cas proposé (`["anniversaire", personne]` sont bien du même type. Jusqu'ici, c'est le cas (ce sont des listes).
— Il s'assure que ces deux listes font la même taille. C'est le cas, elles contiennent deux éléments.
— Il regarde ensuite le premier élément de la liste comparée ("anniversaire") et le premier élément de la valeur proposée ("anniversaire"). Jusqu'ici, tout correspond.
— Il regarde ensuite le second élément de la liste comparée, qui contient "Étienne". Le second élément de la liste proposée contient simplement `personne` (sans guillemets) que Python interprète comme un nom de variable. Il doit donc créer la variable `personne` qui contient "Étienne" et exécuter le bloc d'instructions à l'intérieur de ce `case`.

Si vous ne comprenez pas l'enchaînement de ces étapes, peut-être l'équivalent en `if` vous semblera-t-il plus clair. Je l'ai un peu simplifié :

```
if ( # C'est une longue condition, on l'écrit sur plusieurs lignes
    isinstance(anniversaire, list) # 'anniversaire' est bien une liste
    and len(anniversaire) == 2 # s'assure qu'elle contient 2 éléments
    and anniversaire[0] == "anniversaire" # compare le premier élément
): # Tout est bon !
    personne = anniversaire[1] # On crée la variable personne
    print(f"Pensez à souhaiter un joyeux anniversaire à {personne} !")
```

En comparaison, notre bloc `match` est bien plus léger et facile à comprendre... et à maintenir !

J'ai un peu simplifié ici, car Python accepterait plusieurs séquences : notre variable `anniversaire` aurait pu contenir un tuple, pas une liste, et notre `case` aurait

fonctionné. Python s'intéresse au contenu, mais il ne va pas autoriser une comparaison entre deux séquences qui n'ont pas un type assez voisin.

Nous avons donc créé un `case` spécial pour gérer un anniversaire. On pourrait procéder de façon similaire pour gérer une tâche ou un rendez-vous :

```
>>> match rendez_vous:
...     case ["tâche", objet]:
...         print(f"Pensez à {objet}.")
...     case ["rendez-vous", début, fin, objet]:
...         print(f"Rendez-vous prévu de {début} à {fin}:
 ↪  {objet}.")
...
Rendez-vous prévu de 10:00 à 10:30: Réunion professionnelle.
>>>
```

Le premier `case` ne devrait pas trop vous surprendre. En revanche, le second est plus complexe : notre liste proposée contient 4 éléments : la chaîne "rendez-vous", suivie de trois variables (`début`, `fin` et `objet`). La capture se fait exactement selon les mêmes termes que pour l'anniversaire que nous avons vu plus haut.

Pourquoi est-ce utile ?

Vous vous souvenez sans doute de notre liste de choses à faire aujourd'hui. Essayons de la parcourir pour afficher un message différent selon que l'élément de liste est un anniversaire, une tâche ou un rendez-vous :

```
>>> for à_faire in aujourd_hui:
...     match à_faire:
...         case ["anniversaire", personne]:
...             print(f"Pensez à souhaiter un joyeux
 ↪  anniversaire à {personne} !")
...         case ["tâche", objet]:
...             print(f"Pensez à {objet}.")
...         case ["rendez-vous", début, fin, objet]:
...             print(f"Rendez-vous prévu de {début} à {fin}:
 ↪  {objet}.")
...
Pensez à souhaiter un joyeux anniversaire à Étienne !
Pensez à ranger la cuisine.
Rendez-vous prévu de 10:00 à 10:30: Réunion d'affaire.
>>>
```

Ici, nous parcourons toute la liste des choses à réaliser aujourd'hui et essayons de faire correspondre chaque élément pour afficher un message différent si c'est un anniversaire, une tâche ou un rendez-vous. Bien entendu, votre liste `aujourd_hui` peut contenir

plusieurs anniversaires, plusieurs tâches et plusieurs rendez-vous, dans l'ordre que vous voulez !

Nous avons donc vu que `match` permet non seulement de comparer, mais aussi d'extraire et de créer des variables qui seront utilisées dans le corps du `case`.

Vous pourriez pousser cet agenda plus loin encore en traitant d'autres types de tâche, ou créer une petite calculette que l'utilisateur pourrait manipuler et qui supporterait les opérations simples (addition, soustraction, multiplication et division). Avec `match` et `case`, bien des cas difficiles sont assez faciles à résoudre, deviennent plus rapides à lire et, dans bien des cas, plus performants à exécuter.

Les compréhensions de liste

Les compréhensions de liste (« *list comprehensions* » en anglais, parfois abrégé *list-comp*) sont un moyen de filtrer ou modifier une liste très simplement. La syntaxe est déconcertante au début mais vous allez voir que c'est très puissant.

Parcours simple

Les **compréhensions de liste** permettent de parcourir une liste en en renvoyant une seconde, modifiée ou filtrée. Pour l'instant, nous allons coder une simple modification.

```
1 >>> liste_origine = [0, 1, 2, 3, 4, 5]
2 >>> [nb * nb for nb in liste_origine]
3 [0, 1, 4, 9, 16, 25]
4 >>>
```

Étudions un peu la ligne 2 de ce code. Comme vous avez pu le deviner, elle signifie en langage plus conventionnel « Élever au carré tous les nombres contenus dans la liste d'origine ». Nous trouvons dans l'ordre, entre les crochets qui sont les délimiteurs d'une instruction de compréhension de liste :

- `nb * nb` : la valeur de retour. Pour l'instant, on ne sait pas ce qu'est la variable `nb` ; on sait juste qu'il faut l'élever au carré. Notez qu'on aurait pu écrire `nb ** 2`, ce qui revient au même.
- `for nb in liste_origine` : voilà d'où vient notre variable `nb`. On reconnaît la syntaxe d'une boucle `for`, sauf qu'on n'est pas habitué à la voir sous cette forme.

Quand Python interprète cette ligne, il parcourt la liste d'origine et élève chaque élément de la liste au carré. Il renvoie ensuite le résultat obtenu, sous la forme d'une liste de la même longueur que celle d'origine. On peut naturellement capturer cette nouvelle liste dans une variable.

Filtrage avec un branchement conditionnel

On peut aussi filtrer une liste de la façon suivante :

```
>>> liste_origine = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> [nb for nb in liste_origine if nb % 2 == 0]
[2, 4, 6, 8, 10]
>>>
```

On ajoute à la fin de l'instruction une condition qui va déterminer quelles valeurs seront transférées dans la nouvelle liste : ici, seulement les valeurs paires. Au final, on se retrouve donc avec une liste deux fois plus petite que celle d'origine.

Mélangeons un peu tout cela

Il est possible de filtrer et modifier une liste assez simplement. Par exemple, on a une liste contenant les quantités de fruits stockées pour un magasin (je ne suis pas sectaire, vous pouvez prendre des hamburgers si vous préférez). Chaque semaine, le magasin va prendre dans le stock une certaine quantité de chaque fruit, pour la mettre en vente. À ce moment, le stock de chaque fruit diminue naturellement. Il est inutile de garder les fruits qu'on n'a plus en stock.

Je vais un peu reformuler. On va avoir une liste simple, qui contiendra des entiers, précisant la quantité de chaque fruit (c'est abstrait, les fruits ne sont pas précisés). On va écrire une compréhension de liste pour diminuer d'une quantité donnée toutes les valeurs de cette liste et on en profite pour retirer celles qui sont inférieures ou égales à 0.

```
>>> qtt_à_retirer = 7 # On retire chaque semaine 7 fruits de
 ↪  chaque sorte
>>> fruits_stockés = [15, 3, 18, 21] # Par exemple 15 pommes,
 ↪  3 melons...
>>> [nb_fruits - qtt_à_retirer for nb_fruits in
 ↪  fruits_stockés if nb_fruits > qtt_à_retirer]
[8, 11, 14]
>>>
```

Comme vous le voyez, le fruit de quantité 3 n'a pas survécu à cette semaine d'achats. Bien sûr, cet exemple n'est pas complet : on n'a aucun moyen fiable d'associer les nombres restants aux fruits. C'est juste un exemple de filtrage et modification d'une liste.

Prenez bien le temps de regarder ces exemples : au début, la syntaxe des compréhensions de liste n'est pas forcément simple. Faites des essais, c'est aussi le meilleur moyen de comprendre.

Nouvelle application concrète

De nouveau, c'est à vous de travailler.

Nous allons en gros reprendre l'exemple précédent, en le modifiant un peu pour qu'il soit plus cohérent. Nous travaillons toujours avec des fruits sauf que, cette fois, nous allons

associer un nom de fruit à la quantité restant en magasin. Nous verrons au prochain chapitre comment le faire avec des dictionnaires ; pour l'instant on va se contenter de listes :

```
>>> inventaire = [
...     ("pommes", 22),
...     ("melons", 4),
...     ("poires", 18),
...     ("fraises", 76),
...     ("prunes", 51),
... ]
>>>
```

Recopiez cette liste. Elle contient des tuples, contenant chacun un couple : le nom du fruit et sa quantité en magasin.

Votre mission est de trier cette liste en fonction de la quantité de chaque fruit. Autrement dit, on doit obtenir quelque chose de similaire à :

```
[
    ("fraises", 76),
    ("prunes", 51),
    ("pommes", 22),
    ("poires", 18),
    ("melons", 4),
]
```

Pour ceux qui n'ont pas eu la curiosité de regarder dans la documentation des listes, je signale à votre attention la méthode `sort` qui permet de trier une liste. Vous pouvez également utiliser la fonction `sorted` qui prend en paramètre la liste à trier (ce n'est pas une méthode de liste, faites attention). `sorted` renvoie la liste triée sans modifier celle d'origine, ce qui est utile dans certaines circonstances, précisément celle-ci. À vous de voir, vous pouvez y arriver par les deux méthodes.

Bien entendu, essayez de coder cet exercice en utilisant les compréhensions de liste.

Je vous donne juste un petit indice : vous ne pouvez trier la liste comme cela, il faut l'inverser (autrement dit, placer la quantité avant le nom du fruit) pour pouvoir ensuite la trier par quantité. Un chapitre entier est consacré au tri en Python ; vous y découvrirez d'autres moyens pour trier plus efficacement. En attendant, essayez de travailler avec ce que vous savez faire.

Voici la correction que je vous propose :

```
# On change le sens de l'inventaire, la quantité avant le nom
inventaire_inversé = [(qtt, nom_fruit) for nom_fruit, qtt in
↪  inventaire]
# On n'a plus qu'à trier dans l'ordre décroissant l'inventaire
↪  inversé
# On reconstitue l'inventaire trié
inventaire = [(nom_fruit, qtt) for qtt, nom_fruit in
↪  sorted(inventaire_inversé, reverse=True)]
```

Cela fonctionne et le traitement a été codé en deux lignes.

Vous pouvez trier l'inventaire inversé avant la reconstitution, si vous trouvez cela plus compréhensible. Il faut privilégier la lisibilité du code.

```
# On change le sens de l'inventaire, la quantité avant le nom
inventaire_inversé = [(qtt, nom_fruit) for nom_fruit,qtt in
 ↪  inventaire]
# On trie l'inventaire inversé dans l'ordre décroissant
inventaire_inversé.sort(reverse=True)
# Et on reconstitue l'inventaire
inventaire = [(nom_fruit, qtt) for qtt, nom_fruit in
 ↪  inventaire_inversé]
```

Faites des essais, entraînez-vous ; vous en aurez sans doute besoin, car la syntaxe n'est pas très simple au début. Et évitez de tomber dans l'extrême aussi : certaines opérations ne sont pas faisables avec les compréhensions de listes ou alors elles sont trop condensées pour être facilement compréhensibles. Dans l'exemple précédent, on aurait très bien pu remplacer nos deux à trois lignes d'instructions par une seule, mais cela aurait été dur à lire. Ne sacrifiez pas la lisibilité pour le simple plaisir de raccourcir votre code.

En résumé

— On découpe une chaîne en fonction d'un séparateur grâce à la méthode `split` de la chaîne.
— On joint des chaînes de caractères contenues dans une liste avec la méthode de chaîne `join` appelée sur le séparateur.
— On crée des fonctions attendant un nombre inconnu de paramètres grâce à la syntaxe `def fonction_inconnue(*parametres):` (les paramètres passés se retrouvent dans le tuple *parametres*).
— On peut utiliser `match` pour filtrer et extraire dans une liste.
— Les *compréhensions de listes* servent à parcourir et filtrer une séquence en en renvoyant une nouvelle.
— La syntaxe pour effectuer un filtrage est la suivante : `nouvelle_séquence = [élément for élément in ancienne_séquence if condition]`.

Chapitre 13

Les dictionnaires

Difficulté :

Maintenant que vous commencez à vous familiariser avec la programmation orientée objet, nous irons un peu plus vite sur les manipulations « classiques » de ce type, pour nous concentrer sur quelques petites spécificités propres aux dictionnaires.

Les dictionnaires sont des objets qui en contiennent d'autres, à l'instar des listes. Cependant, au lieu d'héberger des informations dans un ordre précis, ils associent chaque objet contenu à une clé (la plupart du temps, une chaîne de caractères). Par exemple, un dictionnaire peut contenir un carnet d'adresses et on accède à chaque contact en précisant son nom.

Création et édition de dictionnaires

Le dictionnaire est un type de données extrêmement puissant et pratique. Il se rapproche des listes sur certains points mais, sur beaucoup d'autres, il en diffère totalement. Python utilise ce type pour représenter diverses fonctionnalités : on peut par exemple retrouver les attributs d'un objet grâce à un dictionnaire particulier.

N'anticipons pas. Dans les deux chapitres précédents, nous avons découvert les listes. Les objets de ce type sont des conteneurs, dans lesquels on trouve d'autres objets. Pour accéder à ces derniers, il faut connaître leur position dans la liste, qui se traduit par des entiers, appelés indices, compris entre 0 (inclus) et la taille de la liste (non incluse).

Le dictionnaire est aussi un objet conteneur. Il n'a quant à lui aucune structure ordonnée, à la différence des listes. De plus, pour accéder aux objets contenus dans le dictionnaire, on n'utilise pas nécessairement des indices, mais des **clés** qui peuvent être de bien des types distincts.

Créer un dictionnaire

Voici le nom de la classe sur laquelle se construit un dictionnaire : `dict`. Vous devriez du même coup trouver la première méthode d'instanciation d'un tel objet :

```
>>> mon_dictionnaire = dict()
>>> type(mon_dictionnaire)
<class 'dict'>
>>> mon_dictionnaire
{}
>>> # Du coup, vous devriez trouver la deuxième manière de
 ↪  créer un dictionnaire vide
... mon_dictionnaire = {}
>>> mon_dictionnaire
{}
>>>
```

Les tuples sont délimités par des parenthèses, les listes par des crochets et les dictionnaires par des accolades.

Les accolades ont en fait une application un peu plus large dans Python. Elles sont utilisées pour décrire des dictionnaires et des ensembles. Nous verrons ce second type dans le prochain chapitre.

Voyons comment ajouter des clés et valeurs dans notre dictionnaire vide :

```
>>> mon_dictionnaire = {}
>>> mon_dictionnaire["pseudo"] = "Prolixe"
>>> mon_dictionnaire["mot de passe"] = "*"
>>> mon_dictionnaire
{'pseudo': 'Prolixe', 'mot de passe': '*'}
>>>
```

Nous indiquons entre crochets la clé à laquelle nous souhaitons accéder. Si la clé n'existe pas, elle est ajoutée au dictionnaire avec la valeur spécifiée après le signe =. Sinon, l'ancienne valeur à l'emplacement indiqué est remplacée par la nouvelle :

```
1 >>> mon_dictionnaire = {}
2 >>> mon_dictionnaire["pseudo"] = "Prolixe"
3 >>> mon_dictionnaire["mot de passe"] = "*"
4 >>> mon_dictionnaire["pseudo"] = "6pri1"
5 >>> mon_dictionnaire
6 {'pseudo': '6pri1', 'mot de passe': '*'}
7 >>>
```

La valeur `'Prolixe'` pointée par la clé `'pseudo'` a été remplacée, à la ligne 4, par la valeur `'6pri1'`. Cela devrait vous rappeler la création de variables : si la variable n'existe pas, elle est créée, sinon elle est remplacée par la nouvelle valeur.

Pour accéder à la valeur d'une clé précise, c'est très simple :

```
1 >>> mon_dictionnaire["mot de passe"]
2 '*'
3 >>>
```

Si la clé n'existe pas dans le dictionnaire, une exception de type `KeyError` est levée.

Généralisons un peu tout cela : nous avons des dictionnaires, qui contiennent d'autres objets. On place ces derniers et on y accède grâce à des clés. Un dictionnaire ne doit naturellement pas contenir deux clés identiques (comme on l'a vu, la seconde valeur écrase la première). En revanche, rien n'empêche d'avoir deux valeurs identiques dans le dictionnaire.

Nous avons utilisé ici, pour nos clés et nos valeurs, des chaînes de caractères. Ce n'est absolument pas obligatoire. Comme avec les listes, vous pouvez utiliser des entiers comme clés :

```
1  >>> mon_dictionnaire = {}
2  >>> mon_dictionnaire[0] = "a"
3  >>> mon_dictionnaire[1] = "e"
4  >>> mon_dictionnaire[2] = "i"
5  >>> mon_dictionnaire[3] = "o"
6  >>> mon_dictionnaire[4] = "u"
7  >>> mon_dictionnaire[5] = "y"
8  >>> mon_dictionnaire
9  {0: 'a', 1: 'e', 2: 'i', 3: 'o', 4: 'u', 5: 'y'}
10 >>>
```

On a l'impression de recréer le fonctionnement d'une liste, mais ce n'est pas le cas : rappelez-vous qu'un dictionnaire n'a pas de structure ordonnée. Si vous supprimez par exemple l'indice 2, le dictionnaire, contrairement aux listes, ne va pas décaler toutes les clés d'indice supérieur. Il n'a pas été conçu pour.

On peut utiliser quasiment tous les types comme clés et absolument tous les types comme valeurs.

Voici un exemple un peu plus atypique de clés : on souhaite représenter un plateau d'échecs. Traditionnellement, on représente une case de l'échiquier par une lettre (de A à H) suivie d'un chiffre (de 1 à 8). La lettre définit la colonne et le chiffre définit la ligne.

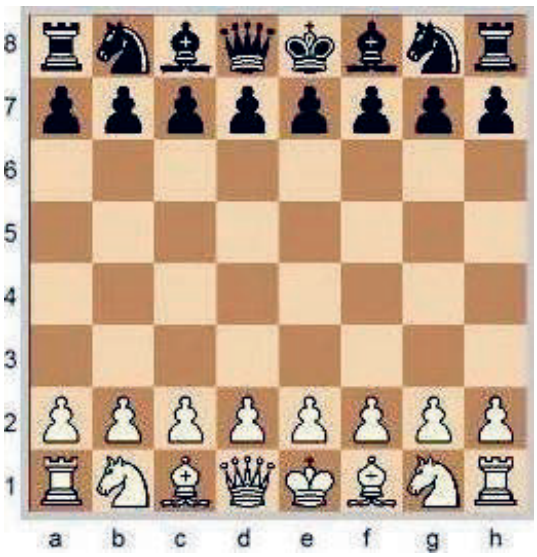


FIGURE 13.1 – Échiquier

Pourquoi ne pas définir un dictionnaire dont les clés sont des tuples contenant la lettre et le chiffre identifiant la case, auxquelles on associe comme valeurs le nom des pièces ?

```
échiquier = {}
échiquier['a', 1] = "tour blanche" # En bas à gauche de
 ↪  l'échiquier
échiquier['b', 1] = "cavalier blanc" # À droite de la tour
échiquier['c', 1] = "fou blanc" # À droite du cavalier
échiquier['d', 1] = "reine blanche" # À droite du fou
# ... Première ligne des blancs
échiquier['a', 2] = "pion blanc" # Devant la tour
échiquier['b', 2] = "pion blanc" # Devant le cavalier, à
 ↪  droite du pion
# ... Seconde ligne des blancs
```

Dans cet exemple, nos tuples sont sous-entendus. On ne les place pas entre parenthèses. Python comprend qu'on veut créer des tuples, ce qui est bien, mais l'important est que vous le compreniez aussi. Certains cours encouragent à toujours placer des parenthèses autour des tuples quand on les utilise. Pour ma part, je pense que, si vous gardez à l'esprit qu'il s'agit de tuples, si vous n'avez aucune peine à l'identifier, cela suffit. Si vous faites la confusion, mettez des parenthèses autour des tuples en toutes circonstances.

On peut aussi créer des dictionnaires déjà remplis :

```
placard = {"chemise": 3, "pantalon": 6, "t-shirt": 7}
```

On précise entre accolades la clé, le signe deux-points et la valeur correspondante. On sépare les différents couples `clé:valeur` par des virgules. C'est d'ailleurs comme cela que Python vous affiche un dictionnaire quand vous le lui demandez.

Certains ont peut-être essayé de créer des dictionnaires déjà remplis avant que je ne montre comment faire et ont écrit une instruction comme la suivante :

```
mon_dictionnaire = {'pseudo', 'mot de passe'}
```

Avec une telle instruction, ce n'est pas un dictionnaire que vous créez, mais un ensemble (`set`). Nous les étudierons au prochain chapitre. Un ensemble (`set`) est un objet conteneur (lui aussi), très semblable aux listes sauf qu'il ne peut contenir deux objets identiques.

Supprimer des clés d'un dictionnaire

Comme pour les listes, vous avez deux possibilités mais elles reviennent sensiblement au même :

- le mot-clé `del` ;
- la méthode de dictionnaire `pop`.

Je ne vais pas m'attarder sur le mot-clé `del`, qui fonctionne de la même façon que pour les listes :

```
placard = {"chemise": 3, "pantalon": 6, "t-shirt": 7}
del placard["chemise"]
```

La méthode `pop` supprime également la clé précisée, mais elle renvoie la valeur associée :

```
>>> placard = {"chemise": 3, "pantalon": 6, "t-shirt": 7}
>>> placard.pop("chemise")
3
>>>
```

Voilà pour le tour d'horizon. Ce fut bref et vous n'avez pas vu toutes les méthodes, bien entendu. Je vous laisse consulter l'aide pour une liste détaillée.

Ordre des dictionnaires

Je vous ai encouragé dès le début à considérer les dictionnaires comme des conteneurs sans ordre. Et si vous avez suivi avec attention les exemples que j'ai donnés, vous avez pu constater que Python conserve bel et bien l'ordre dans lequel les clés sont insérées. Cela semble un peu contradictoire à première vue !

```
>>> dictionnaire = {}
>>> dictionnaire["première clé"] = 150
```

```
>>> dictionnaire["seconde clé"] = -8.3
>>> dictionnaire["troisième clé"] = "autre chose"
>>> dictionnaire["quatrième clé"] = False
>>> dictionnaire
{'première clé': 150, 'seconde clé': -8.3, 'troisième clé':
 ↪  'autre chose', 'quatrième clé': False}
>>>
```

Python conserve l'ordre dans lequel les clés sont ajoutées à un dictionnaire. Il sera respecté, par exemple, lors du parcours que nous verrons plus loin. Ce n'était pas le cas avant Python 3.6. C'est une fonctionnalité du langage, parfois utile, mais n'utilisez pas les dictionnaires comme des listes, ce n'est pas leur première utilité. Malgré tout, si vous avez à la fois besoin d'un ordre stable et d'une correspondance entre clé et valeur, les dictionnaires le proposent nativement ; autant les utiliser dans ce contexte sans craindre le désordre.

Un peu plus loin

On se sert parfois des dictionnaires pour stocker des fonctions.

Je vais juste vous montrer rapidement le mécanisme sans trop m'y attarder. Là, je compte sur vous pour faire des tests si vous êtes intéressés. C'est encore un petit quelque chose que vous n'utiliserez peut-être pas tous les jours mais qu'il est utile de connaître.

Les fonctions sont manipulables comme des variables. Ce sont des objets, un peu particuliers mais des objets tout de même. Donc on peut les prendre pour valeur d'affectation ou les ranger dans des listes ou dictionnaires.

```
>>> print_2 = print # L'objet print_2 pointera sur la
 ↪  fonction print
>>> print_2("Affichons un message")
Affichons un message
>>>
```

On copie la fonction `print` dans une autre variable `print_2`. On peut ensuite appeler `print_2` et la fonction va afficher le texte saisi, tout comme `print` l'aurait fait.

En pratique, on affecte rarement des fonctions de cette manière. C'est peu utile. En revanche, on les range parfois dans des dictionnaires :

```
>>> def fête():
...     print("C'est la fête.")
...
>>> def oiseau():
...     print("Fais comme l'oiseau...")
...
>>> fonctions = {}
>>> fonctions["fête"] = fête # on ne met pas les parenthèses
>>> fonctions["oiseau"] = oiseau
```

```
>>> fonctions["oiseau"]
<function oiseau at 0x00BA5198>
>>> fonctions["oiseau"]() # on essaye de l'appeler
Fais comme l'oiseau...
>>>
```

Prenons dans l'ordre :

- On commence par définir deux fonctions, `fête` et `oiseau` (pardonnez l'exemple).
- On crée un dictionnaire nommé `fonctions`.
- On range dans ce dictionnaire les fonctions `fête` et `oiseau`. La clé pointant vers la fonction est le nom de la fonction, tout bêtement, mais on aurait pu lui donner un nom plus original.
- On essaye d'accéder à la fonction `oiseau` en tapant `fonctions[« oiseau »]`. Python nous renvoie un message assez abscons, `<function oiseau at 0x00BA5198>`, mais vous comprenez l'idée : c'est bel et bien notre fonction `oiseau`. Toutefois, pour l'appeler, il faut des parenthèses, comme pour toute fonction qui se respecte.
- En tapant `fonctions["oiseau"]()`, on accède à la fonction `oiseau` et on l'appelle dans la foulée.

On peut stocker les références des fonctions dans n'importe quel objet conteneur, des listes, des dictionnaires... et d'autres classes, quand nous apprendrons à en faire. Je ne vous demande pas de comprendre absolument la manipulation des références des fonctions, essayez simplement de retenir cet exemple. Dans tous les cas, nous aurons l'occasion d'y revenir.

Les méthodes de parcours

Comme vous l'imaginez, le parcours d'un dictionnaire ne s'effectue pas tout à fait comme celui d'une liste. La différence n'est pas si énorme que cela mais, la plupart du temps, on passe par des méthodes de dictionnaire.

Parcours des clés

Peut-être avez-vous déjà essayé par vous-mêmes de parcourir un dictionnaire comme on l'a fait pour les listes :

```
>>> fruits = {"pommes": 21, "melons": 3, "poires": 31}
>>> for clé in fruits:
...     print(clé)
...
pommes
melons
poires
>>>
```

Comme vous le voyez, si on essaye de parcourir un dictionnaire « simplement », on parcourt en réalité la liste de ses clés.

Une méthode de la classe `dict` retourne ce même résultat. Personnellement, je l'utilise plus fréquemment car on est sûr, en lisant l'instruction, que c'est la liste des clés que l'on parcourt :

```
>>> fruits = {"pommes": 21, "melons": 3, "poires": 31}
>>> for clé in fruits.keys():
...     print(clé)
...
pommes
melons
poires
>>>
```

La méthode `keys` (« clés » en anglais) renvoie la liste des clés contenues dans le dictionnaire. En vérité, ce n'est pas tout à fait une liste (essayez de taper `fruits.keys()` dans votre interpréteur), mais une séquence qui se parcourt comme une liste.

Parcours des valeurs

On peut aussi parcourir les valeurs contenues dans un dictionnaire. Pour ce faire, on utilise la méthode `values` (« valeurs » en anglais).

```
>>> fruits = {"pommes": 21, "melons": 3, "poires": 31}
>>> for valeur in fruits.values():
...     print(valeur)
...
21
3
31
>>>
```

Cette méthode est peu utilisée pour un parcours, car il est plus pratique de parcourir la liste des clés ; cela suffit pour connaître les valeurs correspondantes. Cependant, on peut aussi, bien entendu, l'utiliser dans une condition :

```
>>> if 21 in fruits.values():
...     print("Un des fruits se trouve dans la quantité 21.")
...
Un des fruits se trouve dans la quantité 21.
>>>
```

Parcours des clés et valeurs simultanément

Pour avoir en même temps les indices et les objets d'une liste, on utilise la fonction `enumerate`. Pour faire de même avec les dictionnaires, on utilise la méthode `items`.

Elle renvoie une liste, contenant les couples `clé:valeur`, sous la forme de `tuples`. Voyons comment l'utiliser :

```
>>> fruits = {"pommes": 21, "melons": 3, "poires": 31}
>>> for clé, valeur in fruits.items():
...     print(f"La clé {clé} contient la valeur {valeur}.")
...
La clé pommes contient la valeur 21.
La clé melons contient la valeur 3.
La clé poires contient la valeur 31.
>>>
```

Il est parfois très pratique de parcourir un dictionnaire avec ses clés et les valeurs associées.

Entraînez-vous, il n'y a que cela de vrai. Pourquoi pas reprendre l'exercice du chapitre précédent, avec notre inventaire de fruits, sauf que le type de l'inventaire ne serait pas une liste mais un dictionnaire associant les noms des fruits aux quantités ?

Les dictionnaires et paramètres de fonction

Cela ne vous rappelle pas quelque chose ? J'espère bien que si, on a vu quelque chose de similaire au chapitre précédent.

Si vous vous souvenez, on avait réussi à intercepter tous les paramètres de la fonction... sauf les paramètres nommés.

Récupérer les paramètres nommés dans un dictionnaire

Il existe aussi une façon de capturer les paramètres nommés d'une fonction. Dans ce cas, toutefois, ils sont placés dans un dictionnaire. Si, par exemple, vous appelez `fonction(paramètre='a')`, vous obtiendrez, dans le dictionnaire capturant les paramètres nommés, une clé `'paramètre'` liée à la valeur `'a'`. Voyez plutôt :

```
>>> def fonction_inconnue(**paramètres_nommés):
...     """Fonction permettant de voir comment récupérer
...     les paramètres nommés dans un dictionnaire.
...     """
...     print(f"J'ai reçu en paramètres nommés :
↪ {paramètres_nommés}")
...
>>> fonction_inconnue() # Aucun paramètre
J'ai reçu en paramètres nommés : {}
>>> fonction_inconnue(p=4, j=8)
J'ai reçu en paramètres nommés : {'p': 4, 'j': 8}
>>>
```

Notez que l'ordre dans lequel on appelle les paramètres est également respecté dans notre dictionnaire. C'est une fonctionnalité du langage même si elle est encore peu utilisée.

Pour capturer tous les paramètres nommés non précisés dans un dictionnaire, il faut ajouter deux étoiles ** avant le nom de la variable.

Si vous passez des paramètres non nommés à cette fonction, Python lève une exception.

Ainsi, pour qu'une fonction accepte n'importe quel type de paramètres, nommés ou non, dans n'importe quel ordre, dans n'importe quelle quantité, il faut la déclarer de la manière suivante :

```
def fonction_inconnue(*en_liste, **en_dictionnaire):
```

Tous les paramètres non nommés se retrouveront dans la variable `en_liste` et les paramètres nommés dans la variable `en_dictionnaire`.

Par convention, on a tendance à préférer `args` et `kwargs` comme noms de paramètres.

```
def fonction_inconnue(*args, **kwargs):
```

À quoi cela peut-il bien servir d'avoir une fonction qui accepte n'importe quel paramètre ?

Pour l'instant, pas à grand-chose, mais cela viendra. Quand on abordera le chapitre sur les décorateurs, vous vous en souviendrez et vous pourrez vous féliciter de connaître cette fonctionnalité.

Transformer un dictionnaire en paramètres nommés d'une fonction

Là encore, on peut faire exactement l'inverse : transformer un dictionnaire en paramètres nommés d'une fonction. Voyons un exemple tout simple :

```
>>> paramètres = {"sep": " >> ", "end":" -\n"}
>>> print("Voici", "un", "exemple", "d'appel", **paramètres)
Voici >> un >> exemple >> d'appel -
>>>
```

Les paramètres nommés sont transmis à la fonction par un dictionnaire. Pour indiquer à Python que le dictionnaire doit être transmis comme des paramètres nommés, on place deux étoiles avant son nom ** dans l'appel de la fonction.

Comme vous pouvez le voir, c'est comme si nous avions écrit :

```
>>> print("Voici", "un", "exemple", "d'appel", sep=" >> ",
↪   end= "-\n")
Voici >> un >> exemple >> d'appel -
>>>
```

Pour l'instant, vous devez trouver que c'est bien se compliquer la vie pour si peu. Nous verrons dans la suite de ce cours qu'il n'en est rien, en fait, même si nous n'utilisons pas cette fonctionnalité tous les jours.

Les compréhensions de dictionnaire

Vous souvenez-vous des compréhensions de liste (`list comprehensions`) ? Le même système et presque la même syntaxe permettent de créer des dictionnaires. Au lieu d'encadrer l'expression de crochets, on utilise des accolades et on spécifie bien le couple clé et valeur (séparés par deux points comme d'habitude) :

```
>>> liste = [1, 2, 3, 4, 5, 8, 12]
>>> carrés = {nombre: nombre ** 2 for nombre in liste}
>>> carrés
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 8: 64, 12: 144}
>>> carrés_pairs = {nombre: nombre ** 2 for nombre in liste
↪   if nombre % 2 == 0}
>>> carrés_pairs
{2: 4, 4: 16, 8: 64, 12: 144}
>>>
```

On commence par créer une liste (pour l'exemple) contenant des nombres. On crée ensuite deux dictionnaires :

— Le premier, `carrés`, contient les nombres de la liste en clés, leurs carrés en valeurs. Notez comment nous avons spécifié la clé et la valeur dans notre compréhension de dictionnaire.
— Le second, `carrés_pairs`, est identique sauf que l'on applique un filtrage : les nombres impairs sont ignorés ce qui rend notre dictionnaire plus petit que le premier.

Les compréhensions de dictionnaire sont moins utilisées (et souvent moins connues) que celles de liste. Elles sont utiles dans certains cas, cependant. Leur syntaxe permet de créer des dictionnaires assez considérables sans trop d'effort, comme vous l'avez vu.

Un autre exemple pourrait être utile. En programmation, on se retrouve parfois avec un dictionnaire à double-sens : ses clés et valeurs sont uniques et il est utile d'avoir les clés en valeur et les valeurs en clé. On crée donc un dictionnaire standard puis on le retourne pour en créer un autre :

```
>>> alphabet = {
...     'a': 1,
```

```
...     'b': 2,
...     'c': 3,
...     'd': 4,
...     # ...
... }
>>> alphabet_inversé = {numéro: lettre for lettre, numéro in
 ↪  alphabet.items()}
>>> alphabet_inversé
{1: 'a', 2: 'b', 3: 'c', 4: 'd'}
>>>
```

Notre dictionnaire `alphabet` contient en clés les lettres de l'alphabet et en valeurs leur position dans l'alphabet. Ensuite, nous le retournons. Notez que l'utilisation d'un dictionnaire ici est contestable, mais c'est pour l'exemple. Là encore, il existe plusieurs façons de résoudre ce problème, mais celle-ci est, à mon sens, la plus lisible une fois qu'on est habitué à la syntaxe. Notez bien l'utilisation de `items` ici ; nous avons besoin de la clé et de sa valeur pour l'inverser correctement.

Si ce dernier exemple vous semble un peu difficile, prenez le temps de le décomposer en plusieurs lignes. Une compréhension de dictionnaire (ou de liste ou d'ensemble) est une façon concise et performante de regrouper plusieurs lignes en une seule instruction. Il vous est toujours possible d'écrire la même opération avec une boucle conventionnelle comme nous l'avons vu plus haut.

Match et les dictionnaires

Toujours `match` ! Et la réponse est non, nous n'avons pas encore tout vu de ce mot-clé. Ici, nous allons nous concentrer sur les dictionnaires.

Cette fonctionnalité est assez proche de celle que nous avons vue pour les listes, aussi vais-je garder le même exemple et un code assez semblable.

Au lieu de stocker nos anniversaire, tâche et rendez-vous dans une liste, nous utiliserons un dictionnaire. Il sera plus court, on n'aura pas besoin de garder une chaîne spécifique pour indiquer ce que la structure contient ; la structure du dictionnaire à elle seule sera suffisante :

```
anniversaire = {"anniversaire": "Étienne"}
tâche = {"tâche": "ranger la cuisine"}
rendez_vous = {"rendez-vous": "Réunion professionnelle",
 ↪  "heures": ("10:00", "10:30")}
aujourd_hui = [anniversaire, tâche, rendez_vous]
```

Les dictionnaires ont des structures diverses, mais vous devriez pouvoir les reconnaître sans trop de peine. Si on veut appliquer une action différente en fonction du contenu d'une liste d'anniversaires, tâches et/ou rendez-vous, voici notre `match` complet :

```
>>> for à_faire in aujourd_hui:
...     match à_faire:
```

```
...         case {"anniversaire": personne}:
...             print(f"Pensez à souhaiter un joyeux
↪  anniversaire à {personne} !")
...         case {"tâche": objet}:
...             print(f"Pensez à {objet}.")
...         case {"rendez-vous": objet, "heures": [début,
↪  fin]}:
...             print(f"Rendez-vous prévu de {début} à {fin}:
↪  {objet}.")
...
Pensez à souhaiter un joyeux anniversaire à Étienne !
Pensez à ranger la cuisine.
Rendez-vous prévu de 10:00 à 10:30: Réunion professionnelle.
>>>
```

Cet exemple ne semble pas bien différent de celui du chapitre précédent... mais cette fois nous travaillons sur des dictionnaires. Notez la syntaxe pour ces différents `case` : au lieu d'extraire une séquence, on applique la branche en fonction du contenu du dictionnaire et on extrait des informations supplémentaires de ce dernier. Regardez bien en particulier le dernier `case`, celui concernant le rendez-vous : il est plus complexe, mais la structure est plus lisible également. Je vous laisse expérimenter avec cette syntaxe : elle est très pratique et, comme il arrive souvent qu'on représente des informations via des dictionnaires, savoir utiliser `match` et `case` avec eux n'est pas inutile.

En résumé

- Un dictionnaire est un objet conteneur associant des clés à des valeurs.
- Pour créer un dictionnaire, on utilise la syntaxe `dictionnaire = {cle1:valeur1, cle2=valeur2, cleN=valeurN}`.
- On ajoute ou remplace un élément dans un dictionnaire avec `dictionnaire[cle]=valeur`.
- On supprime une clé (et sa valeur correspondante) d'un dictionnaire en utilisant, au choix, le mot-clé `del` ou la méthode `pop`.
- On parcourt un dictionnaire grâce aux méthodes `keys` (clés), `values` (valeurs) ou `items` (couples `clé:valeur`).
- On capture les paramètres nommés passés à une fonction avec la syntaxe suivante : `def fonction_inconnue(**paramètres_nommés)`.

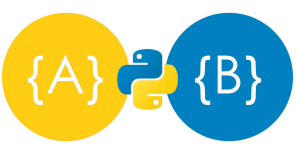
Chapitre 14

Les ensembles

Difficulté :

Nous avons vu les listes, tuples et dictionnaires dans les parties précédentes. Ces types sont sans doute les plus utilisés quand il s'agit de conserver des informations avec Python. Il existe toutefois un autre type de séquence légèrement moins utilisé, peut-être du fait de sa relative nouveauté en Python, mais tout aussi utile : les ensembles (`set`).

Ce chapitre y est consacré. Il est relativement court et optionnel vu la rareté de la séquence, mais il est loin d'être difficile à suivre. En outre, il ajoute quelques exemples de logiques qui aident à mieux comprendre les séquences et les stratégies de programmation dans des cas concrets, ce qui n'est jamais à négliger.



Utilité et création des ensembles

Un ensemble en Python (de type `set`) est une séquence, semblable aux listes ou tuples, mais avec une différence importante : on ne peut conserver deux fois la même information dans un ensemble. Cette propriété répond à quelques problèmes de programmation courants que nous allons voir.

Les ensembles ont été intégrés dans Python relativement tard en comparaison des autres séquences. Ils ont pourtant une syntaxe à part entière assez proche de celle des dictionnaires. On utilise les accolades pour créer des ensembles en Python :

```
>>> ensemble = {0, 1, 2, 3}
>>> ensemble
{0, 1, 2, 3}
>>>
```

Avoir une syntaxe aussi proche de celle des dictionnaires pose parfois quelques problèmes de compréhension :

1. Gardez à l'esprit qu'un dictionnaire est également délimité par des accolades, mais que chaque élément séparé par une virgule est constitué d'une clé et d'une valeur séparées par le signe deux-points, alors que, dans un ensemble, il n'y a que des valeurs. C'est la seule différence entre un ensemble et un dictionnaire.
2. Python ne peut donc pas faire la différence quand on spécifie une séquence vide avec des accolades (`ensemble = {}`). Dans le doute, il définit un dictionnaire vide. Pour créer un ensemble vide, il faut utiliser le constructeur sans paramètre : `ensemble = set()` .

Comme pour les autres séquences en Python, il est possible de convertir une autre séquence en ensemble grâce au constructeur :

```
>>> données = [1, 2, 3, 'b']
>>> ensemble = set(données)
>>> ensemble
{1, 2, 3, 'b'}
>>>
```

1. On commence par créer une liste que l'on place dans une variable `données`.
2. On crée une nouvelle variable appelée `ensemble`, contenant l'ensemble (classe `set`) de la liste.
3. On affiche ensuite le résultat, qui est assez proche de notre liste d'origine.

Jusqu'ici, vous ne voyez sans doute pas grand intérêt aux ensembles. L'exemple précédent a pu éveiller la curiosité de certains. Rappelez-vous, un ensemble ne peut contenir la même donnée qu'une fois. Ce n'est pas le cas des listes. Si donc notre liste contient plusieurs fois la même chose... :

```
>>> données = [1, 2, 1, 3, 2, 4]
>>> ensemble = set(données)
>>> ensemble
{1, 2, 3, 4}
>>>
```

Deux choses importantes sont à noter ici :

1. Les données dupliquées ont été éliminées. `1` est ajouté une fois, `2` également, ainsi de suite jusqu'à `4`.
2. En parcourant la liste de données, on peut voir que chaque valeur est ajoutée dans la séquence si elle n'est pas déjà présente.

L'affichage de notre ensemble aide à comprendre ce qui se passe, mais l'ordre, dans notre cas précis, peut être trompeur : Python ne maintient pas d'ordre particulier dans les ensembles, c'est pourquoi vous verrez des ensembles sans ordre apparent. L'important est le contenu, pas son ordre.

Opérations sur les ensembles

La création d'ensembles en Python est très facile, comme nous l'avons vu dans la partie précédente. Voyons quelques cas concrets et quelques opérations basiques :

Opérations sur un ensemble

```
>>> lettres = set("calamar")
>>> lettres
{'r', 'm', 'l', 'a', 'c'}
>>>
```

Il faut bien se souvenir qu'on peut convertir une séquence dans un ensemble. Et les chaînes de caractères sont des séquences... de caractères. Ici, nous créons donc un ensemble (`set`) depuis le mot `calamar`, contenant chaque lettre (unique) du mot.

Comme indiqué, l'ordre de l'ensemble n'est absolument pas le reflet de notre mot et pourrait même être différent chez vous.

Notre ensemble contient donc les lettres `c`, `a`, `l`, `m` et `r`. `a` qui apparaît deux autres fois n'est pas ajoutée plus d'une fois.

On peut aussi ajouter des éléments dans notre ensemble :

```
>>> lettres.add('i')
>>> lettres
{'i', 'r', 'm', 'l', 'a', 'c'}
>>> lettres.add('m')
>>> lettres
{'i', 'r', 'm', 'l', 'a', 'c'}
>>>
```

On ajoute d'abord `i`, qui n'est pas dans l'ensemble. On utilise la méthode `add` pour ce faire. On essaye ensuite d'ajouter `m`, qui est déjà dans l'ensemble et n'est donc pas prise en compte. Aucune erreur ne survient (la lettre n'est simplement pas ajoutée dans l'ensemble, puisqu'elle y est déjà présente).

Il est facile de vérifier si une valeur est dans l'ensemble :

```
>>> 'i' in lettres
True
>>> 'a' in lettres
True
>>> 'k' in lettres
False
>>>
```

Ou de parcourir l'ensemble :

```
>>> for lettre in lettres:
...     print(f"Ici se trouve la lettre {lettre}.")
...
Ici se trouve la lettre i.
Ici se trouve la lettre r.
Ici se trouve la lettre m.
Ici se trouve la lettre l.
Ici se trouve la lettre a.
Ici se trouve la lettre c.
>>>
```

Supprimer des éléments est en revanche quelque peu problématique :

```
>>> del lettres['i']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support item deletion
>>>
```

Bon ! C'est assez définitif. On ne peut pas retirer d'élément depuis un ensemble ; en tout cas, pas comme cela.

La raison pour laquelle l'utilisation de `del` n'est pas possible dans ce contexte est qu'il y a deux possibilités courantes pour un ensemble. Python nous donne le choix. Il va falloir utiliser des méthodes précises :

```
>>> lettres.discard('i')
>>> lettres
{'r', 'm', 'l', 'a', 'c'}
>>> lettres.discard('x')
>>> lettres
{'r', 'm', 'l', 'a', 'c'}
>>> lettres.remove('a')
>>> lettres.remove('z')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'z'
>>>
```

Supprimer des éléments peut donc se faire en utilisant la méthode `discard` ou la méthode `remove`. Elles sont différentes dans leur gestion des erreurs :

- `discard` ne signale pas d'erreur si la donnée à supprimer n'est pas dans l'ensemble.
- `remove` en revanche lève une exception si la donnée est absente de l'ensemble.

Dans le doute, Python s'abstient de deviner ce que l'on veut utiliser quand on écrit `del`.

On peut aussi effacer un ensemble complet avec `clear` :

```
>>> lettres.clear()
>>> lettres
set()
>>>
```

Il est enfin possible de retirer un élément de l'ensemble (on ne sait pas lequel) :

```
>>> lettres
{'l', 'r', 'c', 'm'}
>>> lettres.pop()
'r'
>>> lettres
{'l', 'c', 'm'}
>>>
```

À quoi sert de retirer un élément au hasard de notre ensemble ?

Nous verrons que cela est assez utile dans certains cas précis, notamment si l'on n'a qu'un seul élément dans l'ensemble.

Enfin, voici deux méthodes courantes pour conclure sur la création et la mise à jour d'ensembles :

```
>>> lettres = set("calamar")
>>> lettres.update("sardine")
>>> lettres
{'i', 'r', 'm', 's', 'l', 'd', 'a', 'c', 'n', 'e'}
>>>
```

La méthode `update` ajoute une nouvelle séquence dans notre ensemble. On ajoute ici les lettres d'un second mot (seulement celles qui ne sont pas déjà présentes). Cet exemple pourrait sembler confus mais il offre une introduction au travail sur deux ensembles que nous allons voir très bientôt... après une dernière méthode un peu plus avancée de création d'ensembles.

Les compréhensions de listes (`list comprehension`) sont très souvent utilisées. Leur syntaxe peut être étendue à d'autres types de données, dont les ensembles.

```
>>> nombres = {indice * 2 for indice in (0, 1, 2, 3, 4)}
>>> nombres
{0, 2, 4, 6, 8}
>>>
```

On peut définir un ensemble en utilisant la même syntaxe que pour les compréhensions de liste, à la différence que les délimiteurs utilisés sont des accolades.

Travail sur deux ensembles

Comme nous l'avons vu, l'utilisation principale des ensembles est d'éliminer des données dupliquées. Notez que vous pourriez faire la même chose avec des listes par exemple, mais le travail avec des ensembles en Python est tellement plus rapide (et lisible, une fois qu'on est habitué à la syntaxe).

Une autre utilisation est la comparaison entre deux ensembles ou plus. Il sera plus simple de voir tout cela suivant quelques exemples :

```
>>> nombres_1 = {0, 1, 2, 3, 4}
>>> nombres_2 = {3, 4, 5, 6, 7}
>>>
```

Je crée volontairement deux ensembles assez simples : `nombres_1` et `nombres_2` contiennent des chiffres qui se suivent. Notez qu'ils présentent des points communs et des différences.

Nous allons étudier quelques opérateurs qui sont moins fréquemment utilisés en Python et dont je n'ai pas parlé auparavant. Python propose des méthodes explicites pour faire la même chose. Il est utile de connaître les deux syntaxes, mais utilisez celle qui vous semble la plus pratique :

```
>>> nombres_1 | nombres_2
{0, 1, 2, 3, 4, 5, 6, 7}
>>> nombres_1.union(nombres_2)
{0, 1, 2, 3, 4, 5, 6, 7}
>>>
```

L'opérateur | renvoie l'union entre deux ensembles : l'union consiste à créer un ensemble contenant `nombres_1` et `nombres_2` (sans dupliquer aucune donnée). La méthode `union` réalise exactement la même chose.

```
>>> nombres_1 & nombres_2
{3, 4}
>>> nombres_1.intersection(nombres_2)
{3, 4}
>>>
```

L'opérateur & calcule l'intersection entre les deux ensembles : il ne garde que les données qui leur sont communes. La méthode `intersection` retourne exactement la même chose.

```
>>> nombres_1 - nombres_2
{0, 1, 2}
>>> nombres_1.difference(nombres_2)
{0, 1, 2}
>>>
```

L'opérateur - calcule la différence entre deux ensembles : la différence consiste ici à prendre tous les éléments présents dans `nombres_1` à l'exception de ceux se trouvant dans `nombres_2` (c'est pour cela que `3` et `4` ne sont pas présents). La méthode `difference` retourne exactement la même chose.

Ces méthodes ont des ramifications mathématiques importantes. Même s'il n'est pas absolument nécessaire d'être bon en maths pour savoir programmer (j'en suis, hélas, la preuve vivante), logique mathématique et programmation se rencontrent fréquemment.

On peut aussi comparer les ensembles pour savoir si l'un d'entre eux est inclus dans un autre. Prenons deux nouvelles variables :

```
>>> grand = set(range(10))
>>> grand
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> petit = {3, 4, 5}
>>> petit <= grand
True
>>> petit >= grand
False
>>>
```

Notez l'utilisation de `range` ici qui permet de créer rapidement un ensemble de `0` à `9` (tout cela est paramétrable).

L'opérateur `<=` teste si `petit` est un sous-élément de `grand`, c'est-à-dire si tous les éléments de `petit` sont dans `grand`. C'est le cas ici. En revanche, `petit` ne contient pas tous les éléments de l'ensemble `grand`, l'opérateur `>=` retourne donc `False`. Les méthodes `issubset` et `issuperset` font la même chose.

Quelques exemples d'utilisation

Voyons un problème relativement fréquent que les ensembles aident à résoudre : admettons que vous avez deux variables, l'une contenant le nom des habitants d'une ville (une grande ville de préférence), l'autre contenant le nom des personnes ayant participé à un salon. Vous cherchez à compter combien d'habitants de la ville ont participé au salon.

En admettant que les variables `ville` et `salon` contiennent des listes de chaînes de caractères, comment coderiez-vous ce problème ? C'est un bon exercice et je vous encourage à le résoudre. Il est inutile de trouver des listes de plusieurs millions de noms, il s'agit d'un exemple ici.

Voici une approche possible :

```
nombre = 0
for nom in salon:
    if nom in ville:
        nombre += 1
```

Cet exemple fonctionne relativement rapidement (même avec plusieurs centaines de noms). Cependant, il existe une solution utilisant les ensembles. Admettons donc que `ville` et `salon` soient des ensembles de chaînes de caractères :

```
nombre = len(salon & ville)
# Ou avec le nom de méthode explicite
nombre = len(salon.intersection(ville))
```

Pour rappel, l'intersection entre nos deux ensembles ici consiste à retourner l'ensemble des noms dans la ville et dans le salon. On prend ensuite la taille de ce dernier ensemble. Cette méthode est nettement plus courte et assez lisible. Elle est également plus rapide que la première.

Le même cas se pose, par exemple, si on a deux listes de coordonnées e-mails et si l'on souhaite extraire seulement les adresses que l'on trouve dans les deux listes.

Un autre problème : admettons que l'on possède un ensemble de deux éléments seulement mais que le programmeur ne connaisse que l'un d'entre eux :

```
>>> deux = {1, 3800}
>>>
```

Notre ensemble a été créé par l'utilisateur par exemple. Nous savons que l'un de ses deux éléments est `1`, mais nous ignorons le second. Là encore, on peut arriver à ce résultat via des listes, mais c'est bien plus facile avec des ensembles :

```
>>> deux
{3800, 1}
>>> connu = 1
>>> inconnu = (deux - {connu}).pop()
>>> inconnu
3800
>>>
```

Le code est assez court, mais la syntaxe peut paraître particulière. En utilisant les ensembles, il est possible d'obtenir ce résultat de bien d'autres façons. Ici, on ne modifie pas `deux` et on essaye de minimiser les opérations à effectuer :

1. On a `deux` contenant deux éléments et `connu` contenant l'élément connu `1`.
2. On fait la différence entre `deux` et un ensemble ne contenant que `connu`. La différence ici est un ensemble ne contenant que notre nombre inconnu.
3. On utilise ensuite `pop()` pour retirer le seul élément dans notre ensemble. Je vous avais bien dit que cette méthode n'était pas inutile !

Si vous êtes un peu perdu, n'hésitez pas à faire cet exercice par étape :

```
>>> deux
{3800, 1}
>>> deux - {connu}
{3800}
>>> (deux - {connu}).pop()
3800
>>>
```

On place des parenthèses ici, sinon, la méthode `pop()` serait appelée sur `{connu}`, pas sur la différence entre les ensembles.

Ces petits exercices ou problèmes ne vous semblent peut-être pas très utiles. Il est difficile de dire si ces méthodes vous seront d'un grand secours par la suite. Néanmoins, résoudre ces exercices, ou même essayer, vous habitue à considérer ces problèmes en termes de logique, ce qui est bien utile en programmation.

Il existe de nombreuses autres méthodes ; je n'ai parlé que des principales ici. Comme toujours, pour une liste complète et à jour, reportez-vous à la documentation officielle en français sur les ensembles : `https://docs.python.org/fr/3/library/stdtypes.html\#set`

En résumé

— Les ensembles en Python permettent de définir des séquences d'éléments uniques.
— Cette propriété est utilisée pour retirer les données dupliquées ou comparer deux séquences.
— La syntaxe des ensembles est proche de celle des dictionnaires en Python, utilisant les accolades pour créer un ensemble non vide, ou bien le constructeur (`set()`) si l'ensemble est vide.
— On peut ajouter ou retirer des éléments aux ensembles, les parcourir et obtenir de nouveaux ensembles comparant deux séquences.

Chapitre 15

Les fichiers

Difficulté :

Poursuivons notre tour d'horizon des principaux objets. Dans ce chapitre, nous allons découvrir les fichiers, comment les ouvrir, les lire et écrire dedans.

Nous finirons en expliquant comment sauvegarder nos objets dans des fichiers, afin de les utiliser d'une session à l'autre de notre programme.

Avant de commencer

Nous allons beaucoup travailler sur des répertoires et des fichiers, autrement dit sur votre disque. Donc je vais vous donner quelques informations générales avant de commencer pour que, malgré vos différents systèmes et configurations, vous puissiez essayer les instructions que je vais vous montrer.

Tout d'abord, pourquoi lire ou écrire dans des fichiers ?

Peut-être que vous ne voyez pas trop l'intérêt de savoir lire et écrire dans des fichiers, hormis quelques applications de temps à autre. Pourtant, souvenez-vous que, quand vous fermez votre programme, aucune de vos variables n'est sauvegardée. Or, les fichiers sont, justement, un excellent moyen de garder les valeurs de certains objets pour les récupérer quand vous rouvrirez votre programme. Par exemple, un petit jeu peut enregistrer les scores des joueurs.

Si, dans notre TP **ZCasino**, nous avions enregistré la somme que nous avions en poche au moment de quitter le casino, nous aurions pu rejouer sans repartir de zéro.

Changer le répertoire de travail courant

Si vous souhaitez travailler dans l'interpréteur Python, et je vous y encourage, vous devrez changer le répertoire de travail courant. En effet, au lancement de l'interpréteur, le répertoire de travail courant est celui dans lequel se trouve l'exécutable de l'interpréteur. Sous Windows, c'est `C:\Python3X`, le X étant différent en fonction de votre version du langage. Dans tous les cas, je vous invite à changer de répertoire de travail courant. Pour cela, vous devez utiliser une fonction du module `os`, qui s'appelle `chdir (Change Directory)`.

```
>>> import os
>>> os.chdir("C:/tests python")
>>>
```

Pour que cette instruction fonctionne, le répertoire doit exister. Modifiez la chaîne passée en paramètre de `os.chdir` en fonction du dossier dans lequel vous souhaitez vous déplacer.

Je vous conseille, que vous soyez sous Windows ou non, d'utiliser le symbole / pour décrire un chemin.

Vous pouvez utiliser, en la doublant, la barre oblique inverse \\ mais, si vous oubliez de la doubler, vous obtiendrez des erreurs. Je vous conseille donc d'utiliser la barre oblique /, qui fonctionne très bien même sous Windows.

Quand vous lancez un programme Python directement, par exemple en double-cliquant dessus, le répertoire courant est celui d'où vous le lancez. Par exemple, si vous avez un fichier `mon_programme.py` contenu sur le disque `C:`, le répertoire de travail courant quand vous lancerez le programme sera `C:\`.

Chemins relatifs et absolus

Pour décrire l'arborescence d'un système, on a deux possibilités :

— les chemins absolus ;
— les chemins relatifs.

Le chemin absolu

Quand on décrit une cible (un fichier ou un répertoire) sous la forme d'un chemin absolu, on décrit la suite des répertoires menant au fichier. Sous Windows, on partira du nom de volume (`C:\`, `D:\`...). Sous les systèmes Unix, ce sera plus vraisemblablement depuis /.

Par exemple, sous Windows, si on a un fichier nommé `fic.txt`, contenu dans un dossier `test`, lui-même présent sur le disque `C:`, le chemin absolu qui y mène est `C:\test\fic.txt`.

Le chemin relatif

Quand on décrit la position d'un fichier grâce à un chemin relatif, cela veut dire que l'on tient compte du dossier dans lequel on se trouve actuellement. Ainsi, si on se trouve dans le dossier `C:\test` et si l'on souhaite accéder au fichier `fic.txt` contenu dans ce même dossier, le chemin relatif qui y mène est tout simplement `fic.txt`.

Maintenant, si on se trouve dans `C:`, notre chemin relatif sera `test\fic.txt`.

Quand on décrit un chemin relatif, on utilise parfois le symbole `..` qui désigne le répertoire parent. Voici un nouvel exemple :

```
— C:
   — test
      — rep1
         — fic1.txt
      — rep2
         — fic2.txt
         — fic3.txt
```

C'est dans notre dossier `test` que tout se passe. Nous avons deux sous-répertoires nommés `rep1` et `rep2`. Dans `rep1`, nous avons un seul fichier : `fic1.txt`. Dans `rep2`, nous avons deux fichiers : `fic2.txt` et `fic3.txt`.

Si le répertoire de travail courant est `rep2` et si l'on souhaite accéder à `fic1.txt`, notre chemin relatif est donc `..\rep1\fic1.txt`.

J'utilise ici des barres obliques inverses parce que l'exemple d'arborescence est un modèle Windows. Dans votre code, je vous conseille quand même d'utiliser une barre oblique (`/`).

Résumé

Les chemins absolus et relatifs sont donc deux moyens de décrire le chemin menant à des fichiers ou répertoires. Cependant, si le résultat est le même, le moyen utilisé n'est pas identique : en absolu, on décrit l'intégralité du chemin menant au fichier, peu importe l'endroit où on se trouve. Un chemin absolu permet d'accéder à un endroit dans le disque quel que soit le répertoire de travail courant. L'inconvénient de cette méthode, c'est qu'on doit préalablement savoir où se trouvent, sur le disque, les fichiers dont on a besoin.

Le chemin relatif décrit la succession de répertoires à parcourir en prenant comme point d'origine non pas la racine, ou le périphérique sur lequel est stockée la cible, mais le répertoire dans lequel on se trouve. Cela présente certains avantages quand on code un projet, on n'est pas obligé de savoir où le projet est stocké pour construire plusieurs répertoires. Néanmoins, ce n'est pas forcément la meilleure solution en toute circonstance.

Comme je l'ai dit, quand on lance l'interpréteur Python, on a bel et bien un répertoire de travail courant. Vous pouvez l'afficher grâce à la fonction `os.getcwd()` [1].

Cela devrait donc vous suffire. Pour les démonstrations qui vont suivre, placez-vous, à l'aide de `os.chdir`, dans un répertoire de test créé pour l'occasion.

Lecture et écriture dans un fichier

Nous allons commencer à lire avant d'écrire dans un fichier. Pour l'exemple donc, je vous invite à créer un fichier dans le répertoire de travail courant que vous avez choisi. En manque flagrant d'inspiration, je vais l'appeler `fichier.txt` et je vais écrire dedans, à l'aide d'un éditeur sans mise en forme (tel que le bloc-notes Windows) : *« C'est le contenu du fichier. Spectaculaire non ? »*

Ouverture du fichier

D'abord, il nous faut ouvrir le fichier avec Python. Le procédé recommandé dans les versions plus récentes du langage est de créer un objet représentant le chemin (relatif ou absolu) menant au fichier et de demander à Python d'ouvrir ce dernier à cet emplacement.

1. CWD = « Current Working Directory »

Pour l'exemple, nous allons simplement essayer de lire `fichier.txt` dans le répertoire courant. Il nous faut donc commencer par créer un objet `pathlib.Path` contenant ce chemin :

```
>>> from pathlib import Path
>>> chemin = Path("fichier.txt")
```

Vous pourriez donner au constructeur `Path` un chemin absolu. Ici on lui a donné un chemin relatif qui fait donc référence au dossier dans lequel on se trouve.

Pourquoi créer un objet intermédiaire ?

C'est une question valide : on veut juste ouvrir notre fichier pour l'heure, alors pourquoi créer un objet juste pour ça ? En Python, il existe une fonction, disponible sans rien importer, qui pourrait ouvrir notre fichier sans créer d'objet intermédiaire pour le chemin. Elle s'appelle `open`. La raison pour laquelle on passe par un objet `pathlib.Path` est qu'il est bien plus facile de manipuler notre fichier si l'on souhaite appliquer d'autres opérations dessus :

```
>>> chemin.exists() # Est-ce que le fichier existe ?
True
>>> chemin.absolute() # Retourne le chemin absolu menant au
 ↪  fichier.
WindowsPath('C:/Users/Prolixe/fichier.txt')
>>>
```

Comme vous le voyez, il existe plusieurs méthodes sur `pathlib.Path` qui sont assez utiles. On peut parcourir tous les fichiers d'une extension spécifique dans le chemin d'un dossier (incluant tous les sous-dossiers), on peut faire des opérations sur les dossiers ou fichiers, supprimer les fichiers ou changer leur nom... Tout cela est réalisable sans passer par `pathlib.Path`, mais c'est quand même assez élégant, je trouve.

Bien, revenons à notre problème le plus pressant. Comment ouvrir notre fichier pour le lire ou l'écrire ?

On va utiliser la méthode `open` de notre chemin. Elle prend en paramètre :

- le mode d'ouverture.
- l'encodage du fichier (optionnel)

Le mode est donné sous la forme d'une chaîne de caractères. En voici les principaux :

- `'r'` : ouverture en lecture (Read).
- `'w'` : ouverture en écriture (Write). Le contenu du fichier est écrasé. Si le fichier n'existe pas, il est créé.
- `'a'` : ouverture en écriture en mode ajout (Append). On écrit à la fin du fichier sans écraser l'ancien contenu du fichier. Si le fichier n'existe pas, il est créé.

On peut ajouter à tous ces modes le signe `b` pour ouvrir le fichier en mode binaire. Nous en verrons plus loin l'utilité ; c'est un mode un peu particulier.

```
>>> mon_fichier = chemin.open("r")
>>> mon_fichier
<_io.TextIOWrapper name='fichier.txt' mode='r'
 ↪  encoding='cp1252'>
>>> type(mon_fichier)
<class '_io.TextIOWrapper'>
>>>
```

L'encodage précisé quand on affiche le fichier dans l'interpréteur peut être très différent suivant votre système. Ici, je suis dans l'interpréteur Python sous Windows, donc avec un encodage Windows propre à la console. Ne soyez pas surpris s'il est différent chez vous. On peut naturellement changer cet encodage (j'ai tendance à toujours utiliser l'UTF-8 quand je manipule des fichiers, sauf si j'ai une très bonne raison pour procéder autrement).

La méthode `open` crée donc un fichier. Elle renvoie un objet de la classe `TextIoWrapper`. Par la suite, nous utiliserons des méthodes de cette classe pour interagir avec le fichier.

Le type de l'objet doit vous surprendre quelque peu. Il aurait très bien pu s'appeler `File` après tout. En fait, `open` ouvre un fichier, mais `TextIoWrapper` est utilisé dans d'autres circonstances, pour afficher du texte à l'écran par exemple. Bon, cela ne nous concerne pas trop ici, je ne vais pas m'y attarder.

Fermer le fichier

N'oubliez pas de fermer un fichier après l'avoir ouvert. Si d'autres applications, ou d'autres morceaux de votre propre code, souhaitent y accéder, ils n'y seront pas autorisés car le fichier sera déjà ouvert. C'est surtout vrai en écriture, mais prenez de bonnes habitudes. La méthode à utiliser est `close` :

```
>>> mon_fichier.close()
>>>
```

Lire l'intégralité du fichier

Pour ce faire, on utilise la méthode `read` de la classe `TextIoWrapper`. Elle renvoie l'intégralité du fichier :

```
>>> chemin = Path("fichier.txt")
>>> mon_fichier = chemin.open("r")
>>> contenu = mon_fichier.read()
>>> print(contenu)
C'est le contenu du fichier. Spectaculaire non ?
>>> mon_fichier.close()
>>>
```

Quoi de plus simple ? La méthode `read` renvoie tout le contenu du fichier, que l'on capture dans une chaîne de caractères. Notre fichier ne contient pas de saut de ligne mais, si c'était le cas, vous verriez des signes `\n` dans votre variable `contenu`.

Maintenant que vous avez une chaîne, vous pouvez naturellement tout faire : la convertir, tout entière ou en partie, si c'est nécessaire, découper la chaîne pour parcourir chaque ligne et les traiter… bref, tout est possible.

Écriture dans un fichier

Bien entendu, il nous faut ouvrir le fichier avant tout. Vous pouvez utiliser le mode `w` ou le mode `a`. Le premier écrase le contenu éventuel, alors que le second ajoute ce que l'on écrit à la fin du fichier. À vous de voir en fonction de vos besoins. Dans tous les cas, ces deux modes créent le fichier s'il n'existe pas.

Écrire une chaîne

Pour écrire dans un fichier, on utilise la méthode `write` en lui passant une chaîne en paramètre. Elle renvoie le nombre de caractères qui ont été écrits. On n'est naturellement pas obligé de récupérer cette valeur, sauf si on en a besoin.

```
1 >>> chemin = Path("fichier.txt")
2 >>> mon_fichier = chemin.open("w") # Argh j'ai tout écrasé !
3 >>> mon_fichier.write("Premier test d'écriture dans un
   ↪  fichier via Python")
4 50
5 >>> mon_fichier.close()
6 >>>
```

Vérifiez que votre fichier contient bien le texte qu'on y a écrit.

Écrire d'autres types de données

La méthode `write` n'accepte en paramètre que des chaînes de caractères. Si vous voulez écrire des nombres dans votre fichier (des scores par exemple), il vous faudra les convertir avant de les écrire et après les avoir lus.

Le mot-clé `with`

Ne désespérez pas, il ne nous reste plus autant de mots-clés à découvrir… mais quelques-uns tout de même ; et même certains dont je ne parlerai pas…

On n'est jamais à l'abri d'une erreur, surtout quand on manipule des fichiers. Il peut se produire des erreurs quand on lit, quand on écrit… et si l'on n'y prend garde, le fichier restera ouvert.

Comme je vous l'ai dit, c'est plutôt gênant et cela peut même être grave. Si votre programme souhaite de nouveau utiliser ce fichier, il ne pourra pas forcément y accéder, puisqu'il a déjà été ouvert.

Il existe un mot-clé qui évite cette situation : `with`. Voici sa syntaxe :

```
with mon_chemin.open(mode_ouverture) as variable:
    # Opérations sur le fichier
```

On trouve dans l'ordre :

- Le mot-clé `with`, prélude au bloc dans lequel on va manipuler notre fichier. Ce mot-clé se retrouve dans la manipulation d'autres objets, mais nous ne l'aborderons pas ici.
- Notre objet. Ici, on appelle `open` qui va renvoyer un objet `TextIOWrapper` (notre fichier).
- Le mot-clé `as` que nous avons déjà vu dans le mécanisme d'importation et dans les exceptions. Il signifie toujours la même chose : « en tant que ».
- La variable qui contiendra notre objet. Si elle n'existe pas, Python la crée.

```
>>> with mon_chemin.open('r') as mon_fichier:
...     texte = mon_fichier.read()
...
>>>
```

Cela ne veut pas dire que le bloc d'instructions ne lèvera aucune exception.

Cela signifie simplement que, si une exception se produit, le fichier sera tout de même fermé à la fin du bloc.

Le mot-clé `with` crée un « context manager » (gestionnaire de contexte) qui vérifie que le fichier est ouvert et fermé, même si des erreurs se produisent pendant le bloc. Vous découvrirez plus loin d'autres objets utilisant le même mécanisme.

Vous pouvez appeler `mon_fichier.closed` pour le vérifier, qui vaudra `True` si le fichier est fermé.

Il est inutile, par conséquent, de fermer le fichier à la fin du bloc `with`. Python va le faire tout seul, qu'une exception soit levée ou non. Je vous encourage à utiliser cette syntaxe ; elle est plus sûre et plus facile à comprendre.

Le module `pathlib`

Depuis la version 3.4, le module `pathlib` a été ajouté à la bibliothèque standard de Python. Il a été conçu pour interagir avec votre système de fichiers, parcourir les répertoires et fichiers. C'est à présent la méthode recommandée quand vous voulez interagir avec votre disque dur, du moins pour ces opérations. Voici un petit code d'exemple des possibilités de ce module :

```
>>> from pathlib import Path
>>> chemin = Path.cwd()
>>> chemin
WindowsPath('D:/livre/python/partie2/chap6')
>>> # C'est le répertoire courant dans lequel je me trouve
... chemin.iterdir()
<generator object Path.iterdir at 0x004C1AF0>
>>> # Cette méthode indique ce qu'il y a dans le répertoire.
... # Mais on nous affiche un générateur (generator) ce qui
... # n'est pas bien utile. Nous verrons les générateurs
... # dans un prochain chapitre, pour l'instant, vous
... # pouvez les considérer comme des listes.
... list(chemin.iterdir())
[WindowsPath('D:/livre/python/partie2/chap6/fichier.txt')]
>>> chemin_fichier = chemin / "fichier.txt"
>>> # chemin_fichier est maintenant le chemin (Path)
... # menant vers fichier.txt
... chemin_fichier.exists() # Le fichier existe-t-il ?
True
>>>
```

`pathlib` propose une interface claire et lisible pour parcourir vos répertoires et vos fichiers. Je vous conseille de jeter un coup d'œil à sa documentation officielle (en français) : `https://docs.python.org/fr/3/library/pathlib.html`

Allez ! Direction le module `pickle`, dans lequel nous allons apprendre à sauvegarder nos objets dans des fichiers.

Enregistrer des objets dans des fichiers

Dans beaucoup de langages de haut niveau, il est possible de sauvegarder ses objets dans un fichier. Python ne fait pas exception. Grâce au module `pickle` que nous allons découvrir, on peut enregistrer n'importe quel objet et le récupérer par la suite, au prochain lancement du programme, par exemple. En outre, le fichier résultant pourra être lu depuis n'importe quel système d'exploitation (à condition, naturellement, que celui-ci prenne en charge Python).

Enregistrer un objet dans un fichier

Il nous faut naturellement d'abord importer le module `pickle`.

```
>>> import pickle
>>>
```

On va ensuite utiliser deux fonctions incluses dans ce module : `dump` et `load`.

C'est la première qui nous intéresse dans cette section.

Pour appeler notre fonction `dump`, nous allons passer en paramètre l'objet à enregistrer et le fichier dans lequel nous voulons enregistrer notre objet :

```
>>> import pickle
>>> from pathlib import Path
>>> mon_chemin = Path("donnees")
>>> with mon_chemin.open("wb") as fichier:
...     pickle.dump("ici, juste une chaîne", fichier)
...
>>>
```

Ici, nous enregistrerons nos objets dans le fichier `donnees`. Je ne lui ai pas donné d'extension, mais vous pouvez le faire (en évitant de préciser une extension utilisée par un programme).

Notez le mode d'ouverture : on ouvre le fichier `donnees` en mode binaire (lettre `b`).

Le fichier que Python va écrire ne sera pas très lisible si vous essayez de l'ouvrir, mais ce n'est pas le but.

Bon. Maintenant que vous savez comment faire, c'est à vous de voir comment vous voulez vous organiser ; cela dépend aussi beaucoup du projet. J'ai pris l'habitude de n'enregistrer qu'un objet par fichier, mais il n'y a aucune obligation.

On utilise la fonction `dump` du module `pickle`. Son emploi est des plus simples :

```
>>> score = {
...     "joueur 1": 5,
...     "joueur 2": 35,
...     "joueur 3": 20,
...     "joueur 4": 2,
>>> }
>>> with mon_chemin.open("wb") as fichier:
...     pickle.dump(score, fichier)
...
>>>
```

Après l'exécution de ce code, vous avez dans votre dossier de test un fichier `donnees` qui contient… eh bien, le dictionnaire contenant les scores de nos quatre joueurs.

Récupérer nos objets enregistrés

Nous allons utiliser une autre fonction définie dans notre module `pickle`. Cette fois, c'est la fonction `load`.

Commençons par créer notre objet. On lui passe le fichier dans lequel on va lire les objets. Puisqu'on va lire, on change de mode : `r` pour la lecture et `b` pour le binaire.

```
>>> with mon_chemin.open("rb") as fichier:
...     chargée = pickle.load(fichier)
...     print(chargée)
...
>>>
```

Il faut appeler la fonction `load`, qui lit l'objet depuis le fichier.

```
>>> with mon_chemin.open("rb") as fichier:
...     score_recuperé = pickle.load(fichier)
...
>>>
```

Et après cet appel, si le fichier a pu être lu, dans votre variable `score_recuperé`, vous trouvez votre dictionnaire contenant les scores. Là, c'est peut-être peu spectaculaire mais, quand vous utilisez ce module pour sauvegarder des objets devant être conservés alors que votre programme n'est pas lancé, c'est franchement très pratique.

En résumé

— On ouvre un fichier avec la méthode `open` de la classe `pathlib.Path`. Elle prend en paramètre le mode d'ouverture.
— On lit dans un fichier à l'aide de la méthode `read`.
— On écrit dans un fichier grâce à la méthode `write`.
— Un fichier doit être refermé après usage avec la méthode `close`.
— Le module `pickle` est utilisé pour enregistrer des objets Python dans des fichiers et les recharger ensuite.

Chapitre 16

Portée des variables et références

Difficulté : 

Dans ce chapitre, je vais m'attarder sur la portée des variables et sur les références. Je ne vais pas vous faire une visite guidée de la mémoire de votre ordinateur (Python est assez haut niveau pour, justement, ne pas avoir à descendre aussi bas), mais je vais simplement souligner quelques cas intéressants que vous pourriez rencontrer dans vos programmes.

Ce chapitre n'est pas indispensable, mais je ne l'écris naturellement pas pour le plaisir : vous pouvez très bien continuer à apprendre Python sans connaître précisément comment il joue avec les références, mais il est utile de le savoir.

N'hésitez pas à relire ce chapitre si vous avez un peu de mal, car les concepts présentés ne sont pas évidents.

La portée des variables

En Python, comme dans la plupart des langages, on trouve des règles qui définissent la **portée des variables**. La portée utilisée dans ce sens, c'est « quand et comment les variables sont accessibles ». Quand vous définissez une fonction, quelles sont les variables utilisables dans son corps ? Uniquement les paramètres ? Est-ce qu'on peut créer dans notre corps de fonction des variables utilisables en dehors ? Si vous ne vous êtes jamais posé ces questions, c'est normal. Je vais tout de même y répondre car elles ne sont pas dénuées d'intérêt.

Dans nos fonctions, quelles sont les variables accessibles ?

On ne change pas une équipe qui gagne : passons aux exemples dès à présent.

```
>>> a = 5
>>> def print_a():
...     """Fonction chargée d'afficher la variable a.
...     Cette variable a n'est pas passée en paramètre.
...     On suppose qu'elle a été créée en dehors et on veut
...     voir si elle est accessible depuis le corps de la
...     fonction.
...     """
...     print(f"La variable a = {a}.")
...
>>> print_a()
La variable a = 5.
>>> a = 8
>>> print_a()
La variable a = 8.
>>>
```

Surprise ! Ou peut-être pas...

La variable `a` n'est pas passée en paramètre de la fonction `print_a`. Et pourtant, Python la trouve, tant qu'elle a été définie avant **l'appel** de la fonction.

C'est là qu'interviennent les différents espaces.

L'espace local

Dans votre fonction, quand vous faites référence à une variable `a`, Python vérifie dans l'**espace local** de la fonction. Cet espace contient les paramètres qui sont passés à la fonction et les variables définies dans son corps. Python apprend ainsi que la variable `a` n'existe pas dans l'espace local de la fonction. Dans ce cas, il va regarder dans l'espace local dans lequel la fonction a été appelée. Et là, il trouve bien la variable `a` et peut donc l'afficher.

De façon générale, évitez d'appeler des variables qui ne sont pas dans l'espace local, sauf si c'est nécessaire. Pour l'instant, on ne s'intéresse qu'aux mécanismes ; on cherche juste à savoir quelles sont les variables accessibles depuis le corps d'une fonction et de quelle façon.

La portée de nos variables

Qu'advient-il des variables définies dans un corps de fonction ?

Voyons un nouvel exemple :

```
def set_var(nouvelle_valeur):
    """Fonction nous permettant de tester la portée des
    ↪ variables
    définies dans notre corps de fonction.

    """
    # On essaye d'afficher la variable var, si elle existe
    try:
        print(f"Avant l'affectation, notre variable var vaut
        ↪ {var0}.")
    except NameError:
        print("La variable var n'existe pas encore.")
    var = nouvelle_valeur
    print(f"Après l'affectation, notre variable var vaut
    ↪ {var}.")
```

Et maintenant, utilisons notre fonction :

```
>>> set_var(5)
La variable var n'existe pas encore.
Après l'affectation, notre variable var vaut 5.
>>> var
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'var' is not defined
>>>
```

Quelques explications s'imposent :

— Lors de notre appel à `set_var`, la variable `var` n'a pu être trouvée par Python : c'est normal, car nous ne l'avons pas encore définie, ni dans la fonction, ni dans le corps de notre programme. Python affecte la valeur `5` à la variable `var`, l'affiche et s'arrête.

— Au sortir de la fonction, on essaye d'afficher la variable `var`... mais Python ne la trouve pas ! En effet : elle a été définie dans le corps de la fonction (donc dans son espace local) et, à la fin de l'exécution de la fonction, l'espace est détruit... et la variable `var` avec.

Python a une règle d'accès spécifique aux variables extérieures à l'espace local : on peut les lire, mais pas les modifier. C'est pourquoi, dans notre fonction `print_a`, on arrivait à afficher une variable qui n'était pas comprise dans l'espace local de la fonction. En revanche, on ne peut la modifier, par affectation du moins. Si dans votre corps de fonction vous écrivez `var = nouvelle_valeur`, vous n'allez *en aucun cas* modifier une variable extérieure au corps.

En fait, quand Python trouve une instruction d'affectation, il change la valeur de la variable dans l'espace local de la fonction. Et rappelez-vous que cet espace local est détruit après l'appel à la fonction.

Pour résumer, et c'est ce qu'il faut retenir, *une fonction ne peut modifier, par affectation, la valeur d'une variable extérieure à son espace local.*

Cela paraît plutôt stupide au premier abord… mais pas d'impatience. Je vais relativiser cela assez rapidement.

Une fonction modifiant des objets

J'espère que vous vous en souvenez : *en Python, tout est objet.* Quand vous passez des paramètres à votre fonction, ce sont des objets qui sont transmis ; pas leur valeur, mais bien les objets eux-mêmes, ceci est très important.

Bon. On ne peut affecter une nouvelle valeur à un paramètre dans le corps de la fonction. Je ne reviens pas là-dessus. En revanche, on pourrait essayer d'appeler une méthode de l'objet qui le modifie… Voyons cela :

```
>>> def ajouter(liste, valeur_à_ajouter):
...     """Cette fonction insère une valeur à la fin
...     de la liste"""
...     liste.append(valeur_à_ajouter)
...
>>> ma_liste = ['a', 'e', 'i']
>>> ajouter(ma_liste, 'o')
>>> ma_liste
['a', 'e', 'i', 'o']
>>>
```

Cela fonctionne ! On passe en paramètres notre objet de type `list` avec la valeur à ajouter. Et la fonction appelle la méthode `append` de l'objet. Cette fois, au sortir de la fonction, notre objet a bel et bien été modifié.

Je ne vois pas pourquoi. Tu as dit qu'une fonction ne pouvait pas affecter de nouvelles valeurs aux paramètres.

Absolument. C'est là la petite subtilité dans l'histoire : on ne modifie pas du tout la valeur du paramètre, on appelle juste une méthode de l'objet. Et cela change tout. Retenez que, dans le corps de fonction, si vous spécifiez `paramètre = nouvelle_valeur`,

le paramètre ne sera modifié que dans le corps de la fonction. En revanche, si vous écrivez `paramètre.methode_pour_modifier(...)`, l'objet derrière le paramètre sera bel et bien modifié.

On peut aussi changer les attributs d'un objet, par exemple une case de la liste ou d'un dictionnaire : ces changements aussi seront effectifs au-delà de l'appel de la fonction.

Et les références, dans tout cela ?

Je vais schématiser volontairement : les variables que nous utilisons depuis le début de ce cours cachent en fait des références vers des objets.

Concrètement, j'ai présenté les variables comme ceci : un nom identifiant pointant vers une valeur. Par exemple, notre variable nommée `a` possède une valeur (disons 0).

En fait, une variable est un nom identifiant, pointant vers une référence d'un objet. La référence, c'est un peu sa position en mémoire. Cela reste plus haut niveau que les pointeurs en C par exemple ; ce n'est pas vraiment la mémoire de votre ordinateur. Et on ne manipule pas ces références directement.

Cela signifie que deux variables peuvent pointer sur le même objet.

Bah... bien sûr, rien n'empêche de créer deux variables avec la même valeur.

Non non, je ne parle pas de valeurs ici, mais d'objets. Voyons un exemple :

```
1 >>> ma_liste1 = [1, 2, 3]
2 >>> ma_liste2 = ma_liste1
3 >>> ma_liste2.append(4)
4 >>> print(ma_liste2)
5 [1, 2, 3, 4]
6 >>> print(ma_liste1)
7 [1, 2, 3, 4]
8 >>>
```

Nous créons une liste dans la variable `ma_liste1`. À la ligne 2, nous affectons `ma_liste1` à la variable `ma_liste2`. On pourrait croire que `ma_liste2` est une copie de `ma_liste1`. Toutefois, quand on ajoute `4` à `ma_liste2`, `ma_liste1` est aussi modifiée.

On dit que `ma_liste1` et `ma_liste2` contiennent une référence vers le même objet : si on modifie l'objet depuis une des deux variables, le changement sera visible depuis les deux variables.

Euh... j'essaye de faire la même chose avec des variables contenant des entiers et cela ne marche pas.

C'est normal. Les entiers, les flottants, les chaînes de caractères, n'ont aucune méthode travaillant sur l'objet lui-même. Les chaînes de caractères ne modifient pas l'objet appelant mais renvoient un nouvel objet modifié. Et comme nous venons de le voir, le processus d'affectation n'est pas du tout identique à un appel de méthode.

Et si je veux modifier une liste sans toucher à l'autre ?

Eh bien c'est impossible en définissant nos listes comme nous l'avons fait. Les deux variables pointent sur le même objet par jeu de références et donc, inévitablement, si vous modifiez l'objet, vous répercutez le changement sur les deux variables. Toutefois, il existe un moyen pour créer un nouvel objet depuis un autre :

```
1 >>> ma_liste1 = [1, 2, 3]
2 >>> ma_liste2 = list(ma_liste1) # Cela revient à copier le
   ↪  contenu de ma_liste1
3 >>> ma_liste2.append(4)
4 >>> print(ma_liste2)
5 [1, 2, 3, 4]
6 >>> print(ma_liste1)
7 [1, 2, 3]
8 >>>
```

À la ligne 2, nous avons demandé à Python de créer un nouvel objet basé sur `ma_liste1`. Du coup, les deux variables ne contiennent plus la même référence : elles modifient des objets différents. Vous pouvez utiliser la plupart des constructeurs (c'est le nom qu'on donne à `list` pour créer une liste par exemple) dans ce but. Par exemple, utilisez le constructeur `dict` en lui passant en paramètre un dictionnaire déjà construit et vous aurez en retour un autre dictionnaire, mais de même contenu. En fait, il s'agit d'une copie de l'objet, ni plus ni moins.

Pour approcher de plus près les références, vous avez la fonction `id` qui prend en paramètre un objet. Elle renvoie la position de l'objet dans la mémoire Python sous la forme d'un entier (plutôt grand). Je vous invite à réaliser quelques tests en passant divers objets en paramètres de cette fonction. Sachez au passage que `is` compare les ID des objets de part et d'autre et c'est pour cette raison que je vous ai mis en garde quant à son utilisation.

```
1 >>> ma_liste1 = [1, 2]
2 >>> ma_liste2 = [1, 2]
3 >>> ma_liste1 == ma_liste2 # On compare le contenu des listes
4 True
5 >>> ma_liste1 is ma_liste2 # On compare leur référence
6 False
7 >>>
```

Les variables globales

Il existe un moyen de modifier, dans une fonction, des variables extérieures à celle-ci. On utilise pour cela des **variables globales**.

Cette distinction entre variables locales et variables globales se retrouve dans d'autres langages et on recommande souvent d'éviter de trop les utiliser. Elles ont leur utilité, toutefois, puisque le mécanisme existe. D'un point de vue strictement personnel, tant que c'est possible, je ne travaille qu'avec des variables locales (comme nous l'avons fait depuis le début de ce cours), mais il m'arrive de faire appel à des variables globales quand c'est nécessaire ou bien plus pratique. Toutefois, ne tombez pas dans l'extrême non plus, ni dans un sens ni dans l'autre.

Le principe des variables globales

On ne peut faire plus simple. On déclare dans le corps de notre programme, donc en dehors de tout corps de fonction, une variable, tout ce qu'il y a de plus normal. Dans le corps d'une fonction qui doit modifier cette variable (changer sa valeur par affectation), on déclare à Python que la variable qui doit être utilisée dans ce corps est globale.

Python va regarder dans les différents espaces : celui de la fonction, celui dans lequel la fonction a été appelée... ainsi de suite jusqu'à mettre la main sur notre variable. S'il la trouve, il nous y donne le plein accès dans le corps de la fonction.

Cela signifie que nous pouvons y accéder en lecture (comme c'est le cas sans avoir besoin de la définir comme variable globale) mais aussi en écriture. Une fonction peut donc ainsi changer la valeur d'une variable directement.

Utiliser concrètement les variables globales

Pour déclarer à Python, dans le corps d'une fonction, que la variable qui sera utilisée doit être considérée comme globale, on utilise le mot-clé `global`. On le place généralement après la définition de la fonction, juste en-dessous de la `docstring`, ce qui permet de retrouver rapidement les variables globales sans parcourir tout le code (c'est une simple convention). On précise derrière ce mot-clé le nom de la variable à considérer comme globale :

```
>>> i = 4 # Une variable, nommée i, contenant un entier
>>> def inc_i():
...     """Fonction chargée d'incrémenter i de 1."""
...     global i # Python recherche i en dehors de l'espace
 ↪  local de la fonction
...     i += 1
...
>>> i
4
>>> inc_i()
```

```
>>> i
5
>>>
```

Si vous ne précisez pas à Python que `i` doit être considérée comme globale, vous ne pourrez pas modifier réellement sa valeur, comme nous l'avons vu plus haut. En précisant `global i`, Python donne l'accès en lecture et en écriture à cette variable.

J'utilise ce mécanisme quand je travaille sur plusieurs classes et fonctions qui doivent s'échanger des informations d'état par exemple. Il existe d'autres moyens mais vous connaissez celui-ci et, tant que vous maîtrisez bien votre code, il n'est pas plus mauvais qu'un autre.

En résumé

- Les variables locales définies avant l'appel d'une fonction seront accessibles, depuis le corps de la fonction, en lecture seule.
- Une variable locale définie dans une fonction sera supprimée après l'exécution de cette dernière.
- On peut cependant appeler les attributs et méthodes d'un objet pour le modifier durablement.
- Les variables globales se définissent à l'aide du mot-clé `global` suivi du nom de la variable préalablement créée.
- Les variables globales peuvent être modifiées depuis le corps d'une fonction (à utiliser avec prudence).

Chapitre 17

TP : un bon vieux pendu

Difficulté :

C'est le moment de mettre en pratique ce que vous avez appris. Vous n'aurez pas besoin de tout, bien entendu, mais je vais essayer de vous faire travailler un maximum de choses.

Nous allons donc coder un jeu de pendu plutôt classique. Ce n'est pas bien original, mais on va pimenter un peu l'exercice, vous allez voir.

Votre mission

Nous y voilà. Je vais vous préciser un peu la mission, sans quoi on va avoir du mal à s'entendre sur la correction.

Un jeu du pendu

Le premier point de la mission est de réaliser un jeu du pendu. Je rappelle brièvement les règles, au cas où : l'ordinateur choisit, au hasard dans une liste, un mot de huit lettres maximum. Le joueur tente de trouver les lettres qui le composent. À chaque coup, il saisit une lettre. Si elle figure dans le mot, l'ordinateur affiche le mot avec les lettres déjà trouvées. Celles qui ne le sont pas encore sont remplacées par des étoiles (*). Le joueur a 8 chances. Au-delà, il a perdu.

On va compliquer un peu les règles en demandant au joueur de donner son nom, au début de la partie, pour que le programme enregistre son score.

Le score du joueur sera simple à calculer : on prend le score courant (0 si le joueur n'a encore jamais joué) et, à chaque partie, on lui ajoute le nombre de coups restants comme points de partie. Si, par exemple, il me reste trois coups au moment où je trouve le mot, je gagne trois points.

Le côté technique du problème

Le jeu du pendu en lui-même, vous ne devriez avoir aucun problème à le mettre en place. Rappelez-vous que le joueur ne doit donner qu'une seule lettre à la fois et que le programme doit bien vérifier que c'est le cas avant de continuer. Nous allons découper notre programme en trois fichiers :

— Le fichier `donnees.py` contiendra les variables nécessaires à notre application (la liste des mots, le nombre de chances autorisées...).
— Le fichier `fonctions.py` contiendra les fonctions utiles à notre application. Là, je ne vous fais aucune liste claire, je vous conseille de bien y réfléchir, avec une feuille et un stylo si cela vous aide (Quelles sont les actions de mon programme ? Que puis-je mettre dans des fonctions ?).
— Enfin, notre fichier `pendu.py` contiendra notre jeu du pendu.

Gérer les scores

Vous avez, je l'espère, une petite idée sur la façon de coder cela... mais je vais quand même clarifier : on va enregistrer les scores du jeu dans un fichier de données, que l'on va appeler `scores` (sans aucune extension). Ils se présenteront sous la forme d'un dictionnaire : en clés, nous aurons les noms des joueurs et en valeurs les scores, sous la forme d'entiers.

Il faut gérer les cas suivants :

— Le fichier n'existe pas. Là, on crée un dictionnaire vide.
— Le joueur n'est pas dans le dictionnaire. Dans ce cas, on l'ajoute avec un score de `0`.

À vous de jouer

Vous avez l'essentiel. Peut-être pas tout ce dont vous avez besoin, cela dépend de comment vous vous organisez, mais le but est aussi de chercher ! Encore une fois, c'est un exercice pratique, ne sautez pas à la correction tout de suite, cela ne vous apprendra pas grand chose.

Bonne chance !

Correction proposée

Voici la correction que je vous propose. J'espère que vous êtes arrivés à un résultat satisfaisant, même si vous n'avez pas forcément réussi à tout faire. Si votre jeu fonctionne, c'est parfait !

▷ Télécharger les fichiers
Code web : 934163

Voici le code des trois fichiers.

donnees.py

```
"""Ce fichier définit quelques données, sous la forme de
↪  variables, utiles au programme du pendu.

Notez qu'on utilise la notation de constantes (des variables
↪  qui ne sont pas censées changer d'état pendant la partie).
↪  C'est juste une convention.

On écrit ces noms tout en majuscules.

"""

# Nombre de coups par partie
NB_COUPS = 8

# Nom du fichier stockant les scores
NOM_FICHIER_SCORES = "scores"

# Liste des mots du pendu
LISTE_MOTS = [
    "armoire",
```

```
    "boucle",
    "buisson",
    "bureau",
    "chaise",
    "carton",
    "couteau",
    "fichier",
    "garage",
    "glace",
    "journal",
    "kiwi",
    "lampe",
    "liste",
    "montagne",
    "remise",
    "sandale",
    "taxi",
    "vampire",
    "volant"
]
```

fonctions.py

```
"""Ce fichier définit des fonctions utiles pour le programme du pendu.

On utilise les données du programme contenues dans donnees.py.

"""

from pathlib import Path
import pickle
from random import choice

from donnees import *

# Gestion des scores

def récupérer_scores() -> dict[str, int]:
    """Cette fonction récupère les scores enregistrés si le fichier existe.

    Dans tous les cas, on renvoie un dictionnaire,
    soit l'objet dépicklé, soit un dictionnaire vide.

    On s'appuie sur NOM_FICHIER_SCORES défini dans donnees.py.
```

```
22
23      """
24      chemin_scores = Path(NOM_FICHIER_SCORES)
25      if chemin_scores.exists(): # Le fichier existe
26          # On le récupère
27          with chemin_scores.open("rb") as fichier_scores:
28              scores = pickle.load(fichier_scores)
29      else: # Le fichier n'existe pas
30          scores = {}
31
32      return scores
33
34  def enregistrer_scores(scores: dict[str, int]):
35      """Cette fonction se charge d'enregistrer les scores dans
        ↪  le fichier NOM_FICHIER_SCORES. Elle reçoit en
        ↪  paramètre le dictionnaire des scores à enregistrer.
36
37      """
38      chemin_scores = Path(NOM_FICHIER_SCORES)
39      with chemin_scores.open("wb") as fichier_scores: # On
        ↪  écrase les anciens scores
40          pickle.dump(scores, fichier_scores)
41
42  # Fonctions gérant les éléments saisis par l'utilisateur
43
44  def récupérer_nom_utilisateur() -> str:
45      """Fonction chargée de récupérer le nom de l'utilisateur.
46
47      Il doit être composé de 4 caractères minimum,
48      chiffres et lettres exclusivement.
49
50      Si ce nom n'est pas valide, on appelle récursivement
51      la fonction pour en obtenir un nouveau.
52
53      """
54      nom_utilisateur = input("Tapez votre nom : ")
55      # On met la première lettre en majuscule et les autres en
        ↪  minuscules
56      nom_utilisateur = nom_utilisateur.capitalize()
57      if not nom_utilisateur.isalnum() or len(nom_utilisateur) <
        ↪  4:
58          print("Ce nom est invalide.")
59          # On appelle de nouveau la fonction pour avoir un
            ↪  autre nom
60          return récupérer_nom_utilisateur()
61      else:
```

```
62          return nom_utilisateur
63
64  def récupérer_lettre() -> str:
65      """Cette fonction récupère une lettre saisie par
        ↪ l'utilisateur.
66
67      Si la chaîne récupérée n'est pas une lettre, on appelle
68      récursivement la fonction jusqu'à obtenir une lettre.
69
70      """
71      lettre = input("Tapez une lettre : ")
72      lettre = lettre.lower()
73      if len(lettre) > 1 or not lettre.isalpha():
74          print("Vous n'avez pas saisi une lettre valide.")
75          return récupérer_lettre()
76      else:
77          return lettre
78
79  # Fonctions du jeu de pendu
80
81  def choisir_mot() -> str:
82      """Cette fonction renvoie le mot choisi dans LISTE_MOTS.
83
84      On utilise la fonction choice du module random
85      (voir l'aide).
86
87      """
88      return choice(LISTE_MOTS)
89
90  def récupérer_mot_masqué(mot_complet: str, lettres_trouvées:
    ↪ set[str]) -> str:
91      """Cette fonction renvoie un mot masqué tout ou en partie,
        ↪ en fonction :
92
93      - du mot d'origine (type str)
94      - des lettres déjà trouvées (type list)
95
96      On renvoie le mot d'origine avec des * remplaçant
97      les lettres que l'on n'a pas encore trouvées.
98
99      """
100     mot_masqué = ""
101     for lettre in mot_complet:
102         if lettre in lettres_trouvées:
103             mot_masqué += lettre
104         else:
```

```
105             mot_masqué += "*"
106
107     return mot_masqué
```

pendu.py

```
1   """Ce fichier contient le jeu du pendu.
2
3   Il s'appuie sur les fichiers :
4   - donnees.py
5   - fonctions.py
6
7   """
8
9   from donnees import *
10  from fonctions import *
11
12  # On récupère les scores de la partie
13  scores = récupérer_scores()
14
15  # On récupère un nom d'utilisateur
16  utilisateur = récupérer_nom_utilisateur()
17
18  # Si l'utilisateur n'a pas encore de score, on l'ajoute
19  if utilisateur not in scores.keys():
20      scores[utilisateur] = 0 # 0 point pour commencer
21
22  # Notre variable pour savoir quand arrêter la partie
23  continuer_partie = 'o'
24
25  while continuer_partie != 'n':
26      print(f"Joueur {utilisateur}: {scores[utilisateur]}
        ↪ point(s)")
27      mot_à_trouver = choisir_mot()
28      lettres_trouvées = set()
29      mot_trouvé = récupérer_mot_masqué(mot_à_trouver,
        ↪ lettres_trouvées)
30      nb_chances = NB_COUPS
31      while mot_à_trouver != mot_trouvé and nb_chances > 0:
32          print(f"Mot à trouver {mot_trouvé} (encore
            ↪ {nb_chances} chances)")
33          lettre = récupérer_lettre()
34          if lettre in lettres_trouvées: # La lettre a déjà été
            ↪ choisie
35              print("Vous avez déjà choisi cette lettre.")
36          elif lettre in mot_à_trouver: # La lettre est dans le
            ↪ mot à trouver
```

```
37              print("Bien joué.")
38          else:
39              nb_chances -= 1
40              print("... non, cette lettre ne se trouve pas dans
                ↪  le mot...")
41          lettres_trouvées.add(lettre)
42          mot_trouvé = récupérer_mot_masqué(mot_à_trouver,
            ↪  lettres_trouvées)
43
44      # A-t-on trouvé le mot ? nos chances sont-elles épuisées ?
45      if mot_à_trouver == mot_trouvé:
46          print(f"Félicitations ! Vous avez trouvé le mot
            ↪  {mot_à_trouver}.")
47      else:
48          print("PENDU !!! Vous avez perdu.")
49
50      # On met à jour le score de l'utilisateur
51      scores[utilisateur] += nb_chances
52
53      continuer_partie = input("Souhaitez-vous continuer la
        ↪  partie (O/N) ? ")
54      continuer_partie = continuer_partie.lower()
55
56  # La partie est finie, on enregistre les scores
57  enregistrer_scores(scores)
58
59  # On affiche les scores de l'utilisateur
60  print(f"Vous finissez la partie avec {scores[utilisateur]}
    ↪  points.")
```

En résumé

Dans l'ensemble, je ne pense pas que le code soit très délicat à comprendre. Vous pouvez vous rendre compte à quel point le code du jeu est facile à lire grâce à nos fonctions. On délègue une partie de l'application à ces dernières qui s'assurent que les choses sont « bien faites ». Si un bogue survient, il est plus facile de modifier une fonction que tout un code sans aucune structure.

Par cet exemple, j'espère que vous prendrez bien l'habitude de documenter un maximum vos fichiers et fonctions. C'est réellement un bon réflexe à avoir.

N'oubliez pas la spécification de l'encodage en tête de chaque fichier, ni la mise en pause du programme sous Windows.

Troisième partie

La programmation orientée objet côté développeur

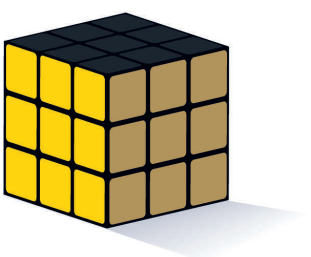
Chapitre 18

Première approche des classes

Difficulté :

Dans ce chapitre, sans plus attendre, nous allons créer nos premières classes, nos premiers attributs et nos premières méthodes. Nous allons aussi essayer de comprendre les mécanismes de la programmation orientée objet en Python.

Au-delà du mécanisme, l'orienté objet est une véritable philosophie et Python est assez différent des autres langages, en termes de philosophie justement. Restez concentrés, ce langage n'a pas fini de vous étonner !



Les classes, tout un monde

Dans la partie précédente, j'avais brièvement décrit les objets comme des variables contenant elles-mêmes des fonctions et variables. Nous sommes allés plus loin tout au long de la seconde partie, pour découvrir que nos « fonctions contenues dans nos objets » sont appelées des méthodes. En vérité, je me suis cantonné à une définition « pratique » des objets, alors que derrière la POO (Programmation Orientée Objet) se cache une véritable philosophie.

Pourquoi utiliser des objets ?

Les premiers langages de programmation n'incluaient pas l'orienté objet. Le C, pour ne citer que lui, n'utilise pas ce concept et il aura fallu attendre C++ pour exploiter la puissance de l'orienté objet dans une syntaxe proche de celle du C.

Java, apparu à peu près en même temps que Python, définit une philosophie assez différente de celle de C++ : contrairement à ce dernier, il exige que tout soit rangé dans des classes. Même l'application standard `Hello World` est contenue dans une classe.

En Python, la liberté est plus grande. Après tout, vous avez pu passer une partie de ce cours sans connaître sa façade objet. Et pourtant, *en Python, tout est objet*, vous n'avez pas oublié ? Quand vous croyez utiliser une simple variable, un module, une fonction... ce sont des objets qui se cachent derrière.

Loin de moi l'idée de comparer différents langages. Ce sur quoi je souhaite attirer votre attention, c'est que plusieurs d'entre eux intègrent l'orienté objet, chacun avec une philosophie distincte. Autrement dit, si vous avez appris l'orienté objet dans un autre langage, tel que C++ ou Java, ne tenez pas pour acquis que vous allez retrouver les même mécanismes. Gardez autant que possible l'esprit dégagé de tout préjugé sur l'approche objet de Python.

Pour l'instant, nous n'avons donc vu qu'un aspect technique de l'objet, une façon « un peu plus esthétique » de coder : il est plus simple et plus compréhensible d'écrire `ma_liste.append(5)` que `append_to_list(ma_liste, 5)`. Cependant, derrière la POO, il n'y a pas qu'un souci esthétique, loin de là.

Choix du modèle

Bon, comme vous vous en souvenez sûrement (du moins, je l'espère), une classe est un peu un modèle suivant lequel on va créer des objets. C'est dans la classe que nous allons définir nos méthodes et attributs, les attributs étant des variables contenues dans notre objet.

Qu'allons-nous modéliser ? L'orienté objet est plus qu'utile dès lors que l'on s'en sert pour modéliser, représenter des données un peu plus complexes qu'un simple nombre ou une chaîne de caractères. Bien sûr, il existe des classes que Python définit pour nous : les nombres, les chaînes et les listes en font partie. On serait toutefois bien limité si on ne pouvait créer ses propres classes.

Pour l'instant, nous allons modéliser... une personne.

Convention de nommage

Loin de moi l'idée de compliquer l'exercice, mais si on se réfère à la PEP 8 [1] de Python, il est préférable d'utiliser pour des noms de classes la convention dite **Camel Case**.

▷ PEP 8 de Python
Code web : 484505

Cette convention n'utilise pas le signe souligné _ pour séparer les mots. Le principe consiste à mettre en majuscule chaque lettre débutant un mot, par exemple : `MaClasse`.

C'est donc cette convention que je vais utiliser pour les noms de classes. Libre à vous d'en changer.

Pour définir une nouvelle classe, on utilise le mot-clé `class`.

Sa syntaxe est assez intuitive : `class NomDeLaClasse:`.

N'exécutez pas encore ce code, nous ne savons pas comment définir nos attributs et nos méthodes.

Petit exercice de modélisation : que va-t-on trouver dans les caractéristiques d'une personne ? Beaucoup de choses, vous en conviendrez. On ne va en retenir que quelques-unes : le nom, le prénom, l'âge, le lieu de résidence... allez, cela suffira.

Cela nous fait donc quatre attributs. Ce sont les variables internes à notre objet, qui vont le caractériser. Une personne telle que nous la modélisons sera caractérisée par son nom, son prénom, son âge et son lieu de résidence.

Pour définir les attributs de notre objet, il faut définir un constructeur dans notre classe. Voyons cela de plus près.

Nos premiers attributs

Nous avons défini les attributs qui allaient caractériser notre objet de classe `Personne`. Maintenant, il faut définir dans notre classe une méthode spéciale, appelée un **constructeur**, qui est appelée invariablement quand on souhaite créer un objet depuis notre classe.

Concrètement, un constructeur est une méthode de notre objet se chargeant de créer nos attributs. En vérité, c'est même la méthode qui sera appelée quand on voudra créer notre objet.

Voyons le code, ce sera plus parlant :

```
1 class Personne: # Définition de notre classe Personne
2     """Classe définissant une personne caractérisée par :
3     - son nom
```

1. Les PEP sont les « Python Enhancement Proposals », c'est-à-dire les propositions d'amélioration de Python.

```
4       - son prénom
5       - son âge
6       - son lieu de résidence
7
8       """
9
10      def __init__(self): # Notre méthode constructeur
11          """Pour l'instant, on ne va définir qu'un seul
            ↪  attribut"""
12          self.nom = "Dupont"
```

Voyons en détail :

— D'abord, la définition de la classe. Elle est constituée du mot-clé `class`, du nom de la classe et des deux points rituels « : ».

— Une `docstring` commentant la classe. Encore une fois, c'est une excellente habitude à prendre et je vous encourage à le faire systématiquement. Ce sera plus qu'utile quand vous vous lancerez dans de grands projets, notamment à plusieurs.

— La définition de notre constructeur. Comme vous le voyez, il s'agit d'une définition presque « classique » d'une fonction. Elle a pour nom `__init__`, c'est invariable : en Python, tous les constructeurs s'appellent ainsi. Nous verrons plus tard que les noms entourés de part et d'autre de deux signes soulignés `(__NomMethode__)` désignent des **méthodes spéciales**. Notez que, dans notre définition de méthode, nous passons un premier paramètre nommé `self`.

— Une nouvelle `docstring`. Je ne complique pas inutilement, je précise donc qu'on va simplement définir un seul attribut pour l'instant dans notre constructeur.

— Dans notre constructeur, nous trouvons l'instanciation de notre attribut `nom`. On crée une variable `self.nom` et on lui donne comme valeur `Dupont`. Je vais détailler un peu plus bas ce qui se passe ici.

Avant tout, pour voir le résultat en action, essayons de créer un objet issu de notre classe :

```
1   >>> bernard = Personne()
2   >>> bernard
3   <__main__.Personne object at 0x00B42570>
4   >>> bernard.nom
5   'Dupont'
6   >>>
```

Quand on demande à l'interpréteur d'afficher directement notre objet `bernard`, il nous sort quelque chose d'un peu incompréhensible... Bon, l'essentiel est la mention précisant la classe dont l'objet est issu. On peut donc vérifier que l'objet est bien issu de notre classe `Personne`. On essaye ensuite d'afficher l'attribut `nom` de notre objet `bernard` et on obtient `'Dupont'` (la valeur définie dans notre constructeur). Notez le point, encore et toujours utilisé pour une relation d'appartenance (`nom` est un attribut de l'objet `bernard`).

Quand on crée notre objet...

Quand on tape `Personne()`, on appelle le constructeur de notre classe `Personne`, d'une façon quelque peu indirecte que je ne détaillerai pas ici. Celui-ci prend en paramètre une variable un peu mystérieuse : `self`. En fait, il s'agit tout bêtement de notre objet en train de se créer. On écrit dans cet objet l'attribut `nom` le plus simplement du monde : `self.nom = "Dupont"`. À la fin de l'appel au constructeur, Python renvoie notre objet `self` modifié, avec notre attribut. On reçoit le tout dans notre variable `bernard`.

Si ce n'est pas très clair, pas de panique ! Contentez-vous de vous familiariser avec la syntaxe du constructeur Python, qui sera souvent la même, et laissez l'aspect un peu théorique de côté, pour plus tard. Nous aurons l'occasion d'y revenir avant la fin du chapitre.

Étoffons un peu notre constructeur

Bon, on avait dit quatre attributs, mais on n'en a créé qu'un. Et puis notre constructeur pourrait éviter de donner les mêmes valeurs par défaut à chaque fois, tout de même !

C'est juste. Dans un premier temps, on va se contenter de définir les autres attributs, le prénom, l'âge, le lieu de résidence. Essayez ; normalement, vous ne devriez éprouver aucune difficulté.

```
1  class Personne:
2      """Classe définissant une personne caractérisée par :
3      - son nom
4      - son prénom
5      - son âge
6      - son lieu de résidence
7
8      """
9
10     def __init__(self): # Notre méthode constructeur
11         """Constructeur de notre classe. Chaque attribut va
           ↪  être instancié
12         avec une valeur par défaut...
13
14         """
15         self.nom = "Dupont"
16         self.prénom = "Jean" # Quelle originalité
17         self.âge = 33 # Cela n'engage à rien
18         self.lieu_résidence = "Paris"
```

Cela vous paraît évident ? Voici encore un petit code d'exemple :

```
>>> jean = Personne()
>>> jean.nom
'Dupont'
>>> jean.prénom
'Jean'
>>> jean.âge
33
>>> jean.lieu_résidence
'Paris'
>>> # Jean déménage...
... jean.lieu_résidence = "Berlin"
>>> jean.lieu_résidence
'Berlin'
>>>
```

Je sens un courant d'air… les habitués de l'objet, une minute !

Cet exemple me paraît assez clair sur le principe de définition/modification des attributs et la façon d'y accéder.

Une toute petite explication s'impose en ce qui concerne la ligne 11 : dans beaucoup de cours, on déconseille de modifier un attribut d'instance (un attribut d'un objet) en écrivant simplement `objet.attribut = valeur`. Si vous venez d'un autre langage, vous avez peut-être entendu parler des accesseurs et mutateurs. Ces concepts sont repris dans certains cours Python, mais ils n'ont pas précisément lieu d'être dans ce langage. Tout cela, je le détaillerai dans le prochain chapitre. Pour l'instant, il vous suffit de savoir que, quand vous voulez modifier un attribut d'un objet, vous écrivez `objet.attribut = nouvelle_valeur`. Nous verrons les cas particuliers plus loin.

Bon. Il nous reste encore à coder un constructeur un peu plus intelligent. Pour l'instant, quel que soit l'objet créé, il possède les mêmes nom, prénom, âge et lieu de résidence. On peut les modifier par la suite, bien entendu, mais il vaut mieux fournir plusieurs paramètres au constructeur, disons… le nom et le prénom pour commencer.

```
class Personne:
    """Classe définissant une personne caractérisée par :
    - son nom
    - son prénom
    - son âge
    - son lieu de résidence

    """

    def __init__(self, nom, prénom):
        self.nom = nom
        self.prénom = prénom
        self.âge = 33
        self.lieu_résidence = "Paris"
```

Je n'ai pas ajouté de documentation dans le constructeur cette fois. La plupart du temps, les méthodes spéciales ne sont pas documentées, leur but étant assez évident. C'est un choix pratique à faire.

```
>>> bernard = Personne("Micado", "Bernard")
>>> bernard.nom
'Micado'
>>> bernard.prénom
'Bernard'
>>> bernard.âge
33
>>>
```

N'oubliez pas que le premier paramètre doit être `self`. En dehors de cela, un constructeur est une fonction plutôt classique : vous pouvez définir des paramètres, par défaut ou non, nommés ou non. Quand vous voudrez créer votre objet, vous appellerez le nom de la classe en passant entre parenthèses les paramètres à utiliser. Faites quelques tests, avec plus ou moins de paramètres ; vous saisirez très rapidement le principe.

Attributs de classe

Dans les exemples que nous avons vus jusqu'à présent, nos attributs sont contenus dans notre objet. Ils lui sont propres : si vous créez plusieurs objets, les attributs `nom`, `prénom`... ne seront pas forcément identiques d'un objet à l'autre. On peut aussi définir des attributs dans notre classe. Voyons un exemple :

```
class Compteur:
    """Cette classe possède un attribut de classe qui
    ↪  s'incrémente à chaque fois que l'on crée un objet de
    ↪  ce type

    """

    objets_créés = 0 # Le compteur vaut 0 au départ

    def __init__(self):
        """À chaque fois qu'on crée un objet, on incrémente le
        ↪  compteur"""
        Compteur.objets_créés += 1
```

On définit notre attribut directement dans le corps de la classe, sous la définition et la `docstring`, avant la définition du constructeur. Quand on veut l'appeler dans le constructeur, on préfixe son nom par celui de la classe. Et on y accède de cette façon également, en dehors de la classe. Voyez plutôt :

```
>>> Compteur.objets_créés
0
```

```
>>> a = Compteur() # On crée un premier objet
>>> Compteur.objets_créés
1
>>> # Notez que a.objets_créés donne le même résultat
... b = Compteur()
>>> Compteur.objets_créés
2
>>>
```

À chaque fois qu'on crée un objet de type `Compteur`, l'attribut de classe `objets_créés` s'incrémente de 1. Les attributs de classe sont utiles quand tous nos objets doivent avoir certaines données identiques. Nous aurons l'occasion d'en reparler par la suite.

Les méthodes, la recette

Les attributs sont des variables propres à notre objet, qui servent à le caractériser. Les méthodes sont plutôt des actions, comme nous l'avons vu dans la partie précédente, agissant sur l'objet. Par exemple, la méthode `append` de la classe `list` permet d'ajouter un élément dans l'objet `list` manipulé.

Pour créer nos premières méthodes, nous allons modéliser... un tableau. Un tableau noir, oui c'est très bien.

Notre tableau va posséder une surface (un attribut) sur laquelle on pourra écrire, que l'on pourra lire et effacer. Pour créer notre classe `TableauNoir` et notre attribut `surface`, vous ne devriez pas avoir de problème :

```
class TableauNoir:
    """Classe définissant une surface sur laquelle on peut
    ↪  écrire, que l'on peut lire et effacer, par jeu de
    ↪  méthodes. L'attribut modifié est 'surface'.

    """

    def __init__(self):
        """Par défaut, notre surface est vide"""
        self.surface = ""
```

Nous avons déjà créé une méthode, aussi vous ne devriez pas être trop surpris par la syntaxe que nous allons voir. Notre constructeur est en effet une méthode, elle en garde la syntaxe. Nous allons donc écrire notre méthode `écrire` pour commencer.

```
class TableauNoir:
    """Classe définissant une surface sur laquelle on peut
    ↪  écrire, que l'on peut lire et effacer, par jeu de
    ↪  méthodes. L'attribut modifié est 'surface'.

    """
```

```
5
6       def __init__(self):
7           """Par défaut, notre surface est vide"""
8           self.surface = ""
9
10      def écrire(self, message_à_écrire):
11          """Méthode permettant d'écrire sur la surface du
            ↪  tableau.
12
13          Si la surface n'est pas vide, on saute une ligne avant
14          d'ajouter le message à écrire.
15
16          """
17          if self.surface != "":
18              self.surface += "\n"
19          self.surface += message_à_écrire
```

Si vous copiez-collez ce code dans l'interpréteur Python, au lieu de l'écrire dans un fichier séparé, je vous conseille de ne pas coller les sauts de ligne. Sinon, vous risquez d'avoir des `IndentationError`, Python pense qu'il s'agit de la fin de la classe quand on saute une ligne pour des raisons de clarté. Vous n'aurez pas ce problème si vous placez le code dans un fichier à part.

Passons aux tests :

```
1  >>> tab = TableauNoir()
2  >>> tab.surface
3  ''
4  >>> tab.écrire("Coooool ! Ce sont les vacances !")
5  >>> tab.surface
6  "Coooool ! Ce sont les vacances !"
7  >>> tab.écrire("Joyeux Noël !")
8  >>> tab.surface
9  "Coooool ! Ce sont les vacances !\nJoyeux Noël !"
10 >>> print(tab.surface)
11 Coooool ! Ce sont les vacances !
12 Joyeux Noël !
13 >>>
```

Notre méthode se charge d'écrire sur notre surface, en sautant une ligne pour séparer chaque message.

On retrouve ici notre paramètre `self`. Il est temps de voir un peu plus en détail à quoi il sert.

Le paramètre `self`

Dans nos méthodes d'instance, qu'on appelle également des **méthodes d'objet**, on trouve ce paramètre `self` dans la définition.

Une chose a son importance : quand vous créez un nouvel objet, ici un tableau noir, ses attributs lui sont propres. C'est logique : si vous créez plusieurs tableaux noirs, ils ne vont pas tous avoir la même surface. Donc les attributs sont contenus dans l'objet.

En revanche, les méthodes sont contenues dans la classe qui définit notre objet. C'est très important. Quand vous tapez `tab.écrire(...)`, Python va chercher la méthode `écrire` non pas dans l'objet `tab`, mais dans la classe `TableauNoir`.

```
>>> tab.écrire
<bound method TableauNoir.écrire of <TableauNoir object at
 ↪  0x0347EC50>>
>>> TableauNoir.écrire
<function TableauNoir.écrire at 0x034886F0>
>>> help(TableauNoir.écrire)
Help on function écrire in module tableau:

écrire(self, message_à_écrire)
    Méthode permettant d'écrire sur la surface du tableau.

    Si la surface n'est pas vide, on saute une ligne avant
 ↪  d'ajouter le message à écrire.

>>> TableauNoir.écrire(tab, "essai")
>>> tab.surface
'essai'
>>>
```

Comme vous le voyez, quand vous tapez `tab.écrire(...)`, cela revient au même que si vous écrivez `TableauNoir.écrire(tab, ...)`. Votre paramètre `self`, c'est l'objet qui appelle la méthode. C'est pour cette raison que vous modifiez la surface de l'objet en appelant `self.surface`.

Pour résumer, quand vous devez travailler dans une méthode de l'objet sur l'objet lui-même, vous allez passer par `self`.

Le nom `self` est une très forte convention de nommage. Je vous déconseille de changer ce nom. Certains programmeurs, qui trouvent qu'écrire `self` à chaque fois est excessivement long, l'abrègent en une unique lettre `s`. *Évitez ce raccourci.* De manière générale, évitez de changer le nom. Une méthode d'instance travaille avec le paramètre `self`.

N'est-ce pas effectivement plutôt long de devoir toujours travailler avec `self` à chaque fois qu'on souhaite faire appel à l'objet ?

Cela peut le sembler, oui. C'est d'ailleurs l'un des reproches qu'on fait à Python. Certains langages travaillent implicitement sur les attributs et méthodes d'un objet sans avoir besoin de les appeler spécifiquement. Cependant, c'est moins clair et cela peut susciter la confusion. En Python, dès qu'on voit `self`, on sait que c'est un attribut ou une méthode interne à l'objet qui va être appelé.

Bon, voyons nos autres méthodes. Nous devons encore coder `lire` qui va se charger d'afficher notre surface et `effacer` qui va supprimer le contenu de notre surface. Si vous avez compris ce que je viens d'expliquer, vous devriez écrire ces méthodes sans aucun problème ; elles sont très simples. Sinon, n'hésitez pas à relire, jusqu'à ce que le déclic se fasse.

```
class TableauNoir:
    """Classe définissant une surface sur laquelle on peut
    ↪ écrire, que l'on peut lire et effacer, par jeu de
    ↪ méthodes. L'attribut modifié est 'surface'.

    """

    def __init__(self):
        """Par défaut, notre surface est vide"""
        self.surface = ""

    def écrire(self, message_à_écrire):
        """Méthode permettant d'écrire sur la surface du
        ↪ tableau. Si la surface n'est pas vide, on saute
        ↪ une ligne avant d'ajouter le message à écrire.

        """
        if self.surface != "":
            self.surface += "\n"
        self.surface += message_à_écrire

    def lire(self):
        """Cette méthode se charge d'afficher, grâce à print,
        ↪ la surface du tableau.

        """
        print(self.surface)

    def effacer(self):
        """Cette méthode permet d'effacer la surface du
        ↪ tableau"""
        self.surface = ""
```

Voici le code de test :

```
>>> tab = TableauNoir()
>>> tab.lire()

>>> tab.écrire("Salut tout le monde.")
>>> tab.écrire("La forme ?")
>>> tab.lire()
Salut tout le monde.
La forme ?
>>> tab.effacer()
>>> tab.lire()

>>>
```

Et voilà ! Avec nos méthodes bien documentées, `help(TableauNoir)` vous donne une belle description de l'utilité de votre classe. C'est très pratique, n'oubliez pas les docstrings.

Un peu d'introspection

Encore de la philosophie ?

Eh bien... le terme d'**introspection**, je le reconnais, fait penser à quelque chose de plutôt abstrait. Pourtant, vous allez très vite comprendre l'idée qui se cache derrière : Python propose plusieurs techniques pour explorer un objet, connaître ses méthodes ou attributs.

Quel est l'intérêt ? Quand on développe une classe, on sait généralement ce qu'il y a dedans, non ?

En effet. L'utilité, à notre niveau, ne saute pas encore aux yeux. Et c'est pour cela que je ne vais pas trop m'attarder dessus. Si vous ne voyez pas l'intérêt, contentez-vous de garder dans un coin de votre tête les deux techniques que nous allons voir. Arrivera un jour où vous en aurez besoin ! Pour l'heure donc, voyons plutôt l'effet :

La fonction `dir`

La première technique d'introspection que nous allons voir est la fonction `dir`. Elle prend en paramètre un objet et renvoie la liste de ses attributs et méthodes.

```
class Test:
    """Une classe de test tout simplement"""
```

```
    def __init__(self):
        """On définit dans le constructeur un unique
         ↪  attribut"""
        self.mon_attribut = "ok"

    def afficher_attribut(self):
        """Méthode affichant l'attribut 'mon_attribut'"""
        print(f"Mon attribut est {self.mon_attribut}.")
```

```
>>> # Créons un objet de la classe Test
... un_test = Test()
>>> un_test.afficher_attribut()
Mon attribut est ok.
>>> dir(un_test)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__',
 ↪  '__format__', '__ge__', '__getattribute__', '__gt__',
 ↪  '__hash__', '__init__', '__le__', '__lt__', '__module__',
 ↪  '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 ↪  '__repr__', '__setattr__', '__sizeof__', '__str__',
 ↪  '__subclasshook__', '__weakref__', 'afficher_attribut',
 ↪  'mon_attribut']
>>>
```

La fonction `dir` renvoie une liste comprenant les noms des attributs et méthodes de l'objet qu'on lui passe en paramètre. Vous remarquez que tout est mélangé : pour Python, les méthodes, les fonctions, les classes, les modules sont des objets. Ce qui différencie en premier lieu une variable d'une fonction, c'est qu'une fonction est exécutable (*callable*). `dir` se contente de renvoyer tout ce qu'il y a dans l'objet, sans distinction.

Euh, qu'est-ce que tout cela ? On n'a jamais défini toutes ces méthodes ou attributs !

Non, en effet. Nous verrons plus loin qu'il s'agit de **méthodes spéciales** utiles à Python.

L'attribut spécial `__dict__`

Par défaut, quand vous développez une classe, tous les objets construits depuis cette classe posséderont un attribut spécial `__dict__`. C'est un dictionnaire qui contient les noms des attributs associés à leurs valeurs.

Voyez plutôt :

```
>>> un_test = Test()
>>> un_test.__dict__
{'mon_attribut': 'ok'}
>>>
```

Pourquoi « attribut spécial » ?

C'est un attribut un peu particulier car ce n'est pas vous qui le créez, mais Python. Il est entouré de deux signes soulignés `__` de part et d'autre, ce qui traduit qu'il a une signification pour Python et n'est pas un attribut « standard ». Vous verrez plus loin dans ce cours des **méthodes spéciales** qui reprennent la même syntaxe.

Peut-on modifier ce dictionnaire ?

Vous le pouvez. Sachez qu'en modifiant la valeur de l'attribut, vous modifiez aussi l'attribut dans l'objet.

```
>>> un_test.__dict__["mon_attribut"] = "plus ok"
>>> un_test.afficher_attribut()
Mon attribut est plus ok.
>>>
```

De manière générale, ne faites appel à l'introspection que si vous avez une bonne raison de le faire et évitez ce genre de syntaxe. Il est quand même plus propre d'écrire `objet.attribut = valeur` que `objet.__dict__[nom_attribut] = valeur`.

Nous n'irons pas plus loin dans ce chapitre. Je pense que vous découvrirez dans la suite de ce livre l'utilité des deux méthodes que je vous ai montrées.

En résumé

— On définit une classe en suivant la syntaxe `class NomClasse:`.
— Les méthodes se définissent comme des fonctions, sauf qu'elles se trouvent dans le corps de la classe.
— Les méthodes d'instance prennent en premier paramètre `self`, l'instance de l'objet manipulé.
— On construit une instance de classe en appelant son constructeur, une méthode d'instance appelée `__init__`.
— On définit les attributs d'une instance dans le constructeur de sa classe, en suivant cette syntaxe : `self.nom_attribut = valeur`.

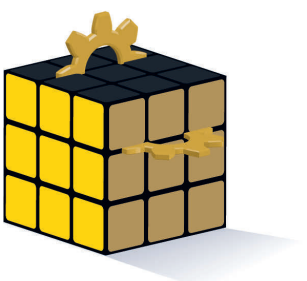
Chapitre 19

Les propriétés

Difficulté : 

Au chapitre précédent, nous avons appris à créer nos premiers attributs et méthodes. Cependant, nous avons encore assez peu parlé de la philosophie objet. Il existe quelques confusions que je vais tâcher de lever.

Nous allons découvrir les propriétés dans ce chapitre, un concept propre à Python et à quelques autres langages, comme Ruby. C'est une fonctionnalité qui, à elle seule, change l'approche objet et le principe d'encapsulation.



Qu'est-ce que l'encapsulation ?

L'encapsulation est un principe qui consiste à cacher ou protéger certaines données de notre objet. Dans la plupart des langages orientés objet, tels que C++, Java ou PHP, on va considérer que nos attributs d'objets ne doivent pas être accessibles depuis l'extérieur de la classe. Autrement dit, vous n'avez pas le droit d'appeler, depuis l'extérieur de la classe, `mon_objet.mon_attribut`.

Mais c'est stupide ! Comment fait-on pour accéder aux attributs ?

On va définir des méthodes un peu particulières, appelées des **accesseurs** et **mutateurs**. Les accesseurs donnent accès à l'attribut. Les mutateurs servent à le modifier. Concrètement, au lieu d'écrire `mon_objet.mon_attribut`, vous allez invoquer `mon_objet.get_mon_attribut()` [1]. De la même manière, pour le modifier, écrivez `mon_objet.set_mon_attribut(valeur)` [2], et non pas `mon_objet.mon_attribut = valeur`.

C'est bien tordu tout cela ! Pourquoi ne peut-on pas accéder aux attributs directement, comme on l'a fait au chapitre précédent ?

Ah mais d'abord, je n'ai pas dit que vous ne *pouviez* pas. Vous avez le droit d'accéder directement aux attributs d'un objet, comme on l'a fait au chapitre précédent. Je ne fais ici que résumer le principe d'encapsulation tel qu'on peut le trouver dans d'autres langages. En Python, c'est un peu plus subtil.

Pour répondre à la question, il est très pratique de sécuriser certaines données de notre objet, par exemple faire en sorte qu'un de ses attributs ne soit pas modifiable, ou alors mettre à jour un attribut dès qu'un autre est modifié. Les cas sont multiples et c'est très utile de pouvoir contrôler l'accès en lecture ou en écriture sur certains attributs de notre objet.

L'inconvénient de devoir écrire des accesseurs et mutateurs, comme vous l'aurez sans doute compris, c'est qu'il faut créer deux méthodes pour chaque attribut de notre classe. D'abord, c'est assez lourd. Ensuite, nos méthodes se ressemblent. Certains environnements de développement proposent, il est vrai, de créer ces accesseurs et mutateurs pour nous, automatiquement. Toutefois, cela ne résout pas vraiment le problème, vous en conviendrez.

Python a une philosophie un peu différente : pour tous les attributs dont on n'attend pas une action particulière, on va y accéder directement, comme nous l'avons fait au chapitre précédent, en écrivant simplement `mon_objet.mon_attribut`. Et pour certains, on va créer des propriétés.

1. `get` signifie « récupérer », c'est le préfixe généralement utilisé pour un accesseur.
2. `set` signifie, dans ce contexte, « modifier » ; c'est le préfixe usuel pour un mutateur.

Les propriétés à la casserole

Pour commencer, une petite précision : en C++ ou en Java par exemple, dans la définition de classe, on met en place des principes d'accès qui indiquent si l'attribut (ou le groupe d'attributs) est privé ou public. Pour schématiser, si l'attribut est public, on peut y accéder depuis l'extérieur de la classe et le modifier. S'il est privé, c'est impossible ; on doit passer par des accesseurs ou mutateurs.

En Python, il n'y a pas d'attribut privé. Tout est public. Pour faire respecter l'encapsulation propre au langage, on la fonde sur des conventions que nous allons découvrir un peu plus loin et surtout sur le bon sens de l'utilisateur de notre classe (si j'ai écrit que cet attribut est inaccessible depuis l'extérieur de la classe, je ne vais pas chercher à y accéder depuis l'extérieur de la classe).

Les propriétés sont un moyen transparent de manipuler des attributs d'objet. Elles disent à Python : « Quand un utilisateur souhaite modifier cet attribut, fais cela ». De cette façon, on peut rendre certains attributs tout à fait inaccessibles depuis l'extérieur de la classe, ou dire qu'un attribut ne sera visible qu'en lecture et non modifiable. Ou encore, on peut faire en sorte que, si on modifie un attribut, Python recalcule la valeur d'un autre attribut de l'objet.

Pour l'utilisateur, c'est absolument transparent : il croit avoir, dans tous les cas, un accès direct à l'attribut. C'est dans la définition de la classe que vous allez préciser que tel ou tel attribut doit être accessible ou modifiable grâce à certaines propriétés.

Que sont ces propriétés ?

Hum… eh bien je pense que, pour le comprendre, il vaut mieux les voir en action. Les propriétés sont des objets un peu particuliers de Python. Elles prennent la place d'un attribut et agissent différemment en fonction du contexte dans lequel elles sont appelées. Si on les appelle pour modifier l'attribut, par exemple, elles vont rediriger vers une méthode que nous avons créée, qui gère le cas où « on souhaite modifier l'attribut ».

Les propriétés en action

Une propriété est une classe, dont le nom est `property`. La syntaxe de création d'une propriété risque d'être encore assez obscur ; nous en verrons le détail dans un prochain chapitre. Pour l'heure, je vais me contenter de prendre un exemple concret et vous montrer succinctement la syntaxe.

Une propriété en lecture seule

Reprenons notre classe `Personne`. Je vais la simplifier pour l'exemple : les objets `Personne` ne vont contenir que deux attributs, le nom et le prénom.

```
class Personne:
    """Classe définissant une personne caractérisée par :
    - son nom
    - son prénom.

    """

    def __init__(self, nom, prénom):
        self.nom = nom
        self.prénom = prénom
```

Prenons un cas concret pour représenter nos propriétés : nous allons créer un *faux* attribut, appelé `nom_complet`, qui fournit directement le prénom et le nom de la personne, dans une chaîne de caractères formatée.

Une propriété n'est pas définie dans le constructeur. Pour être plus précis, une propriété travaille sur des méthodes que nous allons créer et que nous allons *encapsuler* dans notre classe. Voyons d'abord la syntaxe :

```
1  class Personne:
2      """Classe définissant une personne caractérisée par :
3      - son nom
4      - son prénom.
5
6      """
7
8      def __init__(self, nom, prénom):
9          self.nom = nom
10         self.prénom = prénom
11
12     @property
13     def nom_complet(self):
14         return f"{self.prénom} {self.nom}"
```

Une grande inspiration... une profonde expiration... recommençons... bien ! Voyons le code à partir de la ligne 12 :

— `@property` : cette ligne est aussi courte qu'obscure. Et nous n'allons pas trop nous y attarder. Sachez que c'est la façon recommandée par Python pour créer une propriété. On déclare que la méthode qui suit (celle définie à la ligne 13) n'est pas une méthode ordinaire, mais une propriété.
— `def nom_complet(self)` : vous devriez reconnaître la structure générale d'une méthode d'instance. Il n'y a pas de piège. Notez simplement le nom de la méthode, `nom_complet`, qui sera le nom de notre propriété.
— Le corps de la méthode ne devrait pas trop vous surprendre. On retourne une chaîne qui contient le prénom et le nom séparés par un espace.

Voyons comment utiliser notre propriété :

```
>>> jean = Personne(nom="Dupont", prénom="Jean")
>>> jean.nom
'Dupont'
>>> jean.prénom
'Jean'
>>> jean.nom_complet
'Jean Dupont'
>>>
```

On appelle la propriété `nom_complet` comme un attribut de notre objet. Pourtant, ce n'est pas un attribut (il n'est pas défini dans notre constructeur). Lorsque nous écrivons `jean.nom_complet`, Python va :

1. Chercher un attribut ou une méthode `nom_complet` défini sur notre objet (ou dans sa classe). À la place, il trouve une propriété.
2. Appeler la méthode liée à notre propriété pour retourner la valeur du pseudo-attribut. Ici, c'est notre méthode `nom_complet` qui retourne le prénom et le nom séparés par un espace.

En résumé :

— Les propriétés permettent de créer de *faux* attributs dans notre objet.
— Quand on y accède, Python appelle une méthode dans laquelle nous pouvons personnaliser le retour de l'attribut.

Un autre exemple vous aidera à comprendre l'intérêt. On va représenter un panier dans une boutique en ligne. Il contient une liste des objets achetés ainsi que leur prix. Voyons le code de la classe :

```
class Panier:

    """Classe représentant un panier dans une boutique en
    ↪  ligne.

    Il comprend une liste de produits achetés ainsi
    que leur prix correspondants. Pour simplifier
    l'exercice, on regroupe les produits et leur prix
    dans un dictionnaire (on ne peut acheter deux fois
    la même chose pour l'heure).

    """

    def __init__(self):
        self.produits = {}

    @property
    def total(self):
```

```
        """Retourne le prix total du panier (la somme de ses
        ↪ produits)."""
        total = 0
        for produit, prix in self.produits.items():
            total += prix

        return total

    def acheter(self, nom_produit, prix):
        """Achète un produit.

        Arguments :
            nom_produit (str) : le nom du produit à acheter.
            prix (float): le prix du produit à acheter.

        Note :
            Un panier ne peut contenir deux fois
            le même produit.

        """
        self.produits[nom_produit] = prix
```

Notez la propriété `total`. Il ne s'agit pas d'un attribut défini dans le constructeur et pourtant...

```
>>> panier = Panier()
>>> panier.acheter("brosse à dents", 1.5) # Pas bien cher
>>> panier.acheter("paquet de pâtes", 0.75) # L'inflation se
↪ perd
>>> panier.produits
{'brosse à dents': 1.5, 'paquet de pâtes': 0.75}
>>> panier.total
2.25
>>>
```

Quand on appelle `panier.total`, notre propriété est appelée. La méthode calcule rapidement et retourne le total de notre panier.

Mais... à quoi cela sert-il ?

Jusqu'ici l'intérêt vous échappe sans doute. Pourquoi ne pas coder une méthode toute simple, `calculer_total` par exemple, qui retourne notre total ? Au lieu d'écrire panier.total on écrirait panier.calculer_total() ; cela ne semble pas trop différent à première vue. En fait, tout l'intérêt se dévoile avec les propriétés en écriture.

Une propriété en lecture et écriture

Vous avez peut-être essayé de modifier notre *faux* attribut `nom_complet` ou `total` dans les deux exemples précédents. Si ce n'est pas le cas, voici ce qui se passe :

```
>>> jean = Personne(nom="Dupont", prénom="Jean")
>>> jean.nom
'Dupont'
>>> jean.prénom
'Jean'
>>> jean.nom_complet
'Jean Dupont'
>>> # Essayons de modifier le nom complet
... jean.nom_complet = "Alexandre Dupont"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

Si on cherche à modifier `nom_complet`, on obtient une erreur. En effet, notre propriété (sur `nom_complet`) est en lecture seule. Nous avons défini une méthode à appeler quand on souhaite accéder à l'attribut, mais aucune pour le modifier. Nous allons maintenant corriger cela :

```
class Personne:
    """Classe définissant une personne caractérisée par :
    - son nom
    - son prénom.

    """

    def __init__(self, nom, prénom):
        self.nom = nom
        self.prénom = prénom

    @property
    def nom_complet(self):
        return f"{self.prénom} {self.nom}"

    @nom_complet.setter
    def nom_complet(self, nouveau_nom):
        """On souhaite changer l'attribut nom_complet.

        Note :
        Le nom spécifié doit être une chaîne de caractères
        contenant un espace. On part du principe que ce qui
        est avant l'espace est le prénom, ce qui est après
        est le nom.
```

```
25
26          """
27          try:
28              prénom, nom = nouveau_nom.split(" ") # On coupe le
                ↪  nouveau nom à l'espace
29          except ValueError: # Il n'y a pas d'espace, où il y en
            ↪  a plus d'un
30              raise ValueError("préciser un prénom et nom
                ↪  séparés par un espace")
31          else:
32              self.prénom = prénom
33              self.nom = nom
```

Nous définissons une nouvelle méthode, qui sera appelée quand `nom_complet` sera changé. Cette fois, au lieu de `@property`, on a la ligne `@nom_complet.setter` juste au-dessus de notre méthode. Je ne vais pas trop détailler cette syntaxe, il faudra attendre quelques chapitres pour connaître le fin mot de l'énigme. Sachez juste qu'il s'agit de la syntaxe conseillée par Python. On précise le nom de la propriété définie plus haut (à savoir `nom_complet`) et on utilise son attribut `setter` qui contient une fonction appelée quand l'attribut est modifié.

La définition de notre fonction elle-même ne devrait pas vous surprendre. Son nom importe peu ; on garde le même nom (`nom_complet`) si jamais on a la mémoire très courte. Notez qu'en paramètre, au-delà de `self`, on a un nouvel argument : `nouveau_nom`, qui contient le nouveau nom que l'on souhaite donner à notre personne.

Le corps de la méthode risque de vous surprendre. Pour résumer, si le nouveau nom donné à l'attribut contient un espace (et pas plus), on considère que le prénom est avant l'espace et le nom est après. On découpe donc le nom (attention aux exceptions) et on change les attributs `prénom` et `nom` en conséquence.

Utilisons cette nouvelle propriété :

```
1  >>> personne = Personne(nom="Dupont", prénom="Jean")
2  >>> personne.nom_complet
3  'Jean Dupont'
4  >>> personne.nom_complet = "Jacques Durand"
5  >>> personne.prénom
6  'Jacques'
7  >>> personne.nom
8  'Durand'
9  >>> personne.nom_complet = "Pierre"
10 Traceback (most recent call last):
11   File "D:\livre\python\part3\chap2\personne.py", line 27, in
     ↪  nom_complet
12     prénom, nom = nouveau_nom.split(" ") # On coupe le
       ↪  nouveau nom à l'espace
13 ValueError: not enough values to unpack (expected 2, got 1)
14
```

```
During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "D:\livre\python\part3\chap2\personne.py", line 29, in nom_complet
    raise ValueError("préciser un prénom et nom séparés par un espace")
ValueError: préciser un prénom et nom séparés par un espace
>>>
```

À la ligne 4, nous essayons de modifier le nom complet. On précise une chaîne comportant un espace. La propriété appelle notre seconde méthode `nom_complet` qui découpe cette chaîne, change le prénom et le nom. Nous pouvons vérifier le résultat lors des lignes suivantes.

En revanche, nous essayons de changer le nom complet en donnant une chaîne de caractères sans espace. Nous avons bien géré cette erreur et notre message en donne la raison. L'exception est un peu longue car nous levons une exception en en interceptant une autre (nous ne verrons pas la syntaxe plus poussée pour raccourcir l'exception, l'important est que notre erreur s'affiche bien).

Pourrait-on faire pareil pour notre attribut `total` dans `Panier` ?

Oui... sauf que modifier le total dans notre second exemple n'est pas très logique. Si l'on modifie le total, que veut-on faire ? Modifier le prix des produits contenus dans notre panier ? Modifier tous les prix pour avoir un total qui correspond à celui spécifié ? Modifier le prix des produits les plus chers ? Si votre code supporte une logique robuste de la modification du total, vous pourriez en effet définir celui-ci comme une propriété en lecture et écriture.

Puis-je rendre une propriété impossible à lire, mais que l'on peut écrire ?

Oui, mais c'est en fait un cas assez rare et peu logique. Je vous laisse vous renseigner sur la syntaxe si vous désirez le faire. Souvenez-vous, cependant, qu'il est plus logique d'avoir soit une propriété en lecture seule, soit une propriété en lecture et écriture.

Puis-je définir plusieurs propriétés dans une classe ?

Absolument ! Nous n'avons vu qu'une définition de propriété jusqu'ici, mais rien ne vous empêche de créer plusieurs propriétés dans la même classe. En vérité, rien qu'avec les exemples que vous avez vus jusqu'ici, vous pourriez trouver d'autres utilités pour les propriétés. Je vous laisse expérimenter, c'est une grande part de l'apprentissage après tout.

Les attributs privés et les propriétés

Voyons un troisième exemple sur notre personne. Je vous invite à essayer de le faire tout seul avant de regarder la correction.

Nous avions un attribut `lieu_résidence` dans notre classe `Personne`. Comment le convertir en propriété ? Quand on change le lieu de résidence, on voudrait afficher un message pour dire que la personne a déménagé.

Vous avez tous les outils pour faire cet exercice. Une petite subtilité subsiste malgré tout : vous ne pouvez à la fois définir un attribut `lieu_résidence` et une propriété de même nom dans la classe. L'un d'entre eux (devinez lequel) sera inaccessible. Nous allons donc créer un attribut nommé `_lieu_résidence` à la place. Notez le signe souligné (_) : il indique que l'on n'est pas censé lire ou écrire dans cet attribut, sauf depuis l'intérieur de la classe. Notre constructeur pourrait donc ressembler à ce qui suit :

```
class Personne:
    """Classe définissant une personne caractérisée par :
    - son nom
    - son prénom
    - son lieu de résidence.

    """

    def __init__(self, nom, prénom):
        self.nom = nom
        self.prénom = prénom
        self._lieu_résidence = "Paris" # Attribut privé
```

Prêt pour la correction ? En voici une possible :

```
class Personne:
    """Classe définissant une personne caractérisée par :
    - son nom
    - son prénom
    - son lieu de résidence.

    """

    def __init__(self, nom, prénom):
        self.nom = nom
        self.prénom = prénom
```

```
        self._lieu_résidence = "Paris" # Attribut privé

    @property
    def lieu_résidence(self):
        return self._lieu_résidence

    @lieu_résidence.setter
    def lieu_résidence(self, nouveau_lieu):
        """Change le lieu de résidence, affiche un message."""
        print(f"{self.prénom} {self.nom} déménage à
        ↪ {nouveau_lieu}.")
        self._lieu_résidence = nouveau_lieu
```

Utilisons notre personne dans l'interpréteur :

```
>>> jean = Personne(prénom="Jean", nom="Dupont")
>>> jean.lieu_résidence
'Paris'
>>> jean.lieu_résidence = "Berlin"
Jean Dupont déménage à Berlin.
>>> jean.lieu_résidence
'Berlin'
>>> jean._lieu_résidence # NON, NE PAS FAIRE ÇA !
'Berlin'
>>> jean._lieu_résidence = "Paris" # ENCORE PIRE !!!
>>> jean._lieu_résidence
'Paris'
>>>
```

J'espère que cet exemple vous a démontré l'utilisation des attributs privés. Néanmoins, comme vous l'avez vu, rien ne vous empêche d'accéder à ces attributs depuis l'extérieur de la classe : c'est une convention (on ne devrait pas utiliser des attributs ou méthodes commençant par un signe souligné depuis l'extérieur de la classe), mais ce n'est pas une obligation absolue. En modifiant directement notre attribut `_lieu_résidence`, on contourne la propriété qui n'est donc pas appelée.

Il est théoriquement possible de masquer encore plus ces attributs privés en les préfixant par deux signes souligné (`__lieu_résidence`), mais les appeler depuis l'extérieur est toujours possible, même si un peu difficile. Cette seconde protection, à mon avis, est superflue la plupart du temps.

Les propriétés sont un concept très puissant de Python : au lieu de placer chaque attribut derrière un accesseur et un mutateur, on place certains attributs derrière des propriétés. La plupart du temps, quand vous écrivez une classe, vous n'avez besoin de protéger aucun attribut comme cela, ou alors juste un ou deux. La philosophie de Python me semble donc plus logique, mais c'est un avis assez subjectif.

En résumé

— Les propriétés permettent de contrôler l'accès à certains attributs d'une instance.
— Elles sont définis dans le corps de la classe. Des propriétés en lecture sont des méthodes surmontées de la ligne `@property`.
— Les propriétés en écriture doivent être précédées d'une ligne `@nom_propriété.setter`.
— On y fait appel ensuite en écrivant `objet.nom_propriete` comme pour n'importe quel attribut.
— La convention (humaine) utilisée pour des attributs privés est de préfixer leur nom par un signe souligné (`_`).

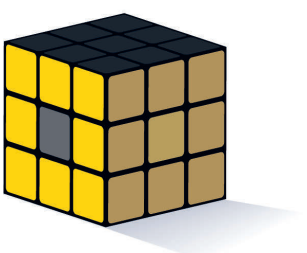
Chapitre 20

Les méthodes spéciales

Difficulté : 

Les méthodes spéciales sont des méthodes d'instance que Python reconnaît et sait utiliser, dans certains contextes. Elles peuvent servir à indiquer à Python ce qu'il doit faire quand il se retrouve devant une expression comme `mon_objet1 + mon_objet2`, voire `mon_objet[indice]`. Et, encore plus fort, elles contrôlent la façon dont un objet se crée, ainsi que l'accès à ses attributs.

Bref, voilà encore une fonctionnalité puissante et utile du langage, que je vous invite à découvrir. Prenez note du fait que je ne peux pas expliquer dans ce chapitre la totalité des méthodes spéciales. Certaines ne sont pas de notre niveau et il y en a sur lesquelles je passerai plus vite que d'autres. En cas de doute, ou si vous êtes curieux, je vous encourage d'autant plus à aller faire un tour sur le site officiel de Python.



Édition de l'objet et accès aux attributs

Vous avez déjà vu, dès le début de cette troisième partie, un exemple de **méthode spéciale** : il s'agit de notre constructeur. Une méthode spéciale, en Python, voit son nom entouré de part et d'autre par deux signes « souligné » `_`. Le nom d'une méthode spéciale prend donc la forme : `__methodespeciale__`.

Les méthodes spéciales sont parfois appelées méthodes magiques. En anglais, on parle souvent de *special method*, *magic method* ou *dunder method* (le *dunder* dans ce cas est une abréviation pour symboliser les quatre signes soulignés, appelés underscore en anglais).

Pour commencer, nous allons étudier les méthodes qui travaillent directement sur l'objet. Nous nous intéresserons ensuite, plus spécifiquement, aux méthodes donnant accès aux attributs.

Édition de l'objet

Les méthodes que nous allons voir permettent de travailler sur l'objet. Elles interviennent au moment de le créer ou de le supprimer. La première, vous devriez la reconnaître : c'est notre constructeur. Elle s'appelle `__init__`, prend un nombre variable d'arguments et contrôle la création de nos attributs.

```
1 class Exemple:
2     """Un petit exemple de classe"""
3     def __init__(self, nom):
4         """Exemple de constructeur"""
5         self.nom = nom
6         self.autre_attribut = "une valeur"
```

Pour créer notre objet, nous utilisons le nom de la classe et nous passons, entre parenthèses, les informations qu'attend notre constructeur :

```
1 mon_objet = Exemple("un premier exemple")
```

J'ai un peu simplifié ce qui se passe mais, pour l'instant, c'est tout ce qu'il vous faut retenir. Comme vous pouvez le constatez, à partir du moment où l'objet est créé, on peut accéder à ses attributs grâce à `mon_objet.nom_attribut` et exécuter ses méthodes grâce à `mon_objet.nom_methode(...)`.

Il existe également une autre méthode, `__del__`, appelée au moment de la destruction de l'objet.

La destruction ? Quand un objet se détruit-il ?

Bonne question. Il y a plusieurs cas : d'abord, quand vous voulez le supprimer explicitement, grâce au mot-clé `del` (`del mon_objet`). Ensuite, si l'espace de noms contenant

l'objet est détruit, l'objet l'est également. Par exemple, si vous instanciez l'objet dans le corps d'une fonction, à la fin de l'appel à cette dernière, la méthode `__del__` de l'objet sera appelée. Enfin, si votre objet résiste envers et contre tout pendant l'exécution du programme, il sera supprimé à la fin de l'exécution.

```
    def __del__(self):
        """Méthode appelée quand l'objet est supprimé"""
        print("C'est la fin ! On me supprime !")
```

À quoi cela peut-il bien servir, de contrôler la destruction d'un objet ?

Souvent, à rien. Python s'en sort comme un grand garçon, il n'a pas besoin d'aide. Parfois cependant, on a besoin de récupérer des informations d'état sur l'objet au moment de sa suppression. Ce n'est qu'un exemple : les méthodes spéciales sont un moyen d'exécuter des actions personnalisées sur certains objets, dans un cas précis. Si l'utilité ne saute pas aux yeux, vous pourrez en trouver une un beau jour, en codant votre projet.

Souvenez-vous que si vous ne définissez pas de méthode spéciale pour telle ou telle action, Python aura un comportement par défaut dans le contexte où cette méthode est appelée. Écrire une méthode spéciale modifie ce comportement par défaut. Dans l'absolu, vous n'êtes même pas obligés d'écrire un constructeur.

Représentation de l'objet

Deux méthodes spéciales contrôlent comment l'objet est représenté et affiché. Vous avez sûrement déjà pu constater que, quand on instancie des objets issus de nos propres classes, si on essaye de les afficher directement dans l'interpréteur ou grâce à `print`, on obtient quelque chose d'assez laid :

```
<__main__.XXX object at 0x00B46A70>
```

On a certes les informations utiles, mais pas forcément celles qu'on veut, et l'ensemble n'est pas magnifique, il faut bien le reconnaître.

La première méthode permettant de remédier à cet état de fait est `__repr__`. Elle affecte la façon dont est affiché l'objet quand on tape directement son nom. On la redéfinit quand on souhaite faciliter le débogage sur certains objets :

```
class Personne:
    """Classe représentant une personne"""

    def __init__(self, nom, prénom):
        """Constructeur de notre classe"""
        self.nom = nom
        self.prénom = prénom
        self.âge = 33
```

```
9
10      def __repr__(self):
11          """Quand on entre notre objet dans l'interpréteur"""
12          return f"Personne: nom({self.nom}),
             ↪  prénom({self.prénom}), âge({self.âge})"
```

Voici le résultat en images :

```
1  >>> p1 = Personne("Micado", "Jean")
2  >>> p1
3  Personne: nom(Micado), prénom(Jean), âge(33)
4  >>>
```

Comme vous le constatez, la méthode `__repr__` ne prend aucun paramètre (sauf, bien entendu, `self`) et renvoie la chaîne de caractères à afficher quand on entre l'objet directement dans l'interpréteur.

On obtient également cette chaîne grâce à la fonction `repr`, qui se contente d'appeler la méthode spéciale `__repr__` de l'objet passé en paramètre :

```
1  >>> p1 = Personne("Micado", "Jean")
2  >>> repr(p1)
3  'Personne: nom(Micado), prénom(Jean), âge(33)'
4  >>>
```

Il existe une seconde méthode spéciale, `__str__`, spécialement utilisée pour afficher l'objet avec `print`. Par défaut, si aucune méthode `__str__` n'est définie, Python appelle la méthode `__repr__` de l'objet. La méthode `__str__` est également appelée si vous désirez convertir votre objet en chaîne avec le constructeur `str`.

```
1  class Personne:
2      """Classe représentant une personne"""
3
4      def __init__(self, nom, prénom):
5          """Constructeur de notre classe"""
6          self.nom = nom
7          self.prénom = prénom
8          self.âge = 33
9
10     def __str__(self):
11         return f"{self.prénom} {self.nom}, âgé de {self.âge}
            ↪  ans"
```

```
1  >>> p1 = Personne("Micado", "Jean")
2  >>> print(p1)
3  Jean Micado, âgé de 33 ans
4  >>> chaîne = str(p1)
5  >>> chaîne
6  'Jean Micado, âgé de 33 ans'
7  >>>
```

Accès aux attributs de notre objet

Nous allons découvrir trois méthodes qui définissent comment accéder à nos attributs et les modifier.

La méthode `__getattr__`

La méthode spéciale `__getattr__` définit un accès à nos attributs plus large que celui que Python propose par défaut. En fait, cette méthode est appelée quand vous tapez `objet.attribut` (non pas pour modifier l'attribut mais simplement pour y accéder). Python recherche l'attribut et, *s'il ne le trouve pas dans l'objet* et si une méthode `__getattr__` existe, il l'appelle en lui passant en paramètre le nom de l'attribut recherché, sous la forme d'une chaîne de caractères.

Un petit exemple ?

```
>>> class Protégée:
...     """Classe possédant une méthode particulière d'accès
...     à ses attributs :
...     Si l'attribut n'est pas trouvé, on affiche une alerte
...     et renvoie None.
...
...     """
...
...     def __init__(self):
...         """On crée quelques attributs par défaut"""
...         self.a = 1
...         self.b = 2
...         self.c = 3
...
...     def __getattr__(self, nom):
...         """Si Python ne trouve pas l'attribut nommé nom,
...         il appelle cette méthode. On affiche une alerte
...
...         """
...         print(f"Alerte ! Il n'y a pas d'attribut {nom}
↪  ici !")
...
>>> pro = Protégée()
>>> pro.a
1
>>> pro.c
3
>>> pro.e
Alerte ! Il n'y a pas d'attribut e ici !
>>>
```

Ici, on se contente d'afficher une alerte, mais on pourrait tout aussi bien rediriger vers un autre attribut.

La méthode `__setattr__`

Cette méthode définit l'accès à un attribut destiné à être modifié. Si vous écrivez `objet.nom_attribut = nouvelle_valeur`, la méthode spéciale `__setattr__` sera appelée ainsi : `objet.__setattr__("nom_attribut", nouvelle_valeur)`. Là encore, le nom de l'attribut recherché est passé sous la forme d'une chaîne de caractères. Cette méthode permet de déclencher une action dès qu'un attribut est modifié, par exemple enregistrer l'objet :

```
1       def __setattr__(self, nom_attr, val_attr):
2           """Méthode appelée quand on écrit objet.nom_attr =
             ↪  val_attr.
3           On se charge d'enregistrer l'objet.
4
5           """
6           object.__setattr__(self, nom_attr, val_attr)
7           self.enregistrer()
```

Une explication s'impose concernant la ligne 6. J'expliquerai bien plus en détail, dans un prochain chapitre, le concept d'héritage. Pour l'instant, il vous suffit de savoir que toutes les classes que nous créons sont héritées de la classe `object`. Cela veut dire essentiellement qu'elles reprennent les mêmes méthodes. La classe `object` est définie par Python. Je disais plus haut que, si vous ne définissiez pas une certaine méthode spéciale, Python avait un comportement par défaut : ce comportement est défini par la classe `object`.

Les méthodes spéciales sont pour la plupart déclarées dans `object`. Si vous invoquez par exemple `objet.attribut = valeur` sans avoir défini de méthode `__setattr__` dans votre classe, c'est la méthode `__setattr__` de la classe `object` qui sera appelée.

En revanche, si vous redéfinissez la méthode `__setattr__` dans votre classe, la méthode appelée sera alors celle que vous définissez, et non celle de `object`. Oui mais... vous ne savez pas comment Python fait, réellement, pour modifier la valeur d'un attribut. Le mécanisme derrière la méthode vous est inconnu.

Si vous essayez, dans la méthode `__setattr__`, d'appeler `self.attribut = valeur`, vous obtenez une jolie erreur : Python veut modifier un attribut, il appelle la méthode `__setattr__` de la classe que vous avez définie, il tombe dans cette méthode sur une nouvelle affectation d'attribut, il appelle donc de nouveau `__setattr__`... et tout cela, jusqu'à l'infini ou presque. Python met en place une protection pour éviter qu'une méthode ne s'appelle elle-même à l'infini, mais cela ne règle pas le problème.

Donc, dans votre méthode `__setattr__`, il n'est pas possible de modifier un attribut de la façon que vous connaissez. À la place, on va se référer à la méthode `__setattr__` définie dans la classe `object`, la classe mère dont toutes nos classes héritent.

Si toutes ces explications vous ont paru plutôt dures, ne vous en faites pas trop : je détaillerai dans un prochain chapitre ce qu'est l'héritage, vous comprendrez sûrement mieux à ce moment.

La méthode `__delattr__`

Cette méthode spéciale est appelée quand on souhaite supprimer un attribut de l'objet, en écrivant `del objet.attribut` par exemple. Elle prend en paramètre, outre `self`, le nom de l'attribut que l'on souhaite supprimer. Voici un exemple d'une classe dont on ne peut supprimer aucun attribut :

```
def __delattr__(self, nom_attr):
    """On ne peut supprimer d'attribut, on lève l'exception
    AttributeError
    """
    raise AttributeError("Vous ne pouvez supprimer aucun
    ↪  attribut de cette classe")
```

Là encore, si vous voulez supprimer un attribut, n'utilisez pas dans votre méthode `del self.attribut`. Sinon, vous risquez de mettre Python très en colère ! Passez par `object.__delattr__` qui sait mieux que nous comment tout cela fonctionne.

Un petit bonus

Voici quelques fonctions qui font à peu près comme les précédentes mais en utilisant des chaînes de caractères pour les noms d'attributs. Vous pourrez en avoir l'usage :

```
objet = MaClasse() # On crée une instance de notre classe
getattr(objet, "nom") # Semblable à objet.nom
setattr(objet, "nom", val) # = objet.nom = val ou
↪  objet.__setattr__("nom", val)
delattr(objet, "nom") # = del objet.nom ou
↪  objet.__delattr__("nom")
hasattr(objet, "nom") # Renvoie True si l'attribut "nom"
↪  existe, False sinon
```

Peut-être ne voyez-vous pas trop l'intérêt de ces fonctions qui prennent toutes, en premier paramètre, l'objet sur lequel travailler et en second le nom de l'attribut (sous la forme d'une chaîne). Toutefois, il est parfois très pratique de travailler avec des chaînes de caractères plutôt qu'avec des noms d'attributs. D'ailleurs, c'est un peu ce que nous venons de faire, dans nos redéfinitions de méthodes accédant aux attributs.

Là encore, si l'intérêt ne saute pas aux yeux, laissez ces fonctions de côté. Vous les retrouverez par la suite.

Les méthodes de conteneur

Nous allons commencer à travailler sur ce que l'on appelle la **surcharge d'opérateurs**. Il s'agit assez simplement d'expliquer à Python quoi faire quand on utilise tel ou tel

opérateur. Nous allons ici examiner quatre méthodes spéciales qui interviennent quand on travaille sur des objets conteneurs.

Accès aux éléments d'un conteneur

Les objets conteneurs, j'espère que vous vous en souvenez, ce sont les chaînes de caractères, les listes et les dictionnaires, entre autres. Tous ont un point commun : ils contiennent d'autres objets, auxquels on accède grâce à l'opérateur `[]`.

Les trois premières méthodes que nous allons étudier sont `__getitem__`, `__setitem__` et `__delitem__`. Elles servent respectivement à définir quoi faire quand on écrit :

— `objet[index]` ;
— `objet[index] = valeur` ;
— `del objet[index]` ;

Pour cet exemple, nous allons voir une classe enveloppe de dictionnaire. Elles ressemblent à d'autres classes mais n'en sont pas réellement. Cela vous avance-t-il ?

Nous allons créer une classe nommée `ZDict`. Elle va posséder un attribut auquel on ne devra pas accéder de l'extérieur de la classe, un dictionnaire que nous appellerons `_dictionnaire`. Quand on créera un objet de type `ZDict` et qu'on voudra appeler `objet[index]`, à l'intérieur de la classe on appellera `self._dictionnaire[index]`. En réalité, notre classe fera semblant d'être un dictionnaire ; elle réagira de la même manière, mais elle n'en sera pas réellement un.

```
1  class ZDict:
2      """Classe enveloppe d'un dictionnaire"""
3
4      def __init__(self):
5          """Notre classe n'accepte aucun paramètre"""
6          self._dictionnaire = {}
7
8      def __getitem__(self, index):
9          """Cette méthode spéciale est appelée quand on écrit
           ↪  objet[index]
10         Elle redirige vers self._dictionnaire[index]
11
12         """
13         return self._dictionnaire[index]
14
15     def __setitem__(self, index, valeur):
16         """Cette méthode est appelée quand on écrit
           ↪  objet[index] = valeur
17         On redirige vers self._dictionnaire[index] = valeur
18
19         """
20         self._dictionnaire[index] = valeur
```

Vous avez un exemple d'utilisation des deux méthodes `__getitem__` et `__setitem__` qui, je pense, est assez clair. Pour `__delitem__`, je crois que c'est assez évident : elle ne prend qu'un seul paramètre qui est l'index que l'on souhaite supprimer. Vous pouvez étendre cet exemple avec d'autres méthodes présentées précédemment, notamment `__repr__` et `__str__`. N'hésitez pas, entraînez-vous, tout cela peut vous servir.

La méthode spéciale derrière le mot-clé `in`

Il existe une quatrième méthode, appelée `__contains__`, qui est utilisée quand on souhaite savoir si un objet se trouve dans un conteneur.

Voici un exemple classique :

```
ma_liste = [1, 2, 3, 4, 5]
8 in ma_liste # Revient au même que ...
ma_liste.__contains__(8)
```

Ainsi, si vous voulez que votre classe enveloppe puisse utiliser le mot-clé `in` comme une liste ou un dictionnaire, vous devez redéfinir cette méthode `__contains__` qui prend en paramètre, outre `self`, l'objet qui nous intéresse. Si l'objet est dans le conteneur, on doit renvoyer `True` ; sinon `False`.

Je vous laisse redéfinir cette méthode, vous avez toutes les indications nécessaires.

Connaître la taille d'un conteneur

Il existe enfin une méthode spéciale `__len__`, appelée quand on souhaite connaître la taille d'un objet conteneur, grâce à la fonction `len`.

`len(objet)` équivaut à `objet.__len__()`. Cette méthode spéciale ne prend aucun paramètre et renvoie une taille sous la forme d'un entier. Là encore, je vous laisse faire l'essai.

Les méthodes mathématiques

Pour cette section, nous étudier les méthodes spéciales permettant la surcharge d'opérateurs mathématiques, comme `+`, `-`, `*` et j'en passe.

Ce qu'il faut savoir

Pour cette section, nous allons utiliser un nouvel exemple, une classe capable de contenir des durées sous la forme d'un nombre de minutes et un nombre de secondes.

Voici le corps de la classe, gardez-le sous la main :

```
class Durée:
    """Classe contenant des durées sous la forme d'un nombre
    ↪  de minutes et de secondes.
```

```
    """

    def __init__(self, minutes=0, secondes=0):
        """Constructeur de la classe"""
        self.minutes = minutes # Nombre de minutes
        self.secondes = secondes # Nombre de secondes

    def __str__(self):
        """Affichage un peu plus joli de nos objets"""
        return f"{self.minutes:02}:{self.secondes:02}"
```

On définit simplement deux attributs contenant notre nombre de minutes et notre nombre de secondes, ainsi qu'une méthode pour afficher tout cela un peu mieux. Si vous vous interrogez sur l'utilisation des symboles curieux dans la chaîne formatée de `__str__`, sachez simplement que le but est de voir la durée sous la forme `MM:SS` ; pour plus d'informations sur le formatage des chaînes, consultez la page suivante :

`https://docs.python.org/fr/3/library/string.html#format-specification-mini-language`

Créons un premier objet `Durée` que nous appelons `d1`.

```
>>> d1 = Durée(3, 5)
>>> print(d1)
03:05
>>>
```

Si vous essayez de demander `d1+4`, par exemple, vous allez obtenir une erreur. Python ne sait pas comment additionner un type `Durée` et un `int`. Il ne sait même pas comment ajouter deux durées ! Nous allons donc le lui expliquer.

La méthode spéciale à redéfinir est `__add__`. Elle prend en paramètre l'objet que l'on souhaite ajouter. Voici deux lignes de code qui reviennent au même :

```
d1 + 4
d1.__add__(4)
```

Comme vous le voyez, quand vous utilisez le symbole `+` ainsi, c'est en fait la méthode `__add__` de l'objet `Durée` qui est appelée. Elle prend en paramètre l'objet que l'on souhaite ajouter, peu importe son type. Et elle doit renvoyer un objet exploitable, ici une nouvelle durée.

Jusqu'ici, nous avons utilisé la fonction `type` pour connaître le type d'un objet. Cependant, pour l'exercice suivant, nous voulons tester si un objet est d'un certain type. Pour le test, Python déconseille très fortement d'utiliser `type`. À la place, il recommande `isinstance`, qui a une syntaxe quelque peu différente :

```
>>> valeur = 3
>>> isinstance(valeur, int) # valeur est un int ?
```

```
True
>>> isinstance(valeur, float) # valeur est un float ?
False
>>> isinstance("bonjour", (str, float)) # 'bonjour' est une
 ↪  str OU un float ?
True
>>>
```

Comme vous le voyez, `isinstance` prend deux arguments : la valeur à tester et une classe. `isinstance` retourne `True` si la valeur testée est de cette classe, `False` sinon. On peut tester si une valeur est de l'une des classes spécifiées dans un tuple en second paramètre.

Revenons à `__add__` :

```
    def __add__(self, objet_à_ajouter):
        """L'objet à ajouter est un entier, le nombre de
         ↪  secondes"""
        nouvelle_durée = Durée()
        # On va copier self dans l'objet créé pour avoir la
         ↪  même durée
        nouvelle_durée.minutes = self.minutes
        nouvelle_durée.secondes = self.secondes
        # On ajoute la durée
        nouvelle_durée.secondes += objet_à_ajouter
        # Si le nombre de secondes >= 60
        if nouvelle_durée.secondes >= 60:
            nouvelle_durée.minutes += nouvelle_durée.secondes
             ↪  // 60
            nouvelle_durée.secondes = nouvelle_durée.secondes
             ↪  % 60
        # On renvoie la nouvelle durée
        return nouvelle_durée
```

Prenez le temps de comprendre le mécanisme et le petit calcul pour vous assurer d'avoir une durée cohérente. D'abord, on crée une nouvelle durée qui est l'équivalent de celle contenue dans `self`. On l'augmente du nombre de secondes à ajouter et on s'assure que le temps est cohérent (le nombre de secondes n'atteint pas 60). Si le temps n'est pas cohérent, on le corrige. On renvoie enfin notre nouvel objet modifié. Voici un petit code qui montre comment utiliser notre méthode :

```
>>> d1 = Durée(12, 8)
>>> print(d1)
12:08
>>> d2 = d1 + 54 # d1 + 54 secondes
>>> print(d2)
13:02
>>>
```

Pour mieux comprendre, remplacez `d2 = d1 + 54` par `d2 = d1.__add__(54)` : cela revient au même. Ce remplacement ne sert qu'à bien comprendre le mécanisme. Il va de soi que ces méthodes spéciales ne sont pas à appeler directement depuis l'extérieur de la classe, les opérateurs n'ont pas été inventés pour rien.

Sachez que sur le même modèle, il existe les méthodes :

— `__sub__` : surcharge de l'opérateur `-` ;
— `__mul__` : surcharge de l'opérateur `*` ;
— `__truediv__` : surcharge de l'opérateur `/` ;
— `__floordiv__` : surcharge de l'opérateur `//` (division entière) ;
— `__mod__` : surcharge de l'opérateur `%` (modulo) ;
— `__pow__` : surcharge de l'opérateur `**` (puissance) ;
— ...

Il y en a d'autres que vous pouvez consulter dans la documentation officielle de Python : `https://docs.python.org/fr/3/reference/datamodel.html` .

Tout dépend du sens

Vous l'avez peut-être remarqué, et c'est assez logique si vous avez suivi mes explications, mais écrire `objet1 + objet2` ne revient pas au même qu'écrire `objet2 + objet1` si les deux objets ont des types différents.

En effet, suivant le cas, c'est la méthode `__add__` de l'un ou l'autre des objets qui est appelée.

Cela signifie que, lorsqu'on utilise la classe `Durée`, si on écrit `d1 + 4` cela fonctionne, alors que `4 + d1` ne marche pas. En effet, la classe `int` ne sait pas quoi faire de votre objet `Durée`.

Il existe cependant une panoplie de méthodes spéciales pour faire le travail de `__add__` si vous écrivez l'opération dans l'autre sens. Il suffit de préfixer le nom des méthodes spéciales par un `r` (right).

```
    def __radd__(self, objet_à_ajouter):

        """Cette méthode est appelée si on écrit 4 + objet et
        ↪  si le premier objet (4 dans cet exemple) ne sait
        ↪  pas comment ajouter le second. On se contente de
        ↪  rediriger sur __add__ puisque, ici, cela revient au
        ↪  même : l'opération doit avoir le même résultat,
        ↪  posée dans un sens ou dans l'autre.

        """
         return self + objet_à_ajouter
```

À présent, on peut écrire `4 + d1`, cela revient au même que `d1 + 4`.

N'hésitez pas à relire ces exemples s'ils vous paraissent peu clairs.

Vous l'avez remarqué, on ne peut ajouter deux durées entre elles :

```
>>> d1 = Durée(5, 10)
>>> d2 = Durée(3, 15)
>>> print(d1 + d2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "D:\livre\python\part3\chap3\duree.py", line 25, in
  ↪ __add__
    if nouvelle_durée.secondes >= 60:
TypeError: '>=' not supported between instances of 'Durée'
↪ and 'int'
>>>
```

Nous sommes dans un cas d'exception quelque peu obscur : il semble que l'erreur soit due à un autre opérateur manquant. Cependant, en regardant attentivement, vous pourriez voir le problème : on essaye de comparer des durées et on ne sait tout simplement pas le faire. Dans ce contexte, il faut simplement modifier `__add__` pour gérer l'addition de données :

```
    def __add__(self, objet_à_ajouter):
        """L'objet à ajouter est un entier, le nombre de
        ↪ secondes, ou une durée."""
        nouvelle_durée = Durée()
        # On va copier self dans l'objet créé pour avoir la
        ↪ même durée
        nouvelle_durée.minutes = self.minutes
        nouvelle_durée.secondes = self.secondes
        # On ajoute la durée
        if isinstance(objet_à_ajouter, int):
            nouvelle_durée.secondes += objet_à_ajouter
        elif isinstance(objet_à_ajouter, Durée):
            nouvelle_durée.secondes +=
            ↪ objet_à_ajouter.secondes
            nouvelle_durée.minutes += objet_à_ajouter.minutes
        else:
            # Autant limiter l'addition
            raise TypeError("on ne peut ajouter de durée
            ↪ qu'aux entiers et autres durées")

        # Si le nombre de secondes >= 60
        if nouvelle_durée.secondes >= 60:
            nouvelle_durée.minutes += nouvelle_durée.secondes
            ↪ // 60
            nouvelle_durée.secondes = nouvelle_durée.secondes
            ↪ % 60
        # On renvoie la nouvelle durée
        return nouvelle_durée
```

```
>>> d1 = Durée(5, 10)
>>> d2 = Durée(3, 15)
>>> print(d1 + d2)
08:25
>>> d3 = Durée(12, 54)
>>> print(d1 + d2 + d3)
21:19
>>> print(d1 + 112 + d3)
19:56
>>>
```

Magnifique non ?

D'autres opérateurs

Il est également possible de surcharger les opérateurs `+=`, `-=`, etc. On préfixe cette fois-ci les noms de méthode par un `i` (increment).

Exemple de méthode `__iadd__` pour notre classe `Durée` :

```
    def __iadd__(self, objet_à_ajouter):
        """L'objet à ajouter est un entier, le nombre de
        ↪ secondes"""
        # On travaille directement sur self cette fois
        # On ajoute la durée
        self.secondes += objet_à_ajouter
        # Si le nombre de secondes >= 60
        if self.secondes >= 60:
            self.minutes += self.secondes // 60
            self.secondes = self.secondes % 60
        # On renvoie self
        return self
```

```
>>> d1 = Durée(8, 5)
>>> d1 += 128
>>> print(d1)
10:13
>>>
```

Je ne peux que vous encourager à faire des tests, pour être bien sûrs de comprendre le mécanisme. Je vous ai donné ici une façon de faire en la commentant mais, si vous ne pratiquez pas ou n'essayez pas par vous-mêmes, vous n'allez pas la retenir et vous n'allez pas forcément en comprendre la logique.

Les méthodes de comparaison

Pour finir, nous allons surcharger les opérateurs de comparaison que vous connaissez depuis quelque temps maintenant : `==`, `!=`, `<`, `>`, `<=`, `>=`.

Ces méthodes sont donc appelées si vous tentez de comparer deux objets entre eux. Comment Python sait-il que 3 est inférieur à 18 ? Une méthode spéciale de la classe `int` le permet, en simplifiant. Donc si vous voulez comparer des durées, par exemple, vous allez devoir redéfinir certaines méthodes que je vais présenter plus loin. Elles devront prendre en paramètre l'objet à comparer à `self` et renvoyer un booléen (`True` ou `False`).

Je vais me contenter d'un petit tableau récapitulatif des méthodes à redéfinir pour comparer deux objets entre eux.

Opérateur	Méthode spéciale	Résumé
`==`	`def __eq__(self, objet_à_comparer):`	Opérateur d'égalité (*equal*). Renvoie `True` si `self` et `objet_à_comparer` sont égaux, `False` sinon.
`!=`	`def __ne__(self, objet_à_comparer):`	Différent de (*non equal*). Renvoie `True` si `self` et `objet_à_comparer` sont différents, `False` sinon.
`>`	`def __gt__(self, objet_à_comparer):`	Teste si `self` est strictement supérieur (*greather than*) à `objet_à_comparer`.
`>=`	`def __ge__(self, objet_à_comparer):`	Teste si `self` est supérieur ou égal (*greater or equal*) à `objet_à_comparer`.
`<`	`def __lt__(self, objet_à_comparer):`	Teste si `self` est strictement inférieur (*lower than*) à `objet_à_comparer`.
`<=`	`def __le__(self, objet_à_comparer):`	Teste si `self` est inférieur ou égal (*lower or equal*) à `objet_à_comparer`.

Sachez que ce sont ces méthodes spéciales qui sont appelées si, par exemple, vous voulez trier une liste contenant vos objets.

Sachez également que, si Python n'arrive pas à évaluer `objet1<objet2`, il essayera l'opération inverse, soit `objet2>=objet1`. Cela vaut aussi pour les autres opérateurs de comparaison.

```
1     def __eq__(self, autre_durée):
2         """Test si self et autre_durée sont égales"""
3         return self.secondes == autre_durée.secondes and
          ↪  self.minutes == autre_durée.minutes
4
5     def __gt__(self, autre_durée):
```

```
        """Test si self > autre_durée"""
        # On calcule le nombre de secondes de self et
        ↪ autre_durée
        nb_secondes1 = self.secondes + self.minutes * 60
        nb_secondes2 = autre_durée.secondes +
        ↪ autre_durée.minutes * 60
        return nb_secondes1 > nb_secondes2
```

Des méthodes spéciales utiles à `pickle`

Vous vous souvenez de `pickle`, j'espère. Pour conclure ce chapitre sur les méthodes spéciales, nous allons en voir deux qui sont utilisées par ce module pour influencer la façon dont nos objets sont enregistrés dans des fichiers.

Prenons un cas concret, d'une utilité pratique discutable.

On crée une classe qui contient plusieurs attributs. L'un d'eux a une valeur temporaire, qui n'est utile que pendant l'exécution du programme. Si on arrête ce programme et si on le relance, on doit récupérer le même objet mais la valeur temporaire doit être remise à `0`, par exemple.

Il y a d'autres moyens d'y parvenir, je le reconnais. Toutefois, les autres applications que j'ai en tête sont plus dures à développer et à expliquer rapidement, donc gardons cet exemple.

La méthode spéciale `__getstate__`

La méthode `__getstate__` est appelée au moment de sérialiser l'objet. Quand vous voulez enregistrer l'objet à l'aide du module `pickle`, `__getstate__` est appelée juste avant l'enregistrement.

Si aucune méthode `__getstate__` n'est définie, `pickle` enregistre le dictionnaire des attributs de l'objet à sauvegarder. Je vous ai déjà expliqué qu'il est contenu dans `objet.__dict__`.

Sinon, `pickle` enregistre dans le fichier la valeur renvoyée par `__getstate__` (généralement, un dictionnaire d'attributs modifié).

Voyons un peu comment coder notre exemple grâce à `__getstate__` :

```
class Temp:
    """Classe contenant plusieurs attributs, dont un
    ↪ temporaire"""

    def __init__(self):
        """Constructeur de notre objet"""
        self.attribut_1 = "une valeur"
        self.attribut_2 = "une autre valeur"
        self.attribut_temporaire = 5
```

```
9
10      def __getstate__(self):
11          """Renvoie le dictionnaire d'attributs à sérialiser"""
12          dict_attr = dict(self.__dict__)
13          dict_attr["attribut_temporaire"] = 0
14          return dict_attr
```

Avant de revenir sur le code, observez-en les effets. Si vous tentez d'enregistrer cet objet grâce à `pickle` et si vous le récupérez ensuite depuis le fichier, vous constatez que `attribut_temporaire` est à `0`, peu importe sa valeur d'origine.

Voyons le code de `__getstate__`. La méthode ne prend aucun argument (excepté `self` puisque c'est une méthode d'instance).

Elle enregistre le dictionnaire des attributs dans une variable locale `dict_attr`. Il a le même contenu que `self.__dict__` (le dictionnaire des attributs de l'objet). En revanche, il a une référence différente. Sans cela, à la ligne suivante, au moment de modifier `attribut_temporaire`, le changement aurait été également appliqué à l'objet, ce que l'on veut éviter.

À la ligne suivante, donc, on change la valeur de l'attribut `attribut_temporaire`. Étant donné que `dict_attr` et `self.__dict__` n'ont pas la même référence, l'attribut n'est changé que dans `dict_attr` et le dictionnaire de `self` n'est pas modifié.

Enfin, on renvoie `dict_attr`. Au lieu d'enregistrer dans notre fichier `self.__dict__`, `pickle` enregistre notre dictionnaire modifié, `dict_attr`.

Si ce n'est pas assez clair, je vous encourage à tester par vous-mêmes ; essayez de modifier la méthode `__getstate__` et manipulez `self.__dict__` pour bien comprendre le code.

La méthode `__setstate__`

À la différence de `__getstate__`, la méthode `__setstate__` est appelée au moment de désérialiser l'objet. Concrètement, si vous récupérez un objet à partir d'un fichier sérialisé, `__setstate__` sera appelée après la récupération du dictionnaire des attributs.

Pour schématiser, voici l'exécution que l'on va observer derrière `pickle.load()` :

1. La fonction `pickle.load` lit le fichier.
2. Elle récupère le dictionnaire des attributs. Je vous rappelle que si aucune méthode `__getstate__` n'est définie dans notre classe, ce dictionnaire est celui contenu dans l'attribut spécial `__dict__` de l'objet au moment de sa sérialisation.
3. Ce dictionnaire récupéré est envoyé à la méthode `__setstate__` si elle existe. Si elle n'existe pas, Python considère que c'est le dictionnaire des attributs de l'objet à récupérer et écrit donc l'attribut `__dict__` de l'objet en y plaçant ce dictionnaire récupéré.

Voici le même exemple mais, cette fois, par la méthode `__setstate__` :

```
...
    def \_\_setstate\_\_(self, dict\_attr):
        """Méthode appelée lors de la désérialisation de
        ↪ l'objet"""
        dict\_attr["attribut\_temporaire"] = 0
        self.\_\_dict\_\_ = dict\_attr
```

Quelle est la différence entre les deux méthodes ?

L'objectif que nous nous étions fixé peut être atteint par ces deux méthodes. Notre classe doit mettre en œuvre soit une méthode `__getstate__`, soit une méthode `__setstate__`.

Dans le premier cas, on modifie le dictionnaire des attributs *avant* la sérialisation. Le dictionnaire enregistré est celui que nous avons modifié avec la valeur de notre attribut temporaire à `0`.

Dans le second cas, on modifie le dictionnaire d'attributs *après* la désérialisation. Le dictionnaire que l'on récupère contient un attribut `attribut_temporaire` avec une valeur quelconque (on ne sait pas laquelle) mais avant de récupérer l'objet, on met cette valeur à `0`.

Ce sont deux moyens différents, qui ici reviennent au même. À vous de choisir la meilleure méthode en fonction de vos besoins (les deux peuvent être présentes dans la même classe si nécessaire).

Là encore, je vous encourage à faire des essais si ce n'est pas très clair.

On peut enregistrer dans un fichier autre chose que des dictionnaires

Votre méthode `__getstate__` n'est pas obligée de renvoyer un dictionnaire d'attributs. Elle peut renvoyer un autre objet, un entier, un flottant, mais dans ce cas une méthode `__setstate__` devra exister pour savoir « quoi faire » avec l'objet enregistré. Si ce n'est pas un dictionnaire d'attributs, Python ne peut pas le deviner !

Là encore, je vous laisse tester si cela vous intéresse.

Je veux encore plus puissant !

`__getstate__` et `__setstate__` sont les deux méthodes les plus connues pour agir sur la sérialisation d'objets. Cependant, il en existe d'autres, plus complexes.

Si vous êtes intéressés, jetez un œil du côté de la PEP 307 (en anglais) : `https://www.python.org/dev/peps/pep-0307/`.

En résumé

— Les méthodes spéciales permettent d'influencer la manière dont Python accède aux attributs d'une instance et réagit à certains opérateurs ou conversions.
— Les méthodes spéciales sont toutes entourées de deux signes « souligné » (`_`).
— Les méthodes `__getattr__`, `__setattr__` et `__delattr__` contrôlent l'accès aux attributs de l'instance.
— Les méthodes `__getitem__`, `__setitem__` et `__delitem__` surchargent l'indexation (`[]`).
— Les méthodes `__add__`, `__sub__`, `__mul__`... surchargent les opérateurs mathématiques.
— Les méthodes `__eq__`, `__ne__`, `__gt__`... surchargent les opérateurs de comparaison.

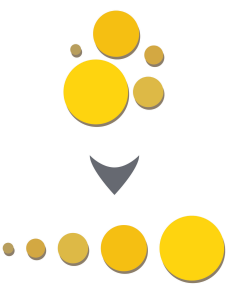
Chapitre 21

Parenthèse sur le tri en Python

Difficulté : 

Trier une liste d'informations quelconque peut s'avérer très utile... et souvent difficile. Python nous offre plusieurs techniques pour trier, que ce soit de simples listes de nombres, de chaînes de caractères ou de données plus complexes (comme des objets dont nous avons créé nous-mêmes les classes).

Ce chapitre est une parenthèse : vous pouvez aller tout de suite au chapitre suivant sans problème et revenir à celui-ci plus tard.



Première approche du tri

La première question que vous devriez vous poser est la suivante : on a une liste, on veut la trier, mais que veut-on dire par « trier » ?

Trier, c'est ordonner la liste d'une façon cohérente ; par exemple, des noms dans l'ordre alphabétique ou des nombres du plus petit au plus grand.

Dans tous les cas, trier une liste c'est la réordonner (changer son ordre, si nécessaire) selon certains critères. Il est important que vous gardiez en tête cette notion de « critères » par la suite, car nous allons en reparler.

Deux méthodes

Pour trier une séquence de données, Python nous propose deux méthodes :

1. La première est une méthode de liste. Elle s'appelle tout simplement `sort` (trier en anglais). Elle travaille sur la liste-même et change donc son ordre, si c'est nécessaire.
2. La seconde est la fonction `sorted`. Il s'agit d'une fonction **builtin**, c'est-à-dire qu'elle est disponible d'office dans Python sans avoir besoin d'importer quoi que ce soit. Contrairement à la méthode `sort` de la class `list`, `sorted` travaille sur n'importe quel type de séquence (tuple, liste ou même dictionnaire). Une importante différence avec la méthode `list.sort` est qu'elle ne modifie pas l'objet d'origine, mais en retourne un nouveau.

Voyons quelques exemples :

```
>>> prénoms = ["Jacques", "Laure", "André", "Victoire",
↪  "Albert", "Sophie"]
>>> prénoms.sort()
>>> prénoms
['Albert', 'André', 'Jacques', 'Laure', 'Sophie', 'Victoire']
>>> # Et avec la fonction 'sorted'
... prénoms = ["Jacques", "Laure", "André", "Victoire",
↪  "Albert", "Sophie"]
>>> sorted(prénoms)
['Albert', 'André', 'Jacques', 'Laure', 'Sophie', 'Victoire']
>>> prénoms
['Jacques', 'Laure', 'André', 'Victoire', 'Albert', 'Sophie']
>>>
```

Vous devriez remarquer deux choses ici :

1. D'abord, Python a trié notre liste par ordre alphabétique. Nous verrons plus tard pourquoi.
2. Le second moyen (avec la fonction `sorted`) n'a pas modifié la liste, elle a juste retourné une nouvelle liste triée. La méthode de liste `sort`, elle, a travaillé sur notre liste et l'a modifiée.

Aperçu des critères de tri

Python a trié la liste par ordre alphabétique… mais nous ne lui avons rien demandé à cet égard. En un sens, tant mieux, si c'est ce que vous vouliez faire, mais il est préférable de comprendre pourquoi. Le petit code suivant devrait vous aider à comprendre sur quelle information Python se fonde pour déterminer la meilleure méthode de tri :

```
1 >>> sorted([1, 8, -2, 15, 9])
2 [-2, 1, 8, 9, 15]
3 >>> sorted(["1", "8", "-2", "15", "9"])
4 ['-2', '1', '15', '8', '9']
5 >>>
```

La réponse se trouve dans la différence entre les lignes 1 et 3. Avez-vous trouvé ?

Pour Python, la méthode de tri dépend du type des éléments que la séquence contient. On lui a demandé de trier une liste de nombres (type `int`) et Python trie du plus petit au plus grand. Sans surprise.

À la ligne 3 cependant, on lui demande de trier la même liste, sauf que nos nombres sont devenus des chaînes de caractères (type `str`). Python choisit donc de trier la liste par ordre alphabétique.

Et si on a une liste contenant plusieurs types ?

Dans ce cas, Python va vous dire, à sa façon, qu'il ne sait pas quelle méthode de tri choisir.

```
1 >>> sorted([1, "8", "-2", "15", 9])
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 TypeError: unorderable types: str() < int()
5 >>>
```

Notre liste contient des nombres et des chaînes de caractères. Le message d'erreur n'est peut-être pas très explicite tant qu'on ne connaît pas la façon dont Python trie une séquence ; nous y reviendrons plus loin dans le chapitre.

En attendant, intéressons-nous à des types plus particuliers !

Trier avec des clés précises

Les deux moyens précédents sont pratiques, mais limités. Si nous voulons trier une liste contenant des données de types différents, selon des critères un peu plus particuliers, on va rencontrer quelques problèmes.

Considérez cet exemple : on veut conserver, dans une liste simple, les étudiants, leur âge et leur note moyenne (entre 0 et 20). On va commencer par créer une liste assez

simple, contenant des tuples. Pour chacun, on indiquera le nom de l'étudiant, son âge et sa moyenne. Voyons le code :

```
étudiants = [
    ("Clément", 14, 16),
    ("Charles", 12, 15),
    ("Oriane", 14, 18),
    ("Thomas", 11, 12),
    ("Damien", 12, 15),
]
```

Voici ce que vous obtenez si vous essayez de trier cette liste sans préciser de méthode :

```
>>> sorted(étudiants)
[
    ('Charles', 12, 15),
    ('Clément', 14, 16),
    ('Damien', 12, 15),
    ('Oriane', 14, 18),
    ('Thomas', 11, 12)
]
>>>
```

La liste ne s'affiche pas sous cette forme dans l'interpréteur par défaut, j'ai juste modifié le résultat pour qu'il soit plus lisible.

Le plus important pour nous, c'est que le tri semble s'effectuer sur la première colonne (prénoms) et par ordre alphabétique.

Maintenant, supposons que nous voulions trier par note.

Il suffit de changer les colonnes de notre liste, non ?

Oui, c'est une solution et il s'agit probablement de celle à laquelle on pense le plus vite : changer les colonnes, pour mettre les notes au début de notre tuple et, après, trier la liste.

Cependant, il y a plus simple !

L'argument `key`

La méthode `list.sort` ou la fonction `sorted` ont un paramètre optionnel, appelé `key`.

Cet argument attend... une fonction. Attendez ! Je m'explique.

La fonction à passer en paramètre prend un élément de la liste et retourne ce sur quoi doit s'effectuer le tri.

Donc la première chose est de créer une fonction ?

Oui, mais de façon assez simple : nous allons utiliser nos fonctions **lambda**. Vous vous en souvenez ? Je vous donne un petit exemple de code si besoin :

```
>>> doubler = lambda x: x * 2
>>> doubler
<function <lambda> at 0x00000000029AD1E0>
>>> doubler(8)
16
>>>
```

Il s'agit de fonctions particulières que l'on peut créer grâce au mot-clé `lambda`. Leur syntaxe est la suivante :

1. d'abord, après le mot-clé `lambda`, les arguments de la fonction à créer, séparés par des virgules s'il y en a plusieurs ;
2. ensuite, les deux points ;
3. et enfin le retour de la fonction (ici, le double du paramètre, tout simplement).

Pourquoi ce rappel sur les lambda ?

Parce que, pour trier, nous allons nous en servir. Pour préciser la méthode de tri, il nous faut une fonction qui prenne en paramètre un élément de la liste à trier et retourne l'élément qui doit être utilisé pour trier.

— L'élément de notre liste `étudiants`, c'est un tuple contenant le prénom, l'âge et la moyenne de l'étudiant.
— On veut trier le tableau des étudiants en fonction des notes (la troisième colonne du tuple).

Voici notre fonction lambda :

```
lambda colonnes: colonnes[2]
```

`colonnes` contiendra un élément de la liste des étudiants (c'est-à-dire un tuple). Si on retourne `colonnes[2]`, cela signifie qu'on veut récupérer la moyenne de l'étudiant (troisième colonne). Souvenez-vous, pour un tuple, la première colonne est toujours `0`.

Essayons à présent de trier notre liste d'étudiants en fonction de leur moyenne :

```
>>> sorted(étudiants, key=lambda colonnes: colonnes[2])
[
```

```
    ('Thomas', 11, 12),
    ('Charles', 12, 15),
    ('Damien', 12, 15),
    ('Clément', 14, 16),
    ('Oriane', 14, 18)
]
>>>
```

Si le code ne vous paraît pas clair, prenez le temps de relire les explications. Il faut un peu de temps pour s'adapter aux fonctions lambda, mais vous verrez qu'elles sont parfois très utiles.

Trier une liste d'objets

Jusqu'ici, nous avons trié des listes contenant des nombres ou des chaînes de caractères. Ce sont des objets, bien entendu, mais maintenant je voudrais vous montrer comment trier des objets issus de classes que nous avons créées.

Je vais reprendre le même exemple de notre tableau d'étudiants. Simplement, au lieu de conserver des tuples, nous allons conserver des objets, plus intuitifs et plus lisibles :

```
class Etudiant:

    """Classe représentant un étudiant.

    On représente un étudiant par son prénom,
    son âge et sa note moyenne.

    Paramètres du constructeur :
        prénom (str) : le prénom de l'étudiant.
        âge (int) : l'âge de l'étudiant.
        moyenne (int) : la moyenne de l'étudiant
        (entre 0 et 20).

    """

    def __init__(self, prénom, âge, moyenne):
        self.prénom = prénom
        self.âge = âge
        self.moyenne = moyenne

    def __repr__(self):
        return f"<Étudiant {self.prénom} (âge={self.âge},
        ↪  moyenne={self.moyenne})>"
```

Maintenant, recréons notre liste :

```
étudiants = [
    Étudiant("Clément", 14, 16),
```

```
    Étudiant("Charles", 12, 15),
    Étudiant("Oriane", 14, 18),
    Étudiant("Thomas", 11, 12),
    Étudiant("Damien", 12, 15)
]
```

Si vous essayez de trier notre liste telle quelle, vous allez rencontrer une erreur qui devrait vous sembler familière :

```
>>> étudiants
[
    <Étudiant Clément (âge=14, moyenne=16)>,
    <Étudiant Charles (âge=12, moyenne=15)>,
    <Étudiant Oriane (âge=14, moyenne=18)>,
    <Étudiant Thomas (âge=11, moyenne=12)>,
    <Étudiant Damien (âge=12, moyenne=15)>
]
>>> sorted(étudiants)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: étudiant() < étudiant()
>>>
```

Python ne sait pas comment trier nos étudiants. Il y a deux façons de le lui expliquer :

1. L'une est de définir la méthode spéciale `__lt__` de notre classe. C'est en effet cette méthode (utilisée pour la comparaison) qui est utilisée par Python pour trier une liste, en comparant chacun de ses éléments. La méthode `__lt__` *(lower than)* correspond à l'opérateur `<`.
2. On peut aussi utiliser l'argument `key`, comme nous l'avons fait précédemment.

Redéfinir la méthode `__lt__` est une bonne idée si notre objet est un nombre (par exemple une durée ou bien une heure). Dans notre exemple, il est plus pertinent d'utiliser l'argument `key` de la fonction `sorted` (ou de la méthode `list.sort`).

Saurez-vous trier cette liste d'étudiants en fonction de leur moyenne ?

Voici le code :

```
>>> sorted(étudiants, key=lambda étudiant: étudiant.moyenne)
[
    <Étudiant Thomas (âge=11, moyenne=12)>,
    <Étudiant Charles (âge=12, moyenne=15)>,
    <Étudiant Damien (âge=12, moyenne=15)>,
    <Étudiant Clément (âge=14, moyenne=16)>,
    <Étudiant Oriane (âge=14, moyenne=18)>
]
>>>
```

On obtient la même chose que dans notre exercice précédent, quand nous utilisions des tuples. Cette méthode est plus lisible.

Trier dans l'ordre inverse

Il arrive souvent que l'on veuille trier dans l'ordre inverse ; par exemple, nos étudiants du plus âgé au plus jeune.

Une solution consiste à trier et ensuite à inverser la liste mais, là encore, il existe plus rapide : l'argument `reverse`.

C'est un `booléen` que l'on peut passer à la méthode de liste `sort` ou à la fonction `sorted`.

Essayons par exemple de trier nos étudiants par ordre inverse d'âge :

```
>>> sorted(étudiants, key=lambda étudiant: étudiant.âge,
 ↪  reverse=True)
[
    <Étudiant Clément (âge=14, moyenne=16)>,
    <Étudiant Oriane (âge=14, moyenne=18)>,
    <Étudiant Charles (âge=12, moyenne=15)>,
    <Étudiant Damien (âge=12, moyenne=15)>,
    <Étudiant Thomas (âge=11, moyenne=12)>
]
>>>
```

Plus rapide et plus efficace

Les méthodes de tri que nous avons vues jusqu'ici sont très pratiques. Leur plus grand inconvénient est de reposer sur des fonctions lambda. Il est vrai que leur définition est assez rapide (et, une fois qu'on s'est habitué à la syntaxe, assez lisible). Cependant, ces fonctions ne sont pas le meilleur choix en ce qui concerne la rapidité d'exécution, si vous voulez trier une liste contenant beaucoup d'objets.

Tu as dit que le paramètre `key` attendait une fonction ; ne peut-on définir une fonction « ordinaire » ?

Si si. C'est tout à fait possible. La plupart du temps toutefois, une des fonctions du module `operator` que nous allons découvrir fait très bien le travail.

Les fonctions du module `operator`

Le module `operator` propose plusieurs fonctions qui vont s'avérer utiles pour nous, dans ce cas précis. Nous allons nous intéresser tout particulièrement aux fonctions `itemgetter` et `attrgetter`, mais sachez qu'il en existe d'autres et que le module `operator` n'est pas uniquement utile pour le tri, loin s'en faut.

Trier une liste de tuples

D'abord, voyons notre exemple avec les tuples :

```
étudiants = [
    ("Clément", 14, 16),
    ("Charles", 12, 15),
    ("Oriane", 14, 18),
    ("Thomas", 11, 12),
    ("Damien", 12, 15)
]
```

Si on veut trier par moyenne ascendante, nous avons vu qu'il suffisait d'écrire :

```
sorted(étudiants, key=lambda étudiant: étudiant[2])
```

Pour réaliser la même chose sans lambda, utilisons la fonction `itemgetter` du module `operator` :

```
from operator import itemgetter
sorted(étudiants, key=itemgetter(2))
```

On appelle la fonction `itemgetter` avec le paramètre `2`. Un objet `operator.itemgetter` est créé et passé au paramètre `key` de la fonction `sorted`. Ensuite, pour chaque étudiant contenu dans notre liste, l'objet `operator.itemgetter` est appelé et retourne sa note moyenne.

Au final, on obtient le même résultat qu'avec notre fonction **lambda**, mais cette méthode est plus rapide sur un grand nombre de données et, une fois qu'on s'est habitué à son aspect, plus facile à lire.

Trier une liste d'objets

On peut faire la même chose si on parcourt une liste d'objets mais, cette fois, on utilise la fonction `attrgetter` :

```
from operator import attrgetter
sorted(étudiants, key=attrgetter("moyenne"))
```

Le système est le même, sauf que l'on travaille ici sur une liste d'objets et que le calcul est fait sur un attribut de l'objet (ici `moyenne`) au lieu d'un tuple.

Trier selon plusieurs critères

Trier selon un critère, c'est déjà très bien, mais combiner plusieurs critères, c'est encore mieux. Disons que nous voulons trier nos étudiants par âge et note moyenne. Cela revient à dire que le tri se fera par âge, mais si deux étudiants ont le même âge, c'est leur moyenne qui servira à les classer.

Il n'y a rien de nouveau ; vous devez juste passer un nouveau paramètre à la fonction `attrgetter` :

```
>>> sorted(étudiants, key=attrgetter("âge", "moyenne"))
[
    <Étudiant Thomas (âge=11, moyenne=12)>,
    <Étudiant Charles (âge=12, moyenne=15)>,
    <Étudiant Damien (âge=12, moyenne=15)>,
    <Étudiant Clément (âge=14, moyenne=16)>,
    <Étudiant Oriane (âge=14, moyenne=18)>
]
>>>
```

Vous avez peut-être remarqué que l'ordre de Charles et Damien dans la liste est identique à avant, même si d'autres étudiants ont changé de place : en effet, comme ils ont le même âge et la même moyenne, leur ordre n'est pas modifié par Python.

Cette propriété est appelée « stabilité ». Si deux éléments de la séquence à comparer sont identiques, leur ordre est conservé.

Cette propriété du tri en Python permet de chaîner nos tris.

Chaînage de tris

Pour vous montrer un exemple concret, nous allons changer d'objets et travailler sur un inventaire de produits avec leur prix et la quantité vendue.

```
class LigneInventaire:

    """Classe représentant une ligne d'un inventaire de vente.

    Attributs attendus par le constructeur :
        produit (str) : le nom du produit.
        prix (int) : le prix unitaire du produit
        quantité (int) : la quantité vendue du produit.

    """

    def __init__(self, produit: str, prix: int | float,
    ↪  quantité: int):
        self.produit = produit
        self.prix = prix
        self.quantité = quantité

    def __repr__(self) -> str:
        return f"<Ligne d'inventaire {self.produit}
        ↪  ({self.prix}X{self.quantité})>"

# Création de l'inventaire
inventaire = [
    LigneInventaire("pomme rouge", 1.2, 19),
```

```
    LigneInventaire("orange", 1.4, 24),
    LigneInventaire("banane", 0.9, 21),
    LigneInventaire("poire", 1.2, 24)
]
```

On veut trier cette liste par prix et par quantité. Facile, c'est ce qu'on a fait un peu plus haut :

```
from operator import attrgetter
sorted(inventaire, key=attrgetter("prix", "quantité"))
```

Cela renvoie le résultat suivant :

```
[
    <Ligne d'inventaire banane (0.9X21)>,
    <Ligne d'inventaire pomme rouge (1.2X19)>,
    <Ligne d'inventaire poire (1.2X24)>,
    <Ligne d'inventaire orange (1.4X24)>
]
```

Qu'en est-il si vous voulez trier par prix croissant et par quantité décroissante (c'est-à-dire que, si deux lignes d'inventaire ont le même prix, alors on trie dans l'ordre décroissant de quantité) ?

Le plus simple ici est de réaliser deux tris en utilisant la propriété de stabilité : on va trier d'abord par notre second critère et ensuite par le premier. Ici, nous allons donc trier d'abord par ordre décroissant de quantité, puis ensuite par ordre croissant de prix.

Si vous vous demandez pourquoi, faites plusieurs essais (dans l'ordre que j'ai indiqué et dans l'ordre inverse). Si cela vous aide, essayez d'écrire l'inventaire sur une feuille et de trier dans un ordre et dans l'autre.

Voici le code pour notre tri :

```
inventaire.sort(key=attrgetter("quantité"), reverse=True)
sorted(inventaire, key=attrgetter("prix"))
```

Et vous devriez obtenir :

```
[
    <Ligne d'inventaire banane (0.9X21)>,
    <Ligne d'inventaire poire (1.2X24)>,
    <Ligne d'inventaire pomme rouge (1.2X19)>,
    <Ligne d'inventaire orange (1.4X24)>
]
```

Cette solution fait appel à la méthode de liste `sort`, mais on aurait pu aussi utiliser la fonction `sorted`.

Regardez surtout l'ordre dans lequel la poire et la pomme rouge apparaissent : les deux lignes d'inventaire ont le même prix mais, puisque la poire a été vendue en plus grande quantité, elle apparaît en premier. Ceci n'aurait pas été possible sans la stabilité ; le second tri (par prix) aurait complètement modifié l'ordre de notre liste, rendant inutile notre premier tri (par quantité inverse).

Voilà pour ce tour d'horizon des méthodes de tri proposées par Python. Sachez que vous retrouverez les fonctions clés (souvent en paramètre `key` d'une fonction) pour d'autres usages que le tri.

En résumé

— Le tri en Python se fait grâce à la méthode `sort`, qui modifie la liste d'origine, et à la fonction `sorted`, qui renvoie une nouvelle liste.
— On spécifie des fonctions clés grâce à l'argument `key`. Elles sont appelées pour chaque élément de la séquence à trier et retournent le critère du tri.
— Le module `operator` propose les fonctions `itemgetter` et `attrgetter` qui sont très utiles en tant que fonctions clés, si on veut trier une liste de tuples ou une liste d'objets selon un attribut.
— Le tri en Python est « stable », c'est-à-dire que l'ordre de deux éléments dans la liste n'est pas modifié s'ils sont égaux. Cette propriété permet le chaînage de tris.

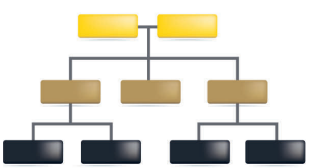
Chapitre 22

L'héritage

Difficulté : 

J'entends souvent dire qu'un langage de programmation orienté objet n'incluant pas l'héritage serait incomplet, sinon inutile. Après avoir découvert par moi-même cette fonctionnalité et les techniques qui en découlent, je suis forcé de reconnaître que sans l'héritage, le monde serait moins beau !

En quoi cette fonctionnalité est-elle si utile ? Nous allons le voir, bien entendu. Et je vais surtout essayer de vous montrer des exemples d'applications. Car très souvent, quand on découvre l'héritage, on ne sait pas trop quoi en faire... Ne vous attendez donc pas à un chapitre où vous n'allez faire que coder. Vous allez devoir vous pencher sur de la théorie et travailler sur quelques exemples de modélisation.



Pour bien commencer

Je ne vais pas faire durer le suspense plus longtemps : l'héritage est une fonctionnalité objet qui permet de déclarer que telle classe sera elle-même modelée sur une autre classe, qu'on appelle la classe parente, ou la **classe mère**. Concrètement, si une classe `b` **hérite** de la classe `a`, les objets créés sur le modèle de la classe `b` auront accès aux méthodes et attributs de la classe `a`.

Et c'est tout ? Cela ne sert à rien !

Non, ce n'est pas tout et, si, cela sert énormément, mais vous allez devoir me laisser un peu de temps pour vous en montrer l'intérêt.

La première chose, c'est que la classe `b` dans notre exemple ne se contente pas de reprendre les méthodes et attributs de la classe `a` : elle va en définir d'autres en plus, qui lui seront propres. Et elle va également redéfinir certaines méthodes de la classe mère.

Prenons un exemple simple : on a une classe `Animal` permettant de définir des animaux. Les animaux tels que nous les modélisons ont certains attributs (le régime : carnivore ou herbivore) et certaines méthodes (manger, boire, crier…).

On définit maintenant une classe `Chien` qui hérite de `Animal`, c'est-à-dire qu'elle en reprend les méthodes. Nous verrons plus loin ce que cela implique exactement.

Si vous ne voyez pas très bien dans quel cas on fait hériter une classe d'une autre, faites un test similaire au suivant :

— On fait hériter la classe `Chien` de `Animal` parce qu'*un chien est un animal.*
— On ne fait pas hériter `Animal` de `Chien` parce qu'*un animal n'est pas forcément un chien.*

Sur ce modèle, vous pouvez vous rendre compte qu'*une voiture est un véhicule.* La classe `Voiture` pourrait donc hériter de `Vehicule`.

Intéressons-nous à présent au code.

L'héritage simple

On oppose l'**héritage simple**, dont nous venons de voir les aspects théoriques, à l'**héritage multiple** que nous découvrirons dans la prochaine section.

Il est temps d'en aborder la syntaxe. Nous allons définir une première classe `A` et une seconde classe `B` qui hérite de `A`.

```
1 class A:
2     """Classe A, pour illustrer notre exemple d'héritage"""
3     pass # On laisse la définition vide, ce n'est qu'un
      ↪  exemple
```

```
4
5   class B(A):
6       """Classe B, qui hérite de A.
7
8       Elle reprend les mêmes méthodes et attributs (dans cet
9       exemple, la classe A ne possède de toute façon ni méthode
10      ni attribut).
11
12      """
13      pass
```

Vous expérimenterez par la suite sur des exemples plus constructifs. Pour l'instant, l'important est de bien noter la syntaxe, qui est des plus simples : `class MaClasse(MaClasseMere):`. Dans la définition, entre le nom et les deux points, vous précisez entre parenthèses la classe dont il faut hériter. Dans un premier temps, toutes les méthodes de la classe `A` se retrouveront dans la classe `B`.

J'ai essayé de définir des constructeurs dans les deux classes mais, dans la classe fille, je ne retrouve pas les attributs déclarés dans ma classe mère. Est-ce normal ?

Tout à fait. Rappelez-vous : je vous ai dit que les méthodes étaient définies dans la classe, alors que les attributs étaient directement déclarés dans l'instance d'objet (`self`).

Quand une classe `B` hérite d'une classe `A`, les objets de type `B` reprennent bel et bien les méthodes de la classe `A` en même temps que celles propres à la classe `B`. Toutefois, assez logiquement, ce sont celles de la classe `B` qui sont appelées d'abord.

Si vous écrivez `objet_de_type_b.ma_méthode()`, Python va chercher `ma_méthode` dans la classe `B` dont l'objet est directement issu. S'il ne l'y trouve pas, il va chercher récursivement dans les classes dont hérite `B`, c'est-à-dire `A` dans notre exemple. Ce mécanisme est très important : il induit que si aucune méthode n'a été redéfinie dans la classe, on cherche dans la classe mère. On peut ainsi redéfinir une certaine méthode dans une classe et laisser d'autres directement hériter de la classe mère.

Voici un petit code d'exemple :

```
1   class Personne:
2       """Classe représentant une personne"""
3       def __init__(self, nom):
4           """Constructeur de notre classe"""
5           self.nom = nom
6           self.prénom = "Martin"
7
8       def __str__(self):
9           """Méthode appelée lors d'une conversion de l'objet en
            ↪  chaîne"""
10          return f"{self.prénom} {self.nom}"
11
```

```
class AgentSpécial(Personne):
    """Classe définissant un agent spécial.

    Elle hérite de la classe Personne.

    """

    def __init__(self, nom, matricule):
        """Un agent se définit par son nom et son matricule"""
        self.nom = nom
        self.matricule = matricule

    def __str__(self):
        return f"Agent {self.nom}, matricule {self.matricule}"
```

Vous voyez ici un exemple d'héritage simple. Seulement, si vous essayez de créer des agents spéciaux, vous risquez d'avoir de drôles de surprises :

```
>>> agent = AgentSpécial("Fisher", "18327-121")
>>> agent.nom
'Fisher'
>>> print(agent)
Agent Fisher, matricule 18327-121
>>> agent.prénom
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'AgentSpécial' object has no attribute
↪  'prénom'
>>>
```

Argh... mais n'avais-tu pas dit qu'une classe reprenait les méthodes et attributs de sa classe mère ?

Si mais, en suivant bien l'exécution, vous allez comprendre : tout commence à la création de l'objet. Quel constructeur appeler ? S'il n'y avait pas de constructeur défini dans notre classe `AgentSpécial`, Python appellerait celui de `Personne`. Cependant, il en existe bel et bien un dans la classe `AgentSpécial` et c'est donc celui-ci qui est appelé. Dans ce constructeur, on définit deux attributs, `nom` et `matricule`. Et c'est tout : le constructeur de la classe `Personne` n'est pas appelé (il vous faudrait l'appeler explicitement dans le constructeur d'`AgentSpécial`).

Dans le premier chapitre, je vous ai expliqué que `mon_objet.ma_méthode()` revenait au même que `MaClasse.ma_méthode(mon_objet)`. Dans `ma_méthode`, le premier paramètre `self` sera `mon_objet`. Nous allons nous servir de cette équivalence. La plupart du temps, écrire `mon_objet.ma_méthode()` suffit mais, dans une relation d'héritage, il peut y avoir plusieurs méthodes du même nom définies dans différentes classes. Laquelle appeler ? Python choisit, s'il la trouve, celle déclarée directement dans

la classe dont est issu l'objet et, sinon, parcourt la hiérarchie de l'héritage. Cependant, on peut aussi se servir de la notation `MaClasse.ma_méthode(mon_objet)` pour appeler une méthode précise d'une classe précise. Et cela est utile dans notre cas :

```
class Personne:
    """Classe représentant une personne"""
    def __init__(self, nom):
        """Constructeur de notre classe"""
        self.nom = nom
        self.prénom = "Martin"

    def __str__(self):
        """Méthode appelée lors d'une conversion de l'objet en
        ↪ chaîne"""
        return f"{self.prénom} {self.nom}"

class AgentSpécial(Personne):
    """Classe définissant un agent spécial.

    Elle hérite de la classe Personne.

    """

    def __init__(self, nom, matricule):
        # On appelle explicitement le constructeur de
        ↪ Personne:
        Personne.__init__(self, nom)
        self.matricule = matricule

    def __str__(self):
        return f"Agent {self.nom}, matricule {self.matricule}"
```

Si cela vous paraît encore un peu vague, expérimentez : c'est toujours le meilleur moyen. Entraînez-vous, contrôlez l'écriture des attributs, ou revenez au premier chapitre de cette partie pour vous rafraîchir la mémoire au sujet du paramètre `self`.

Reprenons notre code précédent qui, cette fois, passe sans problème :

```
>>> agent = AgentSpécial("Fisher", "18327-121")
>>> agent.nom
'Fisher'
>>> print(agent)
Agent Fisher, matricule 18327-121
>>> agent.prénom
'Martin'
>>>
```

Cette fois, notre attribut `prénom` se trouve bien dans notre objet car le constructeur de la classe `AgentSpécial` appelle explicitement celui de `Personne`.

Notez également que, dans le constructeur d'`AgentSpécial`, on n'instancie pas l'attribut `nom`. Celui-ci est en effet écrit par le constructeur de la classe `Personne` que nous appelons en lui passant en paramètre le nom de notre agent.

On pourrait très bien faire hériter une nouvelle classe de `Personne` ; la classe mère est souvent un modèle pour plusieurs classes filles.

Petite précision

Dans le chapitre précédent, je suis passé très rapidement sur l'héritage, ne voulant pas trop m'y attarder et brouiller les cartes inutilement. J'ai toutefois expliqué brièvement que toutes les classes que vous créez héritent de la classe `object`. C'est elle, notamment, qui définit toutes les méthodes spéciales et qui connaît, bien mieux que nous, le mécanisme interne de l'objet. Vous devriez un peu mieux, à présent, comprendre le code du chapitre précédent. Le voici, en substance :

```
    def __setattr__(self, nom_attribut, valeur_attribut):
        """Méthode appelée quand on écrit objet.attribut =
        ↪ valeur"""
        print(f"Attention, on modifie l'attribut
        ↪ {nom_attribut} de l'objet !")
        object.__setattr__(self, nom_attribut,
        ↪ valeur_attribut)
```

En redéfinissant la méthode `__setattr__`, on ne peut, dans le corps de cette méthode, modifier les valeurs de nos attributs comme on le fait habituellement (`self.attribut = valeur`) car alors, la méthode s'appellerait elle-même. On fait donc appel à la méthode `__setattr__` de la classe `object`, cette classe dont héritent implicitement toutes les nôtres. On est sûr que la méthode de cette classe sait écrire une valeur dans un attribut, alors que nous ignorons le mécanisme et que nous n'avons pas besoin de le connaître : c'est la magie du procédé, une fois qu'on a bien compris le principe !

Appel d'une méthode parente avec `super()`

La syntaxe que nous avons vue pour appeler une méthode parente est explicite et aide à comprendre le mécanisme de l'héritage. Cependant, elle est un peu longue à écrire et elle complique l'appel dynamique. Il existe une seconde syntaxe, généralement préférée. Reprenez notre exemple et regardez comment notre constructeur `AgentSpécial` appelle celui de `Personne` :

```
class Personne:
    """Classe représentant une personne"""
    def __init__(self, nom):
        """Constructeur de notre classe"""
        self.nom = nom
        self.prénom = "Martin"

    def __str__(self):
```

```
9           """Méthode appelée lors d'une conversion de l'objet en
            ↪  chaîne"""
10          return f"{self.prénom} {self.nom}"
11
12  class AgentSpécial(Personne):
13      """Classe définissant un agent spécial.
14
15      Elle hérite de la classe Personne.
16
17      """
18
19      def __init__(self, nom, matricule):
20          # On appelle explicitement le constructeur de
            ↪  Personne:
21          super().__init__(nom)
22          self.matricule = matricule
23
24      def __str__(self):
25          return f"Agent {self.nom}, matricule {self.matricule}"
```

Ce code revient strictement au même que le précédent. La différence est l'appel à `super()` :

— `super()` : notez bien que `super` est une fonction ou du moins un `callable` (elle a besoin de ses parenthèses). En vérité, vous pourriez redéfinir la classe fille appelée, ainsi que l'objet instancié, mais c'est rarement utile dans les dernières versions de Python. `super()` peut être appelé avec des paramètres par défaut, il sait quoi faire.
— On appelle ensuite la méthode `__init__` sur le retour de `super()`. On lui passe le nom.

On ne passe pas `self` à notre retour de `super()`. C'est aussi un avantage d'utiliser cette fonction : elle rend l'appel aux méthodes parentes bien plus court et lisible.

Si tout ceci vous semble obscur, souvenez-vous simplement que les deux lignes suivantes sont strictement équivalentes :

— `Personne.__init__(self, nom)`
— `super().__init__(nom)`

Quelle méthode choisir, si les deux font la même chose ?

C'est à vous de voir. J'ai tendance à préférer la seconde ligne avec l'utilisation de `super()`. Je la trouve plus courte et aussi lisible que la première. Il est important

de comprendre les deux, cependant, car vous risquez de les rencontrer dans plusieurs autres codes.

Deux fonctions très pratiques

Python définit deux fonctions se révèlent utiles dans bien des cas : `issubclass` et `isinstance`.

`issubclass`

Comme son nom l'indique, elle vérifie si une classe hérite d'une autre. Elle renvoie `True` si c'est le cas, `False` sinon :

```
>>> issubclass(AgentSpécial, Personne) # AgentSpécial hérite de Personne
True
>>> issubclass(AgentSpécial, object)
True
>>> issubclass(Personne, object)
True
>>> issubclass(Personne, AgentSpécial) # Personne n'hérite pas d'AgentSpécial
False
>>>
```

`isinstance`

`isinstance` indique si un objet est issu d'une classe ou de ses filles :

```
>>> agent = AgentSpécial("Fisher", "18327-121")
>>> isinstance(agent, AgentSpécial) # Agent est une instance d'AgentSpécial
True
>>> isinstance(agent, Personne) # Agent est une instance héritée de Personne
True
>>>
```

Peut-être devrez-vous attendre un peu avant de trouver une utilité à ces deux fonctions mais ce moment viendra.

L'héritage multiple

Python inclut un mécanisme d'**héritage multiple**. L'idée est en substance très simple : au lieu d'hériter d'une seule classe, on peut hériter de plusieurs.

N'est-ce pas ce qui se passe quand on hérite d'une classe qui hérite elle-même d'une autre classe ?

Pas tout à fait. La hiérarchie de l'héritage simple permet d'étendre des méthodes et attributs d'une classe à plusieurs autres, mais la structure reste fermée. Pour mieux comprendre, considérez l'exemple qui suit.

On peut s'asseoir dans un fauteuil. On peut dormir dans un lit. Et il est possible de s'asseoir et de dormir dans certains canapés (la plupart en fait, avec un peu de bonne volonté). Nos classes `Fauteuil` et `Lit` hériteront respectivement de `ObjetPourSAsseoir` et `ObjetPourDormir`. Et notre classe `Canapé` alors ? Elle devra logiquement hériter de `ObjetPourSAsseoir` et `ObjetPourDormir`. C'est un cas où l'héritage multiple pourrait se révéler utile.

Assez souvent, on utilisera l'héritage multiple pour des classes qui ont besoin de certaines fonctionnalités définies dans une classe mère. Par exemple, une classe peut produire des objets destinés à être enregistrés dans des fichiers. On peut faire hériter de cette classe toutes celles qui produiront des objets à enregistrer dans des fichiers. Ces dernières hériteront peut-être aussi d'autres classes incluant, pourquoi pas, d'autres fonctionnalités.

Il existe d'autres utilisations de l'héritage multiple. Bien souvent, cette fonctionnalité ne vous semblera évidente qu'en vous penchant sur la hiérarchie d'héritage de votre programme. Pour l'instant, je vais me contenter de vous en donner la syntaxe et un peu de théorie supplémentaire, en vous encourageant à essayer par vous-mêmes :

```
1 class MaClasseHeritée(MaClasseMère1, MaClasseMère2):
```

Vous pouvez faire hériter votre classe de plus de deux autres. Il suffit d'en préciser entre parenthèses les noms séparés par des virgules.

Recherche des méthodes

La recherche des méthodes se fait dans l'ordre de la définition de la classe. Dans l'exemple précédent, si on appelle une méthode d'un objet issu de `MaClasseHeritée`, c'est dans cette classe qu'on va d'abord la chercher. Si la méthode n'est pas trouvée, on la cherche ensuite dans `MaClasseMère1`, puis successivement dans toutes les classes mères de `MaClasseMère1`, si elle en a. Si on ne la trouve toujours pas, on la recherche dans `MaClasseMère2` et ses classes mères successives.

C'est donc l'ordre de définition des classes mères qui importe.

Retour sur les exceptions

Depuis la première partie, nous ne sommes pas revenus sur les exceptions. Toutefois, ce chapitre me donne une opportunité d'aller un peu plus loin.

Les exceptions sont non seulement des classes, mais des classes hiérarchisées selon une relation d'héritage précise.

Cette relation d'héritage devient importante quand vous utilisez le mot-clé `except`. En effet, le type de l'exception que vous précisez après est intercepté... ainsi que toutes les classes qui héritent de ce type.

Mais comment fait-on pour savoir qu'une exception hérite d'autres exceptions ?

Il y a plusieurs possibilités. Si vous vous intéressez à une exception en particulier, consultez l'aide qui lui est liée.

```
Help on class AttributeError in module builtins:

class AttributeError(Exception)
 |  Attribute not found.
 |
 |  Method resolution order:
 |      AttributeError
 |      Exception
 |      BaseException
 |      object
```

Vous apprenez ici que l'exception `AttributeError` hérite de `Exception`, qui hérite elle-même de `BaseException`.

Vous retrouverez la hiérarchie des exceptions *built-in* sur le site de Python : `https://docs.python.org/fr/3/library/exceptions.html` .

Ne sont répertoriées ici que les exceptions dites *built-in* (natives). D'autres peuvent être définies dans des modules que vous importerez ou être créées par vous-mêmes (nous y reviendrons).

Pour l'instant, souvenez-vous que, quand vous écrivez `except TypeException`, vous cherchez à intercepter toutes les exceptions du type `TypeException`, mais aussi celles des classes qui en héritent.

Les exceptions sont pour la plupart levées pour signaler une erreur... mais pas toutes. L'exception `KeyboardInterrupt` est levée quand vous interrompez votre programme, par exemple avec CTRL + C. Quand on souhaite intercepter toutes les erreurs potentielles, on évite d'écrire un simple `except:` et on le remplace par `except Exception:`, toutes les exceptions « d'erreurs » étant dérivées de `Exception`.

Création d'exceptions personnalisées

Il vous sera utile de créer vos propres exceptions. Puisqu'il s'agit de classes, comme nous venons de le voir, rien ne vous empêche de créer les vôtres. Vous les lèverez avec `raise` et les intercepterez avec `except`.

Se positionner dans la hiérarchie

Vos exceptions doivent hériter d'une exception *built-in* proposée par Python. Commencez par parcourir la hiérarchie de ces dernières pour en trouver une proche. La plupart du temps, vous devrez choisir entre les deux exceptions suivantes :

— `BaseException` : la classe mère de *toutes* les exceptions. La plupart du temps, si vous en faites hériter votre classe, ce sera pour modéliser une exception qui ne sera pas foncièrement une erreur (par exemple, une interruption dans le traitement de votre programme).
— `Exception` : c'est la classe mère de toutes les exceptions « d'erreurs ».

Il est préférable de choisir, dans le contexte, une exception qui se trouve le plus bas possible dans la hiérarchie.

Que doit contenir notre classe `exception` ?

Deux choses sont nécessaires : un constructeur et une méthode `__str__` car, au moment où l'exception est levée, elle doit être affichée. Souvent, votre constructeur ne prend en paramètre que le message d'erreur et la méthode `__str__` le renvoie :

```
class MonException(Exception):
    """Exception levée dans un certain contexte... qui reste à
    ↪ définir"""
    def __init__(self, message):
        """On se contente de stocker le message d'erreur"""
        self.message = message
    def __str__(self):
        """On renvoie le message"""
        return self.message
```

Cette exception s'utilise le plus simplement du monde :

```
>>> raise MonException("OUPS... j'ai tout cassé")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MonException: OUPS... j'ai tout cassé
>>>
```

Vos exceptions peuvent aussi prendre plusieurs paramètres à l'instanciation :

```
class ErreurAnalyseFichier(Exception):
    """Cette exception est levée quand un fichier (de
    ↪ configuration) n'a pas pu être analysé.

    Attributs :
        fichier -- le nom du fichier posant problème
        ligne -- le numéro de la ligne posant problème
```

```
        message -- le problème proprement dit

    """

    def __init__(self, fichier, ligne, message):
        self.fichier = fichier
        self.ligne = ligne
        self.message = message

    def __str__(self):
        """Affichage de l'exception"""
        return f"[{self.fichier}:{self.ligne}]:
         ↪ {self.message}"
```

```
>>> raise ErreurAnalyseFichier("plop.conf", 34,
...         "Il manque une parenthèse à la fin de
 ↪ l'expression")
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
__main__.ErreurAnalyseFichier: [plop.conf:34]: il manque une
 ↪ parenthèse à la fin de l'expression
>>>
```

Voilà, ce petit retour sur les exceptions est achevé. Si vous voulez en savoir plus, n'hésitez pas à consulter la documentation Python les concernant, accessible en français : `https://docs.python.org/fr/3/library/exceptions.html` .

En résumé

- L'héritage permet à une classe de bénéficier du comportement d'une autre en en reprenant les méthodes.
- La syntaxe de l'héritage est `class NouvelleClasse(ClasseMere):`.
- On accède aux méthodes de la classe mère directement via la syntaxe : `ClasseMere.methode(self)`.
- L'héritage multiple permet à une classe d'hériter de plusieurs classes mères.
- La syntaxe de l'héritage multiple s'écrit donc de la manière suivante : `class NouvelleClasse(ClasseMere1, ClasseMere2, ClasseMereN):`.
- Les exceptions définies par Python sont ordonnées selon une hiérarchie d'héritage.

Chapitre 23

Derrière la boucle `for`

Difficulté : 

Au début de cet ouvrage, nous avons étudié les boucles. Ne vous alarmez pas, ce que nous avons vu est toujours d'actualité . . . mais nous allons un peu approfondir le sujet, maintenant que nous explorons le monde de l'objet.

Nous allons ici parler d'**itérateurs** et de **générateurs**. Nous allons découvrir ces concepts du plus simple au plus complexe et de telle sorte que chacun reprenne les précédents. N'hésitez pas, par la suite, à revenir sur ce chapitre et à le relire, partiellement ou intégralement si nécessaire.

Les itérateurs

Nous utilisons des itérateurs sans le savoir depuis le moment où nous avons abordé les boucles et surtout, depuis que nous utilisons le mot-clé `for` pour parcourir des objets conteneurs.

```
1 ma_liste = [1, 2, 3]
2 for élément in ma_liste:
```

Utiliser les itérateurs

C'est sur la seconde ligne que nous allons nous attarder : à force d'utiliser cette syntaxe, vous avez dû vous y habituer et ce type de parcours doit vous être familier. Pourtant, il se cache bel et bien un mécanisme derrière cette instruction.

Quand Python tombe sur une ligne comme `for élément in ma_liste:`, il appelle l'itérateur de `ma_liste`. L'itérateur, c'est un objet chargé de parcourir l'objet conteneur, ici une liste.

L'itérateur est créé dans la méthode spéciale `__iter__` de l'objet. Ici, c'est donc la méthode `__iter__` de la classe `list` qui est appelée et qui renvoie un itérateur permettant de parcourir la liste.

À chaque tour de boucle, Python appelle la méthode spéciale `__next__` de l'itérateur, qui doit renvoyer l'élément suivant du parcours ou lever l'exception `StopIteration` si le parcours touche à sa fin. Ce n'est peut-être pas très clair… alors voyons un exemple.

Avant de plonger dans le code, sachez que Python utilise deux fonctions pour appeler et manipuler les itérateurs : `iter` appelle la méthode spéciale `__iter__` de l'objet passé en paramètre et `next` appelle la méthode spéciale `__next__` de l'itérateur passé en paramètre.

```
1  >>> ma_chaîne = "test"
2  >>> iterateur_de_ma_chaîne = iter(ma_chaîne)
3  >>> iterateur_de_ma_chaîne
4  <str_iterator object at 0x00B408F0>
5  >>> next(iterateur_de_ma_chaîne)
6  't'
7  >>> next(iterateur_de_ma_chaîne)
8  'e'
9  >>> next(iterateur_de_ma_chaîne)
10 's'
11 >>> next(iterateur_de_ma_chaîne)
12 't'
13 >>> next(iterateur_de_ma_chaîne)
14 Traceback (most recent call last):
15   File "<stdin>", line 1, in <module>
16 StopIteration
17 >>>
```

— On commence par créer une chaîne de caractères (jusque là, rien de compliqué).
— On appelle ensuite la fonction `iter` en lui passant la chaîne en paramètre. Cette fonction appelle la méthode spéciale `__iter__` de la chaîne, qui renvoie l'itérateur permettant de parcourir `ma_chaîne`.
— On va ensuite appeler plusieurs fois la fonction `next` en lui passant l'itérateur en paramètre. Cette fonction appelle la méthode spéciale `__next__` de l'itérateur. Elle renvoie successivement chaque lettre contenue dans notre chaîne et lève une exception `StopIteration` quand toutes ont été parcourues.

Quand on parcourt une chaîne grâce à une boucle `for` (`for lettre in chaîne:`), c'est ce mécanisme d'itérateur qui est appelé. Chaque lettre renvoyée par notre itérateur se retrouve dans la variable `lettre` et la boucle s'arrête quand l'exception `StopIteration` est levée.

Vous pouvez reprendre ce code avec d'autres objets conteneurs, des listes par exemple.

Créons nos itérateurs

Pour notre exemple, nous allons créer deux classes :

— `RevStr` : une classe héritant de `str` qui se contentera de redéfinir la méthode `__iter__`. Son mode de parcours sera ainsi altéré : au lieu de parcourir la chaîne de gauche à droite, on la parcourra de droite à gauche (de la dernière lettre à la première).
— `ItRevStr` : notre itérateur. Il sera créé depuis la méthode `__iter__` de `RevStr` et devra parcourir notre chaîne du dernier caractère au premier.

Ce mécanisme est un peu nouveau ; je vous donne le code sans trop de suspense. Si vous vous sentez de faire l'exercice, n'hésitez pas, mais je vous donnerai de toute façon l'occasion de pratiquer dès le prochain chapitre.

```
1  class RevStr(str):
2      """Classe reprenant les méthodes et attributs des chaînes
       ↪  construites depuis 'str'. On se contente de définir
       ↪  une méthode de parcours différente : au lieu de
       ↪  parcourir la chaîne de la première à la dernière
       ↪  lettre, on la parcourt de la dernière à la première.
3
4      Les autres méthodes, y compris le constructeur, n'ont pas
5      besoin d'être redéfinies.
6
7      """
8
9      def __iter__(self):
10         """Cette méthode renvoie un itérateur parcourant la
           ↪  chaîne dans le sens inverse de celui de 'str'.
11
12         """
```

```
        return ItRevStr(self) # On renvoie l'itérateur créé
        ↪  pour l'occasion

class ItRevStr:

    """Un itérateur permettant de parcourir une chaîne de la
    ↪  dernière lettre à la première. On stocke dans des
    ↪  attributs la position courante et la chaîne à
    ↪  parcourir.

    """

    def __init__(self, chaîne_à_parcourir):
        """On se positionne à la fin de la chaîne"""
        self.chaîne_à_parcourir = chaîne_à_parcourir
        self.position = len(chaîne_à_parcourir)

    def __next__(self):
        """Cette méthode doit renvoyer l'élément suivant dans
        ↪  le parcours, ou lever l'exception 'StopIteration'
        ↪  si le parcours est fini.

        """
        if self.position == 0: # Fin du parcours
            raise StopIteration

        self.position -= 1 # On décrémente la position
        return self.chaîne_à_parcourir[self.position]
```

À présent, vous pouvez créer des chaînes à parcourir du dernier caractère vers le premier.

```
>>> ma_chaîne = RevStr("Bonjour")
>>> ma_chaîne
'Bonjour'
>>> for lettre in ma_chaîne:
...     print(lettre)
...
r
u
o
j
n
o
B
>>>
```

Sachez qu'il est aussi possible de mettre en œuvre directement la méthode `__next__` dans notre objet conteneur. Dans ce cas, la méthode `__iter__` pourra renvoyer `self`.

Un exemple est présenté dans la documentation officielle : `https://docs.python.org/fr/3/tutorial/classes.html\#iterators` .

Cela reste quand même plutôt lourd, de devoir définir des itérateurs à chaque fois, non ? Surtout si nos objets conteneurs doivent se parcourir de plusieurs façons, comme les dictionnaires.

Oui, il subsiste quand même beaucoup de répétitions dans le code que nous devons produire, surtout si nous devons créer plusieurs itérateurs pour un même objet. Souvent, on utilisera des itérateurs existants, par exemple celui des listes. Cependant, il existe aussi un autre mécanisme, plus simple et plus intuitif : la raison pour laquelle je ne vous l'explique pas en premier, c'est qu'il passe quand même par des itérateurs, même si c'est implicite, et qu'il n'est pas mauvais de savoir comment cela fonctionne en coulisses.

Les générateurs

Les générateurs sont avant tout un moyen plus pratique de créer et manipuler des itérateurs. Vous verrez un peu plus loin dans ce chapitre qu'ils permettent des choses assez complexes, mais leur puissance tient surtout à leur simplicité et leur petite taille.

Les générateurs simples

Pour créer des générateurs, nous allons découvrir un nouveau mot-clé : `yield`. Ce mot-clé ne peut s'utiliser que dans le corps d'une fonction et il est suivi d'une valeur à renvoyer.

Attends un peu... une valeur ? À renvoyer ?

Oui. Le principe des générateurs étant un peu particulier, il nécessite un mot-clé pour lui tout seul. L'idée consiste à définir une fonction pour un type de parcours. Quand on demande le premier élément du parcours (grâce à `next`), la fonction commence son exécution. Dès qu'elle rencontre une instruction `yield`, elle renvoie la valeur qui suit et se met en pause. Quand on demande l'élément suivant de l'objet (grâce, une nouvelle fois, à `next`), l'exécution reprend à l'endroit où elle s'était arrêtée et s'interrompt au `yield` suivant... et ainsi de suite. À la fin de l'exécution de la fonction, l'exception `StopIteration` est automatiquement levée par Python.

Nous allons prendre un exemple très simple pour commencer :

```
>>> def mon_générateur():
...     """Notre premier générateur. Il va simplement
 ↪  renvoyer 1, 2 et 3"""
...     yield 1
...     yield 2
...     yield 3
...
>>> mon_générateur
<function mon_générateur at 0x00B494F8>
>>> mon_générateur()
<generator object mon_générateur at 0x00B9DC88>
>>> mon_itérateur = iter(mon_générateur())
>>> next(mon_itérateur)
1
>>> next(mon_itérateur)
2
>>> next(mon_itérateur)
3
>>> next(mon_itérateur)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Je pense que cela vous rappelle quelque chose ! Cette fonction, à part l'utilisation de `yield`, est plutôt classique. Quand on l'exécute, on se retrouve avec un générateur. Ce générateur est un objet créé par Python qui définit sa propre méthode spéciale `__iter__` et donc son propre itérateur. Nous aurions tout aussi bien pu écrire ce qui suit :

```
for nombre in mon_générateur(): # Attention on exécute la
 ↪  fonction
    print(nombre)
```

Cela rend quand même le code bien plus simple à comprendre.

Notez qu'on doit exécuter la fonction `mon_générateur` pour obtenir un générateur. Si vous essayez de parcourir notre fonction (`for nombre in mon_générateur`), cela ne marchera pas.

Bien entendu, la plupart du temps, on ne se contentera pas d'appeler `yield` comme cela. Le générateur de notre exemple n'a pas beaucoup d'intérêt, il faut bien le reconnaître.

Essayons de coder une chose un peu plus utile : un générateur prenant en paramètres deux entiers, une borne inférieure et une borne supérieure, et renvoyant chaque entier compris entre ces bornes. Si on écrit par exemple `intervalle(5, 10)`, on parcourra les entiers de 6 à 9. Le résultat attendu est donc le suivant :

```
>>> for nombre in intervalle(5, 10):
...     print(nombre)
...
6
7
8
9
>>>
```

Essayez de faire l'exercice ; c'est un bon entraînement et pas très compliqué de surcroît.

Voici la correction :

```
def intervalle(borne_inf, borne_sup):
    """Générateur parcourant la série des entiers entre
    ↪ borne_inf et borne_sup.

    Note : borne_inf doit être inférieure à borne_sup.

    """
    borne_inf += 1
    while borne_inf < borne_sup:
        yield borne_inf
        borne_inf += 1
```

Là encore, vous pouvez améliorer cette fonction. Pourquoi ne pas faire en sorte que, si la borne inférieure est supérieure à la borne supérieure, le parcours se fasse dans l'autre sens ?

L'important est que vous compreniez bien l'intérêt et le mécanisme sous-jacent. Je vous encourage, là encore, à tester, à disséquer cette fonctionnalité, à essayer de reprendre les exemples d'itérateurs et à les convertir en générateurs.

Si, dans une classe quelconque, la méthode spéciale `__iter__` contient un appel à `yield`, alors ce sera ce générateur qui sera appelé quand on voudra parcourir la boucle. Même quand Python passe par des générateurs, comme vous l'avez vu, il utilise (implicitement) des itérateurs. C'est juste plus confortable pour le codeur, qui n'a pas besoin de créer une classe par itérateur ni de coder une méthode `__next__`, ni même de lever l'exception `StopIteration` : Python fait tout cela pour nous. Pratique non ?

Les générateurs comme coroutines

Jusqu'ici, que ce soit avec les itérateurs ou avec les générateurs, nous créons un moyen de parcourir notre objet au début de la boucle `for`, en sachant que nous ne pourrons pas modifier le comportement du parcours par la suite. Toutefois, les générateurs possèdent un certain nombre de méthodes permettant, justement, d'interagir avec eux pendant le parcours.

Malheureusement, à notre niveau, les idées d'applications *utiles* me manquent et je vais me contenter de vous présenter la syntaxe et un petit exemple. Peut-être trouverez-vous par la suite une application utile des **coroutines** quand vous vous lancerez dans des programmes conséquents, ou que vous aurez été plus loin dans l'apprentissage de Python.

Les **coroutines** sont un moyen d'altérer le parcours… pendant le parcours. Par exemple, dans notre générateur `intervalle`, on pourrait vouloir passer directement de 5 à 10.

Le système des coroutines en Python est contenu dans le mot-clé `yield` et l'utilisation de certaines méthodes de notre générateur.

Interrompre la boucle

La première méthode que nous allons découvrir est `close`. Elle interrompt prématurément la boucle, comme le mot-clé `break` en somme.

```
générateur = intervalle(5, 20)
for nombre in générateur:
    if nombre > 17:
        générateur.close() # Interruption de la boucle
```

Comme vous le voyez, pour appeler les méthodes du générateur, on doit le stocker dans une variable avant la boucle. Si vous aviez écrit directement `for nombre in intervalle(5, 20)`, vous n'auriez pas pu appeler la méthode `close` du générateur.

Envoyer des données à notre générateur

Pour cet exemple, nous allons étendre notre générateur pour qu'il accepte de recevoir des données pendant son exécution.

Le point d'échange de données se fait au mot-clé `yield`. `yield valeur` « renvoie » `valeur` qui deviendra donc la valeur courante du parcours. La fonction se met ensuite en pause. On peut, à cet instant, envoyer une valeur à notre générateur. Cela altère son fonctionnement pendant le parcours.

Reprenons notre exemple en intégrant cette fonctionnalité :

```
def intervalle(borne_inf, borne_sup):
    """Générateur parcourant la série des entiers entre
    ↪  borne_inf et borne_sup.

    Notre générateur doit pouvoir "sauter" une certaine plage
    de nombres en fonction d'une valeur qu'on lui donne
    pendant le parcours. La valeur qu'on lui passe est la
    nouvelle valeur de borne_inf.

    Note : borne_inf doit être inférieure à borne_sup.

    """
    borne_inf += 1
```

```
    while borne_inf < borne_sup:
        valeur_reçue = (yield borne_inf)
        if valeur_reçue is not None: # Notre générateur a reçu
        ↪  quelque chose
            borne_inf = valeur_reçue
        borne_inf += 1
```

Nous configurons notre générateur pour qu'il accepte une valeur éventuelle au cours du parcours. S'il reçoit une valeur, il l'attribue au point du parcours.

Autrement dit, au cours de la boucle, vous pouvez demander au générateur de sauter tout de suite à 20 si le nombre est 15.

Tout se passe à partir de la ligne du `yield`. Au lieu de simplement renvoyer une valeur à notre boucle, on capture une éventuelle valeur dans `valeur_reçue`. La syntaxe est simple : `variable = (yield valeur_à_renvoyer)` [1].

Si aucune valeur n'a été passée à notre générateur, notre `valeur_reçue` vaudra `None`. On vérifie donc qu'elle ne vaut pas `None` et, dans ce cas, on affecte la nouvelle valeur à `borne_inf`.

Voici le code pour interagir avec notre générateur. On utilise la méthode `send` pour lui envoyer une valeur :

```
générateur = intervalle(10, 25)
for nombre in générateur:
    if nombre == 15: # On saute à 20
        générateur.send(20)
    print(nombre, end=" ")
```

Il existe d'autres méthodes pour interagir avec notre générateur. Vous les retrouverez, ainsi que des explications supplémentaires, sur la documentation officielle traitant du mot-clé `yield`.

En résumé

- Quand on utilise la boucle `for élément in séquence:`, un itérateur de cette séquence permet de la parcourir.
- On récupère l'itérateur d'une séquence grâce à la fonction `iter`.
- Une séquence renvoie l'itérateur permettant de la parcourir grâce à la méthode spéciale `__iter__`.
- Un itérateur possède une méthode spéciale, `__next__`, qui renvoie le prochain élément à parcourir ou lève l'exception `StopIteration` qui arrête la boucle.
- Les générateurs facilitent la création des itérateurs.
- Ce sont des fonctions utilisant le mot-clé `yield` suivi de la valeur à transmettre à la boucle.

1. N'oubliez pas les parenthèses autour de `yield valeur`.

Chapitre 24

TP : un jeu de cartes

Difficulté : 

Voici venue l'heure de vérité, l'heure de pratique ! Il est temps d'appliquer ce que nous avons découvert tout au long de cette partie, en particulier les classes, les propriétés et les méthodes spéciales. Faisons un pas de plus vers les spécifications abstraites ; je vous propose un TP un peu différent des deux autres dans la forme et le code à produire.

Notre mission

Notre but dans ce TP est de créer les classes pour un jeu de cartes. Le jeu en lui-même n'a pas d'importance : il va nous falloir créer les structures qui représentent respectivement une carte et un ensemble de cartes. Ces deux structures nous seront utiles dans le développement de beaucoup de jeux différents.

Contrairement à l'habitude, je ne vais pas vous dresser une liste complète de ce que vous devrez coder, mais plutôt vous donner un retour de l'interpréteur pour démontrer l'utilisation des classes à créer. À vous de remplir vos classes pour que la session de l'interpréteur ci-dessous fonctionne comme attendue.

Session d'utilisation

```
>>> from carte import Carte, Jeu52Cartes
>>> jeu = Jeu52Cartes()
>>> str(jeu)
'2 de pique, 3 de pique, 4 de pique, 5 de pique, 6 de pique,
 ↪  7 de pique, 8 de pique, 9 de pique, 10 de pique, valet de
 ↪  pique, dame de pique, roi de pique, as de pique, 2 de
 ↪  cœur, 3 de cœur, 4 de cœur, 5 de cœur, 6 de cœur, 7 de
 ↪  cœur, 8 de cœur, 9 de cœur, 10 de cœur, valet de cœur,
 ↪  dame de cœur, roi de cœur, as de cœur, 2 de carreau, 3 de
 ↪  carreau, 4 de carreau, 5 de carreau, 6 de carreau, 7 de
 ↪  carreau, 8 de carreau, 9 de carreau, 10 de carreau, valet
 ↪  de carreau, dame de carreau, roi de carreau, as de
 ↪  carreau, 2 de trèfle, 3 de trèfle, 4 de trèfle, 5 de
 ↪  trèfle, 6 de trèfle, 7 de trèfle, 8 de trèfle, 9 de
 ↪  trèfle, 10 de trèfle, valet de trèfle, dame de trèfle,
 ↪  roi de trèfle, as de trèfle'
>>> jeu.mélanger()
>>> str(jeu)
'6 de cœur, 7 de cœur, 5 de cœur, 10 de pique, 7 de pique, 5
 ↪  de pique, 2 de trèfle, 4 de pique, valet de trèfle, valet
 ↪  de carreau, 10 de trèfle, 2 de cœur, dame de cœur, 3 de
 ↪  pique, dame de trèfle, 6 de carreau, 6 de pique, 3 de
 ↪  cœur, roi de trèfle, 4 de carreau, valet de cœur, as de
 ↪  cœur, 9 de trèfle, as de trèfle, 2 de pique, 8 de trèfle,
 ↪  3 de carreau, 3 de trèfle, dame de pique, 9 de carreau,
 ↪  as de pique, 10 de cœur, 5 de carreau, roi de pique, 4 de
 ↪  trèfle, roi de carreau, 8 de cœur, 7 de carreau, dame de
 ↪  carreau, 4 de cœur, valet de pique, 8 de carreau, 2 de
 ↪  carreau, roi de cœur, 10 de carreau, 7 de trèfle, 5 de
 ↪  trèfle, 9 de pique, as de carreau, 6 de trèfle, 9 de
 ↪  cœur, 8 de pique'
>>> carte = jeu.piocher()
```

```
>>> carte
<Carte(valeur='6', enseigne='cœur')
>>> print(carte)
6 de cœur
>>> carte in jeu
False
>>> len(jeu)
51
>>> main = [jeu.piocher() for i in range(7)]
>>> main
[<Carte(valeur='7', enseigne='cœur'), <Carte(valeur='5',
 ↪  enseigne='cœur'),
<Carte(valeur='10', enseigne='pique'), <Carte(valeur='7',
 ↪  enseigne='pique'),
<Carte(valeur='5', enseigne='pique'), <Carte(valeur='2',
 ↪  enseigne='trèfle'),
<Carte(valeur='4', enseigne='pique')]
>>> carte = Carte("valet", "pique")
>>> carte
<Carte(valeur='valet', enseigne='pique')
>>> carte.couleur
'noir'
>>> carte = Carte("autre", "pique")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "D:\livre\python\part3\chap7\carte.py", line 33, in
   ↪  __init__
    raise ValueError(f"valeur inconnue : {valeur}")
ValueError: valeur inconnue : autre
>>> carte = Carte("8", "bleu")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "D:\livre\python\part3\chap7\carte.py", line 36, in
   ↪  __init__
    raise ValueError(f"enseigne inconnue : {enseigne}")
ValueError: enseigne inconnue : bleu
>>>
```

Explications

On dit souvent qu'un peu de code de démonstration vaut bien mieux que plusieurs paragraphes d'explications. C'est le moment de voir si cette phrase est vraie ou pas. Cet exercice est relativement simple en développement, mais il vous demande de réfléchir en partant de la fin (comment utiliser vos objets). C'est assez courant et c'est un bon entraînement pour la suite. Voici quelques explications malgré tout :

— Notez que nous avons deux classes. Je les ai regroupées dans un seul module (un seul fichier), mais c'est à vous de voir au final. La première s'appelle `Carte` et définit une simple carte, avec sa valeur (2, 3, ..., 10, valet, dame, roi, as) et son enseigne (pique, cœur, carreau, trèfle). J'utiliserai ces noms pour éviter la confusion avec le terme de couleur. La seconde définit un jeu de 52 cartes.
— Dans la première session, nous commençons par créer un jeu de 52 cartes. C'est un objet qui contient 13 cartes de chaque enseigne. Si vous n'êtes pas familier avec ce type de jeu, la page Wikipédia consacrée au jeu de cartes français donne bien plus d'informations : `https://fr.wikipedia.org/wiki/Jeu_de_cartes_franÃğais`
— L'objet contenant notre jeu de cartes possède plusieurs méthodes spéciales que je vous laisse découvrir et créer. Il a aussi une méthode pour mélanger le jeu et une autre pour piocher une carte (une carte aléatoire, si le jeu a été mélangé). Cette seconde méthode retire la carte du jeu. Si vous piochez de nouveau, vous ne retomberez pas sur cette carte.
— On utilise les compréhensions de liste pour démontrer que l'on peut piocher plusieurs cartes d'un coup.
— On essaye de créer un objet `Carte` directement, avec de bonnes ou fausses valeurs, puis de bonnes ou fausses enseignes.

Vous devriez avoir assez d'information pour savoir quoi faire : regardez attentivement la session de l'interpréteur donnée dans la section précédente. Listez les classes à créer, les méthodes (et peut-être les propriétés) de chacune. Quand vous pensez avoir une solution valide, essayez la session de l'interpréteur sur vos objets. Voyez s'ils réagissent bien de la façon attendue. Si ce n'est pas le cas, peut-être avez-vous oublié quelque chose ? Quand tout vous semble en ordre, vous pouvez vous féliciter : vous avez résolu, peut-être, votre premier problème en partant des interactions utilisateurs (ou développeurs), pas du code à créer. C'est une étape importante !

Correction

Comme toujours, je vous encourage à essayer par vous-mêmes avant de vous précipiter sur la correction. Gardez également à l'esprit que la correction que je propose n'est que l'une des nombreuses solutions possibles.

Voici mon fichier `carte.py` contenant les deux classes.

```
"""Module contenant les classes Carte et Jeu52Cartes."""

from random import shuffle

VALEURS = "2 3 4 5 6 7 8 9 10 valet dame roi as".split()
ENSEIGNES = "pique cœur carreau trèfle".split()
COULEURS = {
        "pique": "noir",
        "cœur": "rouge",
        "carreau": "rouge",
```

```
        "trèfle": "noir",
}

class Carte:

    """Classe représentant une carte (à jouer), avec une
    ↪ valeur et enseigne.

    La valeur d'une carte représente sa "force" au sein
    d'une enseigne (2, 3, 8, valet, dame, roi, as...).
    Son enseigne est la suite à laquelle elle appartient
    (pique, cœur, carreau, trèfle). Sa couleur (rouge
    ou noir) dépend de son enseigne.

    Note : pour éviter la confusion dans notre code, le nom
    "couleur" définit strictement une carte rouge ou noire.
    Le nom "enseigne" désigne la suite à laquelle la carte
    appartient (aussi appelée couleur dans de nombreux jeux).

    Pour construire une carte, donnez en argument sa valeur
    et son enseigne (toutes deux sous la forme de str).
    Par exemple :

    >>> carte = Carte("3", "carreau")

    (Si la valeur ou l'enseigne n'existe pas, une exception
    ValueError est levée).

    """

    def __init__(self, valeur: str, enseigne: str):
        if valeur not in VALEURS:
            raise ValueError(f"valeur inconnue : {valeur}")

        if enseigne not in ENSEIGNES:
            raise ValueError(f"enseigne inconnue :
            ↪ {enseigne}")

        self.valeur = valeur
        self.enseigne = enseigne

    def __repr__(self) -> str:
        return f"<Carte(valeur='{self.valeur}',
        ↪ enseigne='{self.enseigne}')"

    def __str__(self) -> str:
```

```
54          return f"{self.valeur} de {self.enseigne}"
55
56      @property
57      def couleur(self) -> str:
58          """Retourne la couleur ('noir' ou 'rouge')."""
59          return COULEURS[self.enseigne]
60
61
62  class Jeu52Cartes:
63
64      """Classe représentant un jeu de cartes de 52 cartes.
65
66      Ce jeu possède 13 cartes dans ses 4 enseignes. On peut
    ↪ mélanger ce jeu (méthode mélanger), retirer une carte du
    ↪ sommet (piocher) ainsi que voir tout le jeu ou isoler des
    ↪ cartes par leur indice (avec la notation jeu[indice]).
67
68      """
69
70      def __init__(self):
71          self.cartes = []
72
73          for enseigne in ENSEIGNES:
74              for valeur in VALEURS:
75                  carte = Carte(valeur, enseigne)
76                  self.cartes.append(carte)
77
78      def __str__(self) -> str:
79          return ", ".join([str(carte) for carte in
            ↪  self.cartes])
80
81      def __contains__(self, carte: Carte):
82          return carte in self.cartes
83
84      def __getitem__(self, indice: int) -> Carte:
85          return self.cartes[indice]
86
87      def __len__(self) -> int:
88          return len(self.cartes)
89
90      def mélanger(self):
91          """Mélange le jeu de cartes."""
92          shuffle(self.cartes)
93
94      def piocher(self):
95          """Pioche la première carte du jeu.
```

```
96
97          La carte est retirée du jeu, on ne peut donc plus
98          la piocher ensuite.
99
100         """
101         if len(self.cartes) == 0:
102             raise ValueError(f"Le jeu ne contient plus de
                ↪  cartes.")
103
104         return self.cartes.pop(0)
```

Pour conclure

L'intérêt d'un tel exercice est surtout d'apprendre à créer des classes en fonction de leur utilisation. On planifie leur fonctionnement avant de se précipiter sur le code. Ainsi, on approche le problème de façon plus logique. Vous constatez que le code que je vous propose est assez léger dans tous les cas : avoir une idée claire du problème évite aussi de se perdre trop facilement dans les classes et méthodes du programme.

Maintenant, évidemment, ceci n'est qu'une petite démonstration. Nos classes pourraient être utilisées pour construire un grand nombre de jeux de cartes. Elles pourraient également être améliorées et voici quelques pistes :

- Un itérateur ou générateur serait une bonne solution pour parcourir directement notre jeu de cartes.
- Si on crée une carte directement (avec le constructeur de `Carte`), il faudrait vérifier si elle est présente dans le jeu de cartes ne donne pas des résultats cohérents.
- Il serait intéressant, pour certains jeux, d'avoir un objet explicite destiné à contenir la main (la liste des cartes) d'un joueur, au lieu d'utiliser de simples listes.
- Remettre une carte dans le jeu est indispensable dans certains cas.
- Trier le jeu de cartes serait utile, pour retrouver l'ordre d'origine.

Et ce ne sont que quelques suggestions. Vous pourriez en trouver bien plus au final. Vous avez les compétences pour coder un module très détaillé. Ne négligez pas la pratique, il n'y a que ça de vrai !

Chapitre 25

Les décorateurs

Difficulté : 

Nous allons ici nous intéresser à un aspect fascinant de Python, un concept de programmation assez avancé. Vous n'êtes pas obligés de lire ce chapitre pour la suite de ce livre, ni même de connaître cette fonctionnalité pour coder en Python. Il s'agit d'un plus que j'ai voulu détailler mais qui n'est certainement pas indispensable.

Les décorateurs sont un moyen simple de modifier le comportement « par défaut » de fonctions. C'est un exemple assez flagrant de ce qu'on appelle la **métaprogrammation**, que je vais décrire assez brièvement comme l'écriture de programmes manipulant... d'autres programmes. Cela donne faim, non ?

Qu'est-ce que c'est ?

Les décorateurs sont des fonctions de Python dont le rôle est de modifier le comportement par défaut d'autres fonctions ou classes. Pour schématiser, une fonction modifiée par un décorateur ne s'exécutera pas elle-même mais appellera le décorateur. C'est au décorateur de décider s'il veut exécuter la fonction et dans quelles conditions.

Quel est l'intérêt ? Si on veut juste qu'une fonction fasse quelque chose de différent, il suffit de la modifier, non ? Pourquoi s'encombrer la tête avec une nouvelle fonctionnalité plus complexe ?

Il y a de nombreux cas dans lesquels les décorateurs sont un choix intéressant. Pour comprendre l'idée, je vais prendre un unique exemple.

On souhaite tester les performances de certaines de nos fonctions : en l'occurrence, calculer combien de temps leur est nécessaire pour s'exécuter.

Une possibilité, effectivement, consiste à modifier chacune des fonctions devant intégrer ce test. Toutefois, ce n'est pas très élégant, ni très pratique, ni très sûr... bref ce n'est pas la meilleure solution.

Une autre possibilité consiste à utiliser un décorateur. Il se chargera d'exécuter notre fonction en calculant le temps qu'il lui faut et, par exemple, affichera une alerte si cette durée est trop élevée.

Pour indiquer qu'une fonction doit intégrer ce test, il suffira d'ajouter une simple ligne avant sa définition. C'est bien plus simple, clair et adapté à la situation.

Et ce n'est qu'un exemple d'application.

Les décorateurs sont des fonctions standard de Python mais leur construction est parfois complexe. Quand il s'agit de décorateurs prenant des arguments en paramètres ou devant tenir compte des paramètres de la fonction, le code est plus complexe, moins intuitif.

Je vais faire mon possible pour que vous compreniez bien le principe. N'hésitez pas à y revenir à tête reposée, une, deux, trois fois pour que cela soit bien clair.

Syntaxe et exemples

La syntaxe des décorateurs est assez surprenante. Dans un sens, vous n'allez presque rien découvrir de nouveau (pas de mot-clé, presque pas de syntaxe spécifique), mais les décorateurs sont un chapitre à eux tout seuls... et pourraient occuper davantage de place.

Présentation

Comme je l'ai dit, les décorateurs sont des fonctions « classiques » de Python, dans leur définition. Ils ont une petite subtilité en ce qu'ils prennent en paramètre une fonction et renvoient une fonction.

On déclare qu'une fonction doit être modifiée par un (ou plusieurs) décorateur(s) grâce à une (ou plusieurs) ligne(s) au-dessus de sa définition :

```
@nom_du_décorateur
def ma_fonction(...)
```

Voyons un exemple concret avant de le décortiquer :

```
class DireBonjour:

    """Décorateur... pour dire bonjour."""

    def __init__(self, fonction):
        print(f"Le décorateur se construit. Il travaille sur
         ↪ {fonction}.")
        self.fonction = fonction

    def __call__(self):
        print("Je dis bonjour, depuis le décorateur.")
        return self.fonction()
```

Et pour utiliser ce décorateur :

```
>>> @DireBonjour # On "décore" la fonction ci-dessous
... def salut():
...     print("... ou salut !")
...
Le décorateur se construit. Il travaille sur <function salut
 ↪  at 0x01BA8660>.
>>> # Appelons notre fonction
... salut()
Je dis bonjour, depuis le décorateur.
... ou salut !
>>>
```

Attends ! On devait travailler sur des fonctions et on a créé une classe...

Il existe plusieurs façons de créer un décorateur. Ce que nous venons de faire, créer une classe, est (à mon sens) le plus simple à lire pour débuter. Dans notre classe, vous remarquez deux méthodes : le constructeur (`__init__` que vous devriez reconnaître) et une seconde méthode spéciale appelée `__call__`. Cette seconde méthode est appelée quand on exécute l'objet comme une fonction (avec des parenthèses).

Avant d'entrer trop dans le détail, observons le résultat : on crée une classe qui sera notre décorateur. Il y a deux `print` pour indiquer quand les deux méthodes de la classe sont appelées. Puis, vous voyez la ligne suivante :

```
1 @DireBonjour
```

Elle indique à Python que la fonction définie au-dessous (notre fonction `salut` en l'occurrence) est décorée avec `DireBonjour`.

Décorée ?

Un décorateur classique altère le comportement d'une fonction qu'il décore. Nous allons voir des exemples d'utilisation plus utiles par la suite, mais analysons déjà ce premier résultat :

— Juste après la définition de la fonction, le décorateur (un objet de la classe `DireBonjour`) est instancié. On lui passe notre fonction (`salut` ici).
— Quand on appelle notre fonction, c'est la méthode `__call__` de la classe `DireBonjour` qui est appelée. Après avoir affiché un message, elle appelle notre fonction enregistrée (`salut`) et retourne son résultat.

Est-ce plus clair ? Peut-être pas. Voici, en résumé, deux codes qui font exactement la même chose :

```
1 @DireBonjour
2 def salut():
3     ...
```

... est strictement équivalent à :

```
1 def salut():
2     ...
3 salut = DireBonjour(salut)
```

Ce second code devrait, je l'espère, vous montrer ce qui se passe : notre décorateur est créé et prend la place de notre fonction `salut`. Cela veut dire que, quand on appelle `salut()`, on appelle en vérité la méthode `__call__` d'un objet de la classe `DireBonjour`. Cet objet possède un attribut vers notre fonction décorée et peut donc l'appeler.

En résumé... on remplace une fonction par un objet contenant notre fonction... mais à quoi ça sert ?

Nous verrons de nombreux exemples plus utiles en pratique. Si vous ne retenez qu'une chose de cette section, c'est le code suivant :

```
1 @DireBonjour
2 def salut():
3     pass
```

```
4
5  # ... est équivalent à ... :
6  def salut():
7      pass
8  salut = DireBonjour(salut)
9
10 # ... et ...
11 salut()
12
13 # ... est équivalent à ... :
14 salut.__call__()
```

Si vous êtes perdu, relisez ce code, faites des tests, relisez ce code, faites d'autres tests, relisez... enfin, vous m'avez compris.

Un premier exemple : mesurer la durée de nos fonctions

Essayons de coder un décorateur dont le seul rôle sera de mesurer le temps nécessaire à l'exécution de nos fonctions.

Pour mesurer le temps, nous allons utiliser la fonction `time()` du module `time`. Elle retourne un nombre de secondes. En l'appelant avant et après l'exécution de notre fonction, on pourra mesurer la différence entre les deux temps et connaître la durée d'exécution de notre fonction.

Sans suspense, voici le code que je vais expliquer plus bas :

```
1  from time import time
2
3  class MesurerTemps:
4
5      """Mesure le temps d'exécution d'une fonction."""
6
7      def __init__(self, fonction):
8          self.fonction = fonction
9
10     def __call__(self):
11         avant = time()
12         résultat = self.fonction()
13         après = time()
14         print(f"La fonction {self.fonction} a mis {après -
           ↪  avant} seconde(s) pour s'exécuter.")
15         return résultat
```

Hormis l'appel à `time`, il n'y a rien de très nouveau. Notre décorateur est assez facile à lire : quand on l'appelle (dans la méthode `__call__`, il cherche le temps actuel, puis appelle notre fonction, regarde le temps après exécution, affiche la différence et retourne le résultat de notre fonction.

Comment utiliser ce décorateur ? Très simplement :

```
# On va créer une fonction qui prend un temps aléatoire pour s'exécuter
# time.sleep() sert à "mettre en pause" le programme pendant un certain temps
# On utilise random.randint pour attendre un nombre variable de secondes
from random import randint
from time import sleep

@MesurerTemps
def faire_une_pause():
    """Fait une pause, pendant... quelques secondes."""
    secondes = randint(1, 3)
    print(f"J'aimerais prendre ma pause... peut-être {secondes} seconde(s) ?")
    sleep(secondes)
    secondes = randint(1, 7)
    print(f"À la réflexion, ce n'était pas assez long. Prolongeons de {secondes} seconde(s)...")
    sleep(secondes)
    print("Allez, fin de la pause.")
```

Là encore, notre fonction ne devrait pas trop vous surprendre dans le contenu. Elle met volontairement un peu de temps à s'exécuter (et pour faire joli, nous ajoutons un peu d'aléatoire pour ne jamais vraiment savoir combien de temps la fonction va durer). Notez bien la ligne `@MesurerTemps` au-dessus de notre définition de fonction. C'est elle qui va créer notre décorateur et remplacer `faire_une_pause`.

Essayons d'appeler notre fonction :

```
>>> faire_une_pause()
J'aimerais prendre ma pause... peut-être 3 seconde(s) ?
À la réflexion, ce n'était pas assez long. Prolongeons de 5 seconde(s)...
Allez, fin de la pause.
La fonction <function faire_une_pause at 0x03BB8618> a mis 8.00367546081543 seconde(s) pour s'exécuter.
>>>
```

Si vous exécutez ce code, vous constaterez qu'il lui faut quelques secondes avant d'afficher la durée, grâce à `MesurerTemps`.

Si l'affichage, ou le code, vous semble difficile à comprendre, prenez le temps de l'analyser, ajoutez des lignes `print` si cela peut vous aider, modifiez le code pour écrire des exemples que vous comprenez mieux si besoin.

Des paramètres dans nos fonctions décorées

Vous l'avez sans doute remarqué, notre fonction décorée fonctionne car elle ne prend aucun paramètre. Si elle en contenait, nous obtiendrions une erreur :

```
>>> @MesurerTemps
... def somme(a, b):
...     return a + b
...
>>> somme(5, 8)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __call__() takes 1 positional argument but 3 were
 ↪  given
>>>
```

On dira ce qu'on voudra, mais les exceptions Python sont assez utiles pour remonter à la source de l'erreur. Arrivez-vous à comprendre celle-ci ?

Python se plaint que notre méthode `__call__` ne prend aucun argument (excepté `self`). D'où vient cette méthode `__call__` ? De notre classe `MesurerTemps` bien entendu. Notre méthode ne prend aucun paramètre, mais en appelant `somme(3, 4)`, on a appelé `MesurerTemps().__call__(3, 4)`, d'où l'erreur.

Nous pourrions ajouter `a` et `b` dans notre méthode `__call__`. Cependant, nous ne pourrions alors pas mesurer `faire_une_pause` qui ne prend aucun paramètre. Comment écrire des fonctions prenant un nombre variable d'arguments ?

Vous vous souvenez peut-être de cette fonctionnalité. Nous l'avons présentée aux chapitres sur les listes et les dictionnaires. La syntaxe avec des étoiles devant les noms des arguments vous rappelle-t-elle quelque chose ?

Notre méthode `__call__` doit prendre un nombre variable d'arguments, parfois nommés, d'autres fois non. Modifions un tout petit peu notre décorateur :

```
class MesurerTemps:

    """Mesure le temps d'exécution d'une fonction."""

    def __init__(self, fonction):
        self.fonction = fonction

    def __call__(self, *args, **kwargs):
        avant = time()
        print(f"On appelle {self.fonction} avec args={args},
         ↪  kwargs={kwargs}")
        résultat = self.fonction(*args, **kwargs)
        après = time()
        print(f"La fonction {self.fonction} a mis {après -
         ↪  avant} seconde(s) pour s'exécuter.")
        return résultat
```

Cette fois-ci, notre méthode `__call__` prend un nombre variable d'arguments et appelle notre fonction décorée. Voyons donc le résultat :

```
>>> somme(5, 8)
On appelle <function somme at 0x039C0810> avec args=(5, 8),
 ↪  kwargs={}
La fonction <function somme at 0x039C0810> a mis
 ↪  0.0020101070404052734 seconde(s) pour s'exécuter.
13
>>> faire_une_pause()
On appelle <function faire_une_pause at 0x03B18618> avec
 ↪  args=(), kwargs={}
J'aimerais prendre ma pause... peut-être 1 seconde(s) ?
À la réflexion, ce n'était pas assez long. Prolongeons de 1
 ↪  seconde(s)...
Allez, fin de la pause.
La fonction <function faire_une_pause at 0x03B18618> a mis
 ↪  2.0068068504333496 seconde(s) pour s'exécuter.
>>>
```

Nous pouvons maintenant décorer n'importe quelle fonction avec `MesurerTemps`. Cette syntaxe est extrêmement utile dans ce contexte.

Des décorateurs avec des paramètres

Parfois, les décorateurs prennent des paramètres (entre parenthèses, après leur nom comme n'importe quelle fonction). Cette syntaxe est utile pour modifier le comportement du décorateur. Elle allourdit quelque peu notre classe, ne vous endormez pas !

Pourquoi passer des arguments à nos décorateurs ?

Il arrive parfois que nous définissions un décorateur assez générique, utilisable dans différents contextes, mais que nous désirions au cas par cas lui préciser des paramètres pour modifier un peu son comportement. Nous allons prendre un exemple qui est à la limite du licite en Python : contrôler les types des paramètres transmis à nos fonctions.

Voici comment nous voudrions appeler notre décorateur :

```
@VérifierTypes(int, float)
def ajouter(entier, flottant):
    """On doit préciser deux arguments ici : un entier et un
     ↪  flottant."""
```

Notre décorateur devra s'assurer que l'utilisateur de la fonction précise bien, en premier argument, un objet de type `int` et en second argument, un objet de type `float`.

Pourquoi illicite ? Ce serait assez pratique de vérifier le type de nos arguments et générerait probablement moins d'erreurs.

Ce type de fonctionnalité va à l'encontre de la philosophie de Python. On préfère ajouter de la documentation (ou des annotations) pour indiquer les types, sans forcer l'utilisateur. Notre décorateur est très bien pour notre exemple, mais il va ralentir l'appel de chacune de nos fonctions (pas dans des proportions énormes, mais un petit peu) pour faire une vérification qui n'est pas encouragée. Ceci étant dit, cela reste un bon exemple :

```
class VérifierTypes:

    """Décorateur appelé pour vérifier les types d'appel à une
    fonction donnée.

    Pour l'utiliser, on précise les types dans le même ordre
    que dans la définition de la fonction. Par exemple :

        @VérifierTypes(str, int)
        def répéter(chaîne, nombre):
            ...

    """

    def __init__(self, *args):
        # Cette fois-ci, on ne transmet pas la fonction au
        ↪ constructeur
        self.fonction = None
        self.types = args

    def __call__(self, fonction):
        # La fonction est transmise à __call__ directement
        self.fonction = fonction
        return self.fonction_de_remplacement

    def fonction_de_remplacement(self, *args, **kwargs):
        """Cette méthode prend la place de notre fonction
        ↪ décorée."""
        # On parcourt les types définis dans self.types
        i = 0
        while i < len(args):
            arg = args[i]
            types = self.types[i]

            # Vérifie que arg est de type types
            if not isinstance(arg, types):
```

```
                print(f"Attention ! Argument {i} devrait être
                ↪ {types} mais vaut {arg}")

            i += 1

        # Quoiqu'il arrive, on appelle la fonction
        return self.fonction(*args, **kwargs)
```

Il y a de quoi paniquer !

— Notez que cette fois-ci, notre constructeur de `VérifierTypes` ne reçoit pas la fonction décorée, mais les arguments transmis au constructeur (c'est-à-dire, les types attendus).
— On envoie la fonction à `__call__`, ce qui n'était pas le cas avant.
— Notre méthode `__call__` retourne... une autre méthode ! Une méthode qui va prendre la place de notre fonction décorée.
— Le code de `fonction_de_remplacement` devrait être lisible, en théorie.

En somme, nous ajoutons une nouvelle méthode et on demande à `__call__` de la renvoyer.

`__call__` ne renvoie pas le résultat de `fonction_de_remplacement`, mais une référence sur cette méthode. Notez bien l'absence de parenthèses.

Je suis toujours perdu... je ne sais plus qui appelle quoi.

Revenons donc à notre code de base. Souvenez-vous que :

```
@VérifierTypes(str, int)
def répéter(chaîne, nombre):
    pass

# ... fait la même chose que :
def répéter(chaîne, nombre):
    pass
répéter = VérifierTypes(str, int)(répéter)
```

Regardez bien cette dernière ligne. L'objet de type `VérifierTypes` est instancié (on lui transmet les types à surveiller). Puis on l'appelle comme une fonction en lui envoyant la fonction à décorer. N'oubliez pas, `fonction()` est un raccourci vers `fonction.__call__`.

Quand nous appelons `répéter`... c'est donc le résultat de `VérifierTypes.__call__` qui est appelé. Or, cette méthode retourne une référence vers une autre méthode.

En bref ?

Quand on appelle `répéter`, c'est `VérifierTypes.fonction_de_remplacement` qui est appelée !

Utiliser ce décorateur est encore très simple :

```
>>> @VérifierTypes(str, int)
... def répéter(chaîne, nombre):
...     """Répète chaîne nombre fois."""
...     return nombre * chaîne
...
>>> répéter('peut-être bien ', 3)
'peut-être bien peut-être bien peut-être bien '
>>> répéter(15, 3)
Attention ! Argument 0 devrait être <class 'str'> mais vaut
 ↪  15
45
>>>
```

Écrire nos décorateur nous prend du temps, mais les utiliser est vraiment facile en comparaison. Ce dernier exemple est un peu plus difficile à comprendre, n'hésitez pas à passer un peu de temps dessus, ou à passer à la suite et y revenir plus tard.

La syntaxe sans classe

Beaucoup de cours Python présentent les décorateurs avec une autre syntaxe qui ne nécessite aucune classe. Elle semble plus légère, mais elle est assez délicate à comprendre (de mon point de vue). Je ne vais pas m'étendre sur cette deuxième syntaxe, mais si vous tombez sur des fonctions contenant d'autres fonctions contenant d'autres fonctions... vous penserez aux décorateurs tels que certains les définissent.

Utiliser des classes comme nous l'avons fait (en particulier le dernier exemple) est parfois contestable. L'avantage est de bien différencier les rôles de chaque méthode en maintenant notre code relativement facile à lire.

Peur de rien, pas même des décorateurs ? Je peux vous recommander cet excellent article en anglais, non pas sur le site officiel de Python, mais sur celui de Real Python : `https://realpython.com/primer-on-python-decorators/` .

Quelques décorateurs courants

Nous avons expliqué la syntaxe `@...`, au moins en tant qu'utilisateur. La vérité est que Python a défini certains décorateurs pour nous. Et il y en a beaucoup que nous n'aurons pas le temps de voir.

Les propriétés

Souvenez-vous de la définition des propriétés : nous avons utilisé deux fois la syntaxe des décorateurs (`@property` et `@propriété.setter`) pour en définir. Vous comprenez maintenant que cette syntaxe crée des propriétés en utilisant deux décorateurs.

Commençons par voir la syntaxe des propriétés en lecture seule. Vous en souvenez-vous ?

```
class MaClasse:

    @property
    def ma_propriété(self):
        ...
```

Si vous avez retenu les explications concernant les décorateurs, vous devriez être capable d'écrire cette propriété en lecture seule d'une autre façon, sans utiliser l'opérateur `@` :

```
class MaClasse:

    def ma_propriété(self):
        ...
    ma_propriété = property(ma_propriété)
```

Maintenant déconseillée, cela a été la syntaxe officielle des propriétés pour quelque temps. Même si les deux syntaxes font exactement la même chose, il est clair que celle utilisant `@property` est préférée.

Sur le même principe, la syntaxe pour créer une propriété en lecture et écriture devrait maintenant vous sembler plus claire :

```
class MaClasse:
    @property
    def ma_propriété(self):
        ...

    @ma_propriété.setter
    def ma_propriété(self, nouvelle_valeur):
        ...
```

Ce code revient donc à écrire :

```
class MaClasse:
    def ma_propriété(self):
        ...
    ma_propriété = property(ma_propriété)

    def changer_ma_propriété(self, nouvelle_valeur):
        ...
    ma_propriété = ma_propriété.setter(changer_ma_propriété)
```

Oui, c'est bien moins lisible au final et ce n'est définitivement pas la syntaxe conseillée. Néanmoins, cette comparaison devrait vous aider à mieux comprendre ce qui se passe quand nous utilisons les décorateurs sur `property`.

`property` est une classe. Elle peut être créée de différentes façons, mais les décorateurs sont clairement plus lisibles.

Les méthodes de classe

Jusqu'ici, nous avons travaillé exclusivement avec des méthodes d'instance, c'est-à-dire des méthodes prenant `self` en premier argument et travaillant sur l'instance (l'objet instancié). Il existe deux autres types de méthodes un peu plus rares que nous allons maintenant découvrir.

Commençons par les méthodes de classe. Elles ne travaillent pas sur l'instance (`self`), mais sur la classe-même (par convention, on l'appelle `cls`).

Quel est leur but ?

Il est assez rare de rencontrer des méthodes de classe, du moins si l'on considère à quel point il est fréquent de trouver des méthodes d'instance. Parfois, elles sont utilisées pour créer un objet et retourner son instance.

Comme `__init__` ?

Plus ou moins... sauf qu'`__init__` est une méthode d'instance (vous voyez qu'elle prend `self`). Nous allons éviter de rentrer dans le détail avant le prochain chapitre sur la construction d'objets.

Imaginons, par exemple, que nous avons une classe chargée d'analyser un fichier. La classe elle-même contient les méthodes pour analyser un fichier. L'objet de cette classe contient un fichier analysé. En d'autres termes, quand on veut analyser un fichier, l'objet n'existe pas encore. On pourrait donner ce rôle à `__init__`, mais le constructeur n'est, en théorie, pas censé effectuer d'opérations lourdes susceptibles de ralentir le programme (et l'analyse de gros fichiers risque en effet de prendre du temps).

Voici une solution possible :

```
1  class AnalyseurFichier:
2
3      """Classe représentant un analyseur de fichier.
4
5      Notez que l'objet de cette classe représente un fichier
   ↪   analysé. Utilisez donc la méthode 'analyser' de la classe
   ↪   pour analyser un fichier et retourner une instance déjà
   ↪   prête. Vous pouvez aussi créer un fichier non analysé en
   ↪   instanciant directement cette classe, mais gardez à
   ↪   l'esprit que l'objet sera vide.
6
7      Exemple d'utilisation :
8
9      >>> fichier = AnalyseurFichier.analyser("config.txt")
```

```
    """

    def __init__(self, nom_fichier):
        self.nom_fichier = nom_fichier
        self.contenu = ""
        self.analysé = False

    @classmethod
    def analyser(cls, nom_fichier):
        """Retourne un nouveau fichier analysé."""
        fichier_à_analyser = AnalyseurFichier(nom_fichier)

        # Lit le fichier
        with open(nom_fichier, "r") as fichier:
            fichier_à_analyser.contenu = fichier.read()

        fichier_à_analyser.analysé = True
        return fichier_à_analyser
```

Notre classe peut s'utiliser de deux façons : en instanciant un analyseur de fichier vide (via l'appel au constructeur, comme d'habitude), ou en appelant la méthode de classe `analyser` qui va non seulement créer un analyseur, mais aussi lire le fichier et le retourner analysé.

La ligne importante de la documentation (oui, lisez bien la documentation) est la suivante :

```
fichier = AnalyseurFichier.analyser("config.txt")
```

Nous appelons la méthode `analyser`, non pas sur l'objet (il n'est pas encore créé), mais sur la classe. En contre-partie, dans notre définition de classe, vous voyez les lignes :

```
    @classmethod
    def analyser(cls, nom_fichier):
```

Vous devriez d'abord reconnaître un décorateur en la ligne `@classmethod`. La définition est assez simple... sauf qu'au lieu de prendre `self` en premier paramètre (je vous rappelle qu'aucun objet n'est créé à ce stade), elle prend la classe (`cls`).

Là encore, notre syntaxe de méthode de classe pourrait donc être :

```
    def analyser(cls, nom_fichier):
        ...
    analyser = classmethod(analyser)
```

La première syntaxe, avec les décorateurs, est tout de même plus lisible.

Les méthodes statiques

On peut également définir un autre type de méthode dans nos classes. Si les méthodes de classe sont rares, les méthodes statiques le sont encore davantage. Elles ne prennent ni

l'instance (`self`), ni la classe (`cls`). En un mot... ce sont des fonctions, ni plus ni moins, mais qui sont définies dans le corps de la classe pour regrouper des fonctionnalités.

Là encore, l'intérêt peut vous sembler limité. Il s'agit d'une nouvelle occasion d'utiliser des décorateurs, et c'est une syntaxe assez courante, mais vous n'en aurez probablement pas besoin dans l'immédiat. Pour cette fois, je vais me contenter de vous donner la syntaxe et vous laisser faire des essais, si vous êtes intéressés :

```
class MaClasse:

    @staticmethod
    def ma_méthode():
        print("Faisons quelque chose d'utile !")
```

Notez bien que notre méthode statique ne prend ni `self`, ni `cls` en premier argument. Il s'agit d'une méthode statique (une simple fonction contenue dans une classe). Vous pouvez donc l'appeler très simplement :

```
>>> MaClasse.ma_méthode()
Faisons quelque chose d'utile !
>>> objet = MaClasse()
>>> objet.ma_méthode()
Faisons quelque chose d'utile !
>>>
```

Notez que nous pouvons appeler notre méthode statique depuis la classe (`MaClasse.ma_méthode`) ou depuis un objet de la classe (`objet.ma_méthode()`). Cela revient strictement au même pour Python.

Le code précédent pourrait se traduire par le suivant :

```
class MaClasse:

    def ma_méthode():
        print("Faisons quelque chose d'utile !")
    ma_méthode = staticmethod(ma_méthode)
```

C'est tout pour les méthodes statiques. Vous êtes libres de les utiliser si vous trouvez un cas cohérent d'application.

Les classes de données (dataclasses)

Comme nous l'avons vu, les classes peuvent contenir des informations (attributs) et des comportements (méthodes). Parfois cependant, on n'a pas besoin d'avoir des comportements spécifiques, ce qui nous donne une classe sans méthode à part le constructeur et peut-être des méthodes d'affichage, voire de comparaison. C'est un cas assez fréquent.

Dans le TP sur le jeu du pendu par exemple, nous conservons simplement les points du joueur dans un dictionnaire. Que faire si nous souhaitons conserver plusieurs informations

par joueur ? La date de la dernière partie, le nombre de points, le nombre de parties jouées, le nombre de parties perdues... on pourrait certes tout conserver dans un tuple mais ce ne serait pas bien clair. Considérez aussi notre inventaire de magasin où l'on conserve des produits et leur prix : en pratique, le nom du produit et le prix seront probablement conservés ensemble dans un programme (pour faciliter leur modification), tandis que la quantité (que ce soit en stock ou dans le panier d'un acheteur) sera probablement variable en fonction de la situation. Pour conserver nos produits, on pourrait donc avoir une classe `Produit` qui contiendrait un nom et un prix. Là encore, il est inutile d'avoir un comportement spécifique (si on veut changer le nom ou le prix d'un produit, on modifiera l'attribut correspondant). On aurait donc une classe assez petite qui ne contiendrait qu'un constructeur et sans doute des méthodes d'affichage.

Depuis Python 3.7, il existe un module, `dataclasses`, qui contient le mécanisme pour créer rapidement ces classes de données. Définir une classe `Produit` avec un nom et un prix est assez simple : on utilise les annotations de type.

```
from dataclasses import dataclass

@dataclass
class Produit:

    """Un produit, avec un nom et un prix (unitaire)."""

    nom: str
    prix: float
```

Et en action :

```
>>> pomme = Produit(nom="pomme rouge", prix=1.19)
>>> pomme
Produit(nom='pomme rouge', prix=1.19)
>>> pomme.nom
'pomme rouge'
>>> pomme.prix
1.19
>>> poire = Produit("poire", 1.12)
>>> print(poire)
Produit(nom='poire', prix=1.12)
>>> nouvelle_pomme = Produit(nom="pomme rouge", prix=1.19)
>>> pomme == nouvelle_pomme # Ils ont le même nom et même
 ↪  prix
True
>>> pomme is nouvelle_pomme # Mais ce sont bien deux objets
 ↪  différents
False
>>>
```

Comme vous le voyez, avec `dataclasses.dataclass`, vous pouvez créer des classes de données. La syntaxe est assez claire : elle utilise les annotations de type mais, au lieu

de spécifier l'annotation dans un paramètre de fonction, on la précise pour un attribut de classe. On peut aussi définir des valeurs par défaut.

Est-ce qu'on peut créer un produit avec un prix du mauvais type ?

Comme écrit plus haut, les annotations de type ne valident pas le type des données et c'est toujours le cas dans `dataclasses.dataclass`. En revanche, certains outils externes (comme Pydantic) proposent cette validation à la création et la modification d'objets.

Donc... ce petit code évite juste de définir un constructeur ?

Pas seulement. Il y a le support de l'affichage (méthode `__repr__`). Si vous regardez bien, la comparaison a été créée également (support pour la méthode `__eq__`) : un produit est égal à un autre s'il a le même nom et le même prix. C'est pratique si l'on conserve ces produits par exemple.

Vous pouvez en apprendre plus sur `dataclasses` en accédant à la documentation officielle : `https://docs.python.org/fr/3/library/dataclasses.html`

Il existe bien d'autres décorateurs définis par Python. `property`, `classmethod`, `staticmethod` et `dataclasses.dataclass` n'étaient que quatre exemples assez courants. Vous en trouverez d'autres au hasard de votre utilisation de Python, surtout si vous avez la curiosité de jeter un œil aux modules de la bibliothèque standard.

En résumé

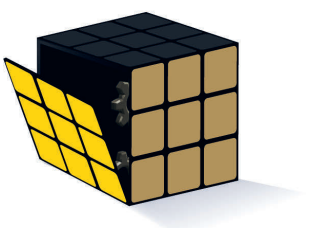
- Les décorateurs modifient le comportement d'une fonction.
- On peut déclarer une fonction comme décorée en plaçant, au-dessus de la ligne de sa définition, la ligne `@nom_décorateur`.
- Au moment de la définition de la fonction, le décorateur est appelé et la fonction qu'il renvoie sera celle utilisée.
- Les décorateurs peuvent également prendre des paramètres pour influer sur le comportement de la fonction décorée.

Chapitre 26

Les métaclasses

Difficulté : 

Toujours plus loin vers la métaprogrammation ! Nous allons ici nous intéresser au concept des métaclasses, ou comment générer des classes à partir... d'autres classes ! Je ne vous cache pas qu'il s'agit d'un concept assez avancé de la programmation Python.



Retour sur le processus d'instanciation

Depuis la troisième partie de ce cours, nous avons créé bon nombre d'objets. Nous avons découvert au début de cette partie le **constructeur**, cette méthode appelée quand on souhaite créer un objet.

Je vous ai dit alors que les choses étaient un peu plus complexes qu'il y paraissait. Nous allons maintenant voir en quoi !

Admettons que vous ayez défini une classe :

```
class Personne:

    """Classe définissant une personne.

    Elle possède comme attributs :
    nom : le nom de la personne
    prénom : son prénom
    âge : son âge
    lieu_résidence : son lieu de résidence

    Le nom et le prénom doivent être passés au constructeur.

    """

    def __init__(self, nom, prénom):
        self.nom = nom
        self.prénom = prénom
        self.âge = 23
        self.lieu_résidence = "Lyon"
```

Cette syntaxe n'a rien de nouveau pour nous.

Maintenant, que se passe-t-il quand on souhaite créer une personne ? Facile, on rédige le code suivant :

```
personne = Personne("Doe", "John")
```

Lorsque l'on exécute cela, Python appelle notre constructeur `__init__` en lui transmettant les arguments fournis à la construction de l'objet. Il y a cependant une étape intermédiaire.

Si vous examinez la définition de notre constructeur :

```
def __init__(self, nom, prénom):
```

Vous ne remarquez rien d'étrange ? Peut-être pas, car vous avez été habitués à cette syntaxe depuis le début de cette partie : la méthode prend en premier paramètre `self`.

Or, `self`, vous vous en souvenez, c'est l'objet que nous manipulons. Sauf que, quand on crée un objet… on souhaite récupérer un nouvel objet mais on n'en passe aucun à la classe.

D'une façon ou d'une autre, notre classe crée un nouvel objet et le passe à notre constructeur. La méthode `__init__` se charge d'écrire ses attributs, mais elle n'est pas responsable de la création de notre objet. Nous allons à présent voir qui s'en charge.

La méthode `__new__`

La méthode `__init__` est là pour *initialiser* notre objet (en écrivant des attributs dedans, par exemple) mais elle n'est pas là pour le *créer*. La méthode qui s'en charge, c'est `__new__`.

C'est une méthode spéciale, vous en reconnaissez la particularité. C'est également une méthode définie par `object`, que l'on peut redéfinir en cas de besoin.

Regardons plus précisément ce qui se passe quand on tente de construire un objet :

- On demande à créer un objet, en écrivant par exemple `Personne("Doe", "John")`.
- La méthode `__new__` de notre classe (ici `Personne`) est appelée et se charge de construire un nouvel objet.
- Si `__new__` renvoie une instance de la classe, on appelle le constructeur `__init__` en lui passant en paramètres cette nouvelle instance ainsi que les arguments passés lors de la création de l'objet.

Maintenant, intéressons-nous à la structure de notre méthode `__new__`.

C'est une méthode de classe, ce qui signifie qu'elle ne prend pas `self` en paramètre. C'est logique, d'ailleurs, son but étant de créer une nouvelle instance. Elle prend la classe manipulée `cls`.

Autrement dit, quand on souhaite créer un objet de la classe `Personne`, la méthode `__new__` de cette dernière est appelée et prend comme premier paramètre la classe `Personne` elle-même.

`__new__` est une méthode de classe implicite, il n'est pas nécessaire de la décorer avec `@classmethod` comme nous l'avons vu dans le chapitre précédent.

Les autres paramètres passés à la méthode `__new__` seront transmis au constructeur.

Voyons un peu cela, exprimé sous forme de code :

```
1  class Personne:
2
3      """Classe définissant une personne.
4
5      Elle possède comme attributs :
6      nom : le nom de la personne
7      prénom : son prénom
8      âge : son âge
9      lieu_résidence : son lieu de résidence
10
```

```
    Le nom et le prénom doivent être passés au constructeur.

    """

    def __new__(cls, nom, prénom):
        print(f"Appel de la méthode __new__ de la classe
        ↪ {cls}")
        # On laisse le travail à object
        return object.__new__(cls)
        # Que l'on pourrait aussi écrire... :
        #    return super().__new__(cls)

    def __init__(self, nom, prénom):
        print(f"Appel de la méthode __init__ sur l'objet
        ↪ {self}")
        self.nom = nom
        self.prénom = prénom
        self.âge = 23
        self.lieu_résidence = "Lyon"
```

Essayons de créer une personne :

```
>>> personne = Personne("Doe", "John")
Appel de la méthode __new__ de la classe <class 'Personne'>
Appel de la méthode __init__ sur l'objet <Personne object at
↪ 0x03B5EC50>
>>>
```

Redéfinir `__new__` permet, par exemple, de créer une instance d'une autre classe. Elle est principalement utilisée par Python pour produire des types **immuables** (en anglais, *immutable*), que l'on ne peut modifier, comme le sont les chaînes de caractères, les tuples, les entiers, les flottants…

La méthode `__new__` est parfois redéfinie dans le corps d'une métaclasse. Nous allons à présent découvrir ce dont il s'agit.

Créer une classe dynamiquement

Je le répète une nouvelle fois : *en Python, tout est objet*. Cela veut dire que les entiers, les flottants, les listes sont des objets, que les modules sont des objets, que les *packages* sont des objets… mais également que les classes sont des objets !

La méthode que nous connaissons

Pour créer une classe, nous n'avons vu qu'une méthode, la plus utilisée, faisant appel au mot-clé `class`.

```
class MaClasse:
```

Vous pouvez ensuite créer des instances sur le modèle de cette classe, je ne vous apprends rien.

Là où cela se complique, c'est que les classes sont également des objets.

Si les classes sont des objets... cela veut dire que les classes sont elles-mêmes modelées sur des classes ?

Eh oui. Les classes, comme tout objet, sont modelées sur une classe. Peut-être l'extrait de code suivant vous aidera-t-il à comprendre l'idée.

```
>>> type(5)
<class 'int'>
>>> type("une chaîne")
<class 'str'>
>>> type([1, 2, 3])
<class 'list'>
>>> type(int)
<class 'type'>
>>> type(str)
<class 'type'>
>>> type(list)
<class 'type'>
>>>
```

On demande le type d'un entier et, sans surprise, Python nous répond `class int`. Cependant, si on lui demande la classe de `int`, Python nous répond `class type`.

En fait, par défaut, toutes nos classes sont modelées sur la classe `type`. Cela signifie que :

1. quand on crée une nouvelle classe (`class Personne:` par exemple), Python appelle la méthode `__new__` de la classe `type` ;
2. une fois la classe créée, on appelle le constructeur `__init__` de la classe `type`.

Cela semble sans doute encore obscur. Ne désespérez pas, vous comprendrez peut-être un peu mieux ce dont je parle en lisant la suite. Sinon, n'hésitez pas à relire ce passage et à faire des tests par vous-mêmes.

Créer une classe dynamiquement

Résumons :

— nous savons que les objets sont modelés sur des classes ;
— nous savons que nos classes, étant elles-mêmes des objets, sont modelées sur une classe ;

— la classe sur laquelle toutes les autres sont modelées par défaut s'appelle `type`.

Essayons de créer une classe dynamiquement, sans passer par le mot-clé `class` mais par la classe `type` directement, qui prend trois arguments :

— le nom de la classe à créer ;
— un **tuple** contenant les classes dont notre nouvelle classe va hériter ;
— un dictionnaire contenant les attributs et méthodes de notre classe.

```
>>> Personne = type("Personne", (), {})
>>> Personne
<class '__main__.Personne'>
>>> john = Personne()
>>> dir(john)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__',
 ↪  '__format__', '__ge__', '__getattribute__', '__gt__',
 ↪  '__hash__', '__init__', '__le__', '__lt__', '__module__',
 ↪  '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 ↪  '__repr__', '__setattr__', '__sizeof__', '__str__',
 ↪  '__subclasshook__', '__weakref__']
>>>
```

J'ai simplifié le code au maximum. Nous créons bel et bien une nouvelle classe que nous stockons dans notre variable `Personne`, mais elle est vide. Elle n'hérite d'aucune classe et elle ne définit aucun attribut ni méthode de classe.

Créons deux méthodes pour notre classe :

— un constructeur `__init__` ;
— une méthode `__str__` affichant le prénom et le nom de la personne.

```
def créer_personne(personne, nom, prénom):
    """La fonction qui jouera le rôle de constructeur pour
     ↪  notre classe Personne.

    Elle prend en paramètre, outre la personne :
    nom -- son nom
    prénom -- son prenom

    """
    personne.nom = nom
    personne.prénom = prénom
    personne.âge = 21
    personne.lieu_résidence = "Lyon"

def présenter_personne(personne):
    """Fonction présentant la personne.

    Elle affiche son prénom et son nom.

```

```
    """
    return f"{personne.prénom} {personne.nom}"

# Dictionnaire des méthodes
méthodes = {
    "__init__": créer_personne,
    "__str__": présenter_personne,
}

# Création dynamique de la classe
Personne = type("Personne", (), méthodes)
```

Voyons les effets de ce code :

```
>>> john = Personne("Doe", "John")
>>> john.nom
'Doe'
>>> john.prénom
'John'
>>> john.âge
21
>>> str(john)
'John Doe'
>>>
```

Je ne vous le cache pas, c'est une fonctionnalité que vous utiliserez sans doute assez rarement. Cette explication est une façon d'aborder les métaclasses.

Pour l'heure, décomposons notre code :

1. On commence par créer deux fonctions, `créer_personne` et `présenter_personne`. Elles sont amenées à devenir les méthodes `__init__` et `__str__` de notre future classe. Étant de futures méthodes d'instance, elles doivent prendre en premier paramètre l'objet manipulé. Nous aurions pu appeler ce premier argument `self`, mais je trouvais cela assez déconcertant, puisque ce sont des fonctions pour cet exemple.
2. On place ces deux fonctions dans un dictionnaire. En clé se trouve le nom de la future méthode et en valeur, la fonction correspondante.
3. Enfin, on fait appel à `type` en lui passant, en troisième paramètre, le dictionnaire que l'on vient de constituer.

Si vous essayez d'ajouter des attributs dans ce dictionnaire passé à `type`, vous devez être conscients du fait qu'il s'agira d'attributs de classe, pas d'attributs d'instance.

Définition d'une métaclasse

Nous avons vu que `type` est la métaclasse de toutes les classes par défaut. Cependant, une classe peut posséder une autre métaclasse que `type`.

Construire une métaclasse se définit de la même façon que construire une classe. Les métaclasses héritent de `type`. Nous allons retrouver la structure de base des classes que nous avons étudiée auparavant.

Nous allons notamment nous intéresser à deux méthodes :

— `__new__`, appelée pour créer une classe ;
— `__init__`, appelée pour construire la classe.

La méthode `__new__`

Elle prend quatre paramètres :

— la métaclasse servant de base ;
— le nom de notre nouvelle classe ;
— un **tuple** contenant les classes dont elle hérite ;
— le dictionnaire de ses attributs et méthodes.

Les trois derniers paramètres, vous devriez les reconnaître : ce sont les mêmes que ceux passés à `type`.

Voici une méthode `__new__` minimaliste.

```
class MaMétaClasse(type):

    """Exemple d'une métaclasse."""

    def __new__(metacls, nom, bases, dict):
        """Création de notre classe."""
        print(f"On crée la classe {nom}")
        return type.__new__(metacls, nom, bases, dict)
```

Pour dire qu'une classe prend comme métaclasse autre chose que `type`, c'est dans la ligne de la définition de la classe que cela se passe :

```
class MaClasse(metaclass=MaMétaClasse):
    pass
```

En exécutant ce code, vous pouvez voir :

```
On crée la classe MaClasse
```

La méthode `__init__`

Le constructeur d'une métaclasse prend les mêmes paramètres que `__new__`, sauf le premier, qui n'est plus la métaclasse servant de modèle mais la classe que l'on vient de créer.

Les trois paramètres suivants restent les mêmes : le nom, le **tuple** des classes-mères et le dictionnaire des attributs et méthodes de classe.

Il n'y a rien de très compliqué dans le procédé. L'exemple précédent peut être repris en le modifiant quelque peu pour qu'il s'adapte à la méthode `__init__`.

Maintenant, voyons concrètement à quoi cela peut servir.

Les métaclasses en action

Comme vous vous en doutez, les métaclasses sont généralement utilisées pour des besoins assez complexes. L'exemple le plus répandu est une métaclasse chargée de tracer l'appel de ses méthodes. Autrement dit, dès qu'on appelle une méthode d'un objet, une ligne s'affiche pour le signaler. Toutefois, cet exemple est assez difficile à comprendre car il fait appel à la fois au concept des métaclasses et à celui des décorateurs, pour décorer les méthodes tracées.

Je vous propose quelque chose de plus simple. Il va de soi qu'il existe bien d'autres usages, dont certains complexes.

Essayons de garder nos classes créées dans un dictionnaire les prenant comme valeurs et leurs noms comme clés.

Par exemple, dans une bibliothèque destinée à construire des interfaces graphiques, on trouve plusieurs *widgets*[1] comme des boutons, des cases à cocher, des menus, des cadres... Généralement, ces objets sont des classes héritant d'une classe mère commune. En outre, l'utilisateur peut, en cas de besoin, créer ses propres classes héritant de celles de la bibliothèque.

Par exemple, la classe mère de tous nos *widgets* s'appellera `Widget`. De cette classe hériteront `Bouton`, `CaseÀCocher`, `Menu`, `Cadre`, etc. L'utilisateur de la bibliothèque pourra par ailleurs en dériver ses propres classes.

Le dictionnaire que l'on aimerait créer se présente comme suit :

```
{
    "Widget": Widget,
    "Bouton": Bouton,
    "CaseÀCocher": CaseÀCocher,
    "Menu": Menu,
    "Cadre": Cadre,
    ...
}
```

Ce dictionnaire pourrait être rempli manuellement à chaque fois qu'on crée une classe héritant de `Widget` mais avouez que ce ne serait pas très pratique.

Dans ce contexte, les métaclasses nous facilitent la vie. Essayez de résoudre l'exercice, le code n'est pas trop complexe. Cela dit, étant donné qu'on a vu beaucoup de choses dans ce chapitre et que les métaclasses sont un concept plutôt avancé, je vous donne directement le code qui vous aidera à comprendre le mécanisme :

```
trace_classes = {} # Notre dictionnaire vide

class MétaWidget(type):

    """Notre métaclasse pour nos Widgets.

```

1. Ce sont des objets graphiques.

```
    Elle hérite de type, puisque c'est une métaclasse. Elle va
↪   écrire dans le dictionnaire trace_classes à chaque fois
↪   qu'une classe sera créée, utilisant cette métaclasse
↪   naturellement.

    """

    def __init__(cls, nom, bases, dict):
        """Constructeur de notre métaclasse, appelé quand on
        ↪  crée une classe."""
        type.__init__(cls, nom, bases, dict)
        trace_classes[nom] = cls
```

Ce n'est pas trop compliqué pour l'heure. Créons notre classe `Widget` :

```
class Widget(metaclass=MétaWidget):

    """Classe mère de tous nos widgets."""

    pass
```

Après avoir exécuté ce code, vous constatez que notre classe `Widget` a bien été ajoutée dans notre dictionnaire :

```
>>> trace_classes
{'Widget': <class '__main__.Widget'>}
>>>
```

Maintenant, construisons une nouvelle classe héritant de `Widget`.

```
class Bouton(Widget):

    """Une classe définissant le widget bouton."""

    pass
```

Si vous affichez de nouveau le contenu du dictionnaire, vous vous rendez compte que la classe `Bouton` a bien été ajoutée. Héritant de `Widget`, elle reprend la même métaclasse (sauf mention contraire explicite) et elle est donc ajoutée au dictionnaire.

Vous pouvez étoffer cet exemple, faire en sorte que l'aide de la classe soit également conservée, ou qu'une exception soit levée si une classe du même nom existe déjà dans le dictionnaire.

Pour conclure

Les métaclasses sont un concept de programmation assez avancé, puissant mais délicat à comprendre de prime abord. Je vous invite, en cas de doute, à tester par vous-mêmes ou à rechercher d'autres exemples, ils sont nombreux.

En résumé

— Le processus d'instanciation d'un objet est assuré par deux méthodes, `__new__` et `__init__`.
— `__new__` est chargée de la création de l'objet et prend en premier paramètre sa classe.
— `__init__` est chargée de l'initialisation des attributs de l'objet et prend en premier paramètre l'objet précédemment créé par `__new__`.
— Les classes étant des objets, elles sont toutes modelées sur une **métaclasse**.
— À moins d'être explicitement modifiée, la métaclasse de toutes les classes est `type`.
— On peut utiliser `type` pour créer des classes dynamiquement.
— On peut faire hériter une classe de `type` pour créer une nouvelle métaclasse.
— Dans le corps d'une classe, pour spécifier sa métaclasse, on exploite la syntaxe suivante : `class MaClasse(metaclass=NomDeLaMetaClasse):`.

Quatrième partie

Les merveilles de la bibliothèque standard

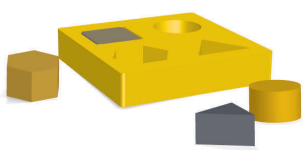
Chapitre 27

Les expressions régulières

Difficulté :

Dans ce chapitre, je vais m'attarder sur les **expressions régulières** et sur le module `re` qui sert à les manipuler. En quelques mots, sachez que les expressions régulières facilitent et accélèrent les recherches sur des chaînes de caractères.

Il existe, naturellement, bien d'autres modules pour manipuler du texte. C'est toutefois sur celui-ci que je vais m'attarder, tout en vous donnant les moyens d'aller plus loin si vous le désirez.



Que sont les expressions régulières ?

Les **expressions régulières** sont un puissant moyen de rechercher et d'isoler des expressions dans une chaîne de caractères.

Pour simplifier, imaginez que vous codez un programme qui demande un certain nombre d'informations à l'utilisateur afin de les stocker dans un fichier. Lui demander son nom, son prénom et quelques autres informations, ce n'est pas bien difficile : on va utiliser la fonction `input` et récupérer le résultat. Jusqu'ici, rien de nouveau.

En revanche, que se passe-t-il si on demande à l'utilisateur de fournir un numéro de téléphone ? Qu'est-ce qui l'empêche de taper n'importe quoi ? Lorsqu'on lui demande de fournir une adresse e-mail et qu'il tape quelque chose d'invalide, par exemple « je_ne_te_donnerai_pas_mon_email », que va-t-il se passer quand on essayera de lui envoyer automatiquement un courriel ?

Si ce cas n'est pas géré, vous risquez d'avoir un problème. Les expressions régulières sont un moyen de rechercher, d'isoler ou de remplacer des expressions dans une chaîne. Ici, elles nous permettraient de vérifier que le numéro de téléphone saisi compte bien dix chiffres, qu'il commence par un 0 et qu'il compte éventuellement des séparateurs tous les deux chiffres. Si ce n'était pas le cas, on demanderait à l'utilisateur de le saisir à nouveau.

Quelques éléments de syntaxe pour les expressions régulières

Si vous connaissez déjà les expressions régulières et leur syntaxe, passez directement à la section consacrée au module `re`. Sinon, sachez que je ne pourrai vous les présenter que brièvement, car c'est un sujet très vaste, qui mérite un livre à lui tout seul.

Concrètement, comment cela se présente-t-il ?

Le module `re` sert à faire des recherches très précises dans des chaînes de caractères et à y remplacer des éléments, le tout en fonction de critères particuliers. Ces critères, ce sont nos **expressions régulières**. Pour nous, elles se présentent sous la forme de chaînes de caractères.

Des caractères ordinaires

Quand on forme une expression régulière, on peut utiliser des caractères spéciaux et d'autres qui ne le sont pas. Par exemple, si nous recherchons le mot `chat` dans notre chaîne, nous pouvons écrire comme expression régulière la chaîne « `chat` ». Jusque là, rien n'est très compliqué.

Toutefois, vous vous doutez bien que les expressions régulières ne se limitent pas à ce type de recherche extrêmement simple, sans quoi les méthodes `find` et `replace` de la classe `str` auraient suffit.

Rechercher au début ou à la fin de la chaîne

Vous recherchez au début de la chaîne en plaçant en tête de votre regex[1] le signe d'accent circonflexe `^`. Si, par exemple, vous voulez rechercher la syllabe `cha` en début de votre chaîne, vous écrirez donc l'expression `^cha`. Cette expression sera trouvée dans la chaîne `'chaton'` mais pas dans la chaîne `'achat'`.

Pour matérialiser la fin de la chaîne, vous utiliserez le signe `$`. Ainsi, l'expression `q$` sera trouvée uniquement si votre chaîne se termine par la lettre q minuscule.

Contrôler le nombre d'occurrences

Les caractères spéciaux que nous allons découvrir contrôlent le nombre de fois où notre expression apparaît dans notre chaîne.

```
1 chat*
```

Nous avons ajouté un astérisque (`*`) après le caractère `t` de `chat`. Cela signifie que notre lettre `t` pourra se retrouver 0, 1, 2, ... fois dans notre chaîne. Autrement dit, notre expression `chat*` sera trouvée dans les chaînes suivantes : `'chat'`, `'chaton'`, `'chateau'`, `'herbe à chat'`, `'chapeau'`, `'chatterton'`, `'chatttttttttt'`...

Regardez un à un ces exemples pour vérifier que vous les comprenez bien. On trouvera dans chacune de ces chaînes l'expression régulière `chat*`. Traduite en français, elle signifie : « on recherche une lettre `c` suivie d'une lettre `h` suivie d'une lettre `a` suivie, éventuellement, d'une lettre `t` qu'on peut trouver zéro, une ou plusieurs fois ». Peu importe que ces lettres soient trouvées au début, à la fin ou au milieu de la chaîne.

Un autre exemple ? Considérez l'expression régulière suivante et essayez de la comprendre :

```
1 bat*e
```

Cette expression est trouvée dans les chaînes suivantes : `'bateau'`, `'batteur'` et `'joan baez'`.

Dans nos exemples, le signe `*` n'agit que sur la lettre qui le précède directement, pas sur les autres.

Il existe d'autres signes pour contrôler le nombre d'occurrences d'une lettre. Je vous ai fait un petit récapitulatif dans le tableau suivant, en prenant des exemples d'expressions avec les lettres `a`, `b` et `c` :

Signe	Explication	Expression	Chaînes contenant l'expression
*	0, 1 ou plus	abc*	`'ab'`, `'abc'`, `'abcc'`, `'abccccc'`
+	1 ou plus	abc+	`'abc'`, `'abcc'`, `'abccc'`
?	0 ou 1	abc?	`'ab'`, `'abc'`

1. Abréviation de *Regular Expression*.

Vous pouvez également contrôler précisément le nombre d'occurrences grâce aux accolades :

— `E{4}` : signifie 4 fois la lettre E majuscule ;
— `E{2,4}` : signifie de 2 à 4 fois la lettre E majuscule ;
— `E{,5}` : signifie de 0 à 5 fois la lettre E majuscule ;
— `E{8,}` : signifie 8 fois minimum la lettre E majuscule.

Les classes de caractères

Il est possible de préciser entre crochets plusieurs caractères ou classes de caractères. Par exemple, si vous écrivez `[abcd]`, cela signifie : l'une des lettres parmi `a`, `b`, `c` et `d`.

Pour exprimer des classes, utilisez le tiret `-` entre deux lettres. Par exemple, l'expression `[A-Z]` signifie « une lettre majuscule ». Vous pouvez préciser plusieurs classes ou possibilités dans votre expression. Ainsi, l'expression `[A-Za-z0-9]` signifie « une lettre, majuscule ou minuscule, ou un chiffre ».

Vous contrôlez l'occurrence des classes comme pour les caractères seuls. Si vous voulez par exemple rechercher 5 lettres majuscules qui se suivent dans une chaîne, votre expression sera `[A-Z]{5}`.

Les groupes

Comme je l'ai dit précédemment, si vous voulez contrôler le nombre d'occurrences d'un caractère, vous ajoutez derrière celui-ci un signe particulier (astérisque, point d'interrogation, accolades). Pour appliquer ce contrôle d'occurrences à plusieurs caractères, placez-les entre parenthèses.

```
1 (cha){2,5}
```

Cette expression sera vérifiée pour les chaînes contenant la séquence `'cha'` répétée entre deux et cinq fois. Les séquences `'cha'` doivent se suivre naturellement.

Les groupes sont également utiles pour remplacer des portions de notre chaîne, mais nous y reviendrons plus loin.

Le module `re`

Le module `re` a été spécialement conçu pour travailler avec les expressions régulières (*Regular Expressions*). Il définit plusieurs fonctions utiles, que nous allons découvrir, ainsi que des objets propres pour modéliser des expressions.

Chercher dans une chaîne

Nous allons pour ce faire utiliser la fonction `search` du module `re`. Bien entendu, il faut d'abord l'importer.

```
>>> import re
>>>
```

La fonction `search` attend deux paramètres obligatoires : l'expression régulière, sous la forme d'une chaîne, et la chaîne de caractères dans laquelle on recherche cette expression. Si l'expression est trouvée, la fonction renvoie un objet symbolisant l'expression recherchée. Sinon, elle renvoie `None`.

Certains caractères spéciaux dans nos expressions régulières sont modélisés par la barre oblique inverse `\`. Vous savez sans doute que Python représente d'autres caractères avec ce symbole. Par exemple, si vous écrivez `\n` dans une chaîne, Python effectuera un saut de ligne.

Pour symboliser les caractères spéciaux dans les expressions régulières, il est nécessaire d'échapper la barre oblique inverse en la faisant précéder d'une autre. Cela veut dire que, pour écrire le caractère spécial `\w`, vous devez le coder `\\w`.

C'est assez peu pratique et parfois gênant pour la lisibilité. C'est pourquoi je vous conseille d'utiliser un format de chaîne que nous n'avons pas vu jusqu'à présent : en plaçant un `r` avant le délimiteur qui ouvre notre chaîne, toutes les barres obliques inverses qu'elle contient sont échappées.

```
>>> r'\n'
'\\n'
>>>
```

Revenons à notre fonction `search` et mettons en pratique ce que nous avons appris précédemment :

```
>>> re.search(r"abc", "abcdef")
<_sre.SRE_Match object at 0x00AC1640>
>>> re.search(r"abc", "abacadaeaf")
>>> re.search(r"abc*", "ab")
<_sre.SRE_Match object at 0x00AC1800>
>>> re.search(r"abc*", "abccc")
<_sre.SRE_Match object at 0x00AC1640>
>>> re.search(r"chat*", "chateau")
<_sre.SRE_Match object at 0x00AC1800>
>>>
```

Comme vous le voyez, si l'expression est trouvée dans la chaîne, un objet de la classe `_sre.SRE_Match` est renvoyé. Si l'expression n'est pas trouvée, la fonction renvoie `None`.

Cela fait qu'il est extrêmement facile de savoir si une expression est contenue dans une chaîne :

```
if re.match(expression, chaîne) is not None:
    # Si l'expression est dans la chaîne
```

```
    # Ou alors, plus intuitivement
if re.match(expression, chaîne):
```

N'hésitez pas à tester des syntaxes plus complexes et plus utiles. Tenez, par exemple, comment obliger l'utilisateur à saisir un numéro de téléphone ?

Avec le bref descriptif que je vous ai donné dans ce chapitre, vous pouvez théoriquement y arriver. Toutefois, c'est quand même une regex assez complexe alors je vous la donne : prenez le temps de la décortiquer si vous le souhaitez.

Notre regex doit vérifier qu'une chaîne est un numéro de téléphone. L'utilisateur peut saisir un numéro de différentes façons :

- `0X XX XX XX XX`
- `0X-XX-XX-XX-XX`
- `0X.XX.XX.XX.XX`
- `0XXXXXXXXX`

Autrement dit :

- le premier chiffre doit être un 0 ;
- le second chiffre, ainsi que tous ceux qui suivent (9 en tout, sans compter le 0 d'origine) doivent être compris entre 0 et 9 ;
- tous les deux chiffres, on peut avoir un délimiteur optionnel (un tiret, un point ou un espace).

Voici la regex que je vous propose :

```
^0[0-9]([ .-]?[0-9]{2}){4}$
```

ARGH ! C'est illisible ton truc !

Je reconnais que c'est assez peu clair. Décomposons la formule :

- D'abord, on trouve un caractère accent circonflexe `^` qui veut dire qu'on cherche l'expression au début de la chaîne. Vous pouvez aussi voir, à la fin de la regex, le symbole `$` qui veut dire que l'expression doit être à la fin de la chaîne. Si l'expression doit être au début et à la fin de la chaîne, cela signifie que la chaîne dans laquelle on recherche ne doit rien contenir d'autre que l'expression.
- Nous avons ensuite le `0` qui veut simplement dire que le premier caractère de notre chaîne doit être un `0`.
- Nous avons ensuite une classe de caractères `[0-9]`. Cela signifie qu'après le `0`, on doit trouver un chiffre compris entre 0 et 9 (peut-être 0, peut-être 1, peut-être 2…).
- Ensuite, cela se complique. Vous avez une parenthèse qui matérialise le début d'un groupe. Dans ce groupe, nous trouvons, dans l'ordre :
  - D'abord une classe `[ .-]` qui veut dire « soit un espace, soit un point, soit un tiret ». Juste après cette classe, vous avez un signe `?` qui signifie que cette classe est optionnelle.

- Après la définition de notre délimiteur, nous trouvons une classe `[0-9]` qui signifie encore une fois « un chiffre entre 0 et 9 ». Après cette classe, entre accolades, vous pouvez voir le nombre de chiffres attendus (2).
- Ce groupe, contenant un séparateur optionnel et deux chiffres, doit se retrouver quatre fois dans notre expression (après la parenthèse fermante, vous trouvez entre accolades le contrôle du nombre d'occurrences).

Si vous regardez bien nos numéros de téléphone, vous vous rendez compte que notre regex s'applique aux différents cas présentés. La définition de notre numéro de téléphone n'est pas vraie pour tous les numéros. Cette regex est un exemple et même une base pour vous permettre de saisir le concept.

Si vous voulez que l'utilisateur saisisse un numéro de téléphone, voici le code auquel vous pourriez arriver :

```
import re
chaîne = ""
expression = r"^0[0-9]([ .-]?[0-9]{2}){4}$"
while not re.search(expression, chaîne):
    chaîne = input("Saisissez un numéro de téléphone (valide)
    ↪  : ")
```

Remplacer une expression

Le remplacement est un peu plus complexe. Je ne vais pas vous montrer d'exemples réellement utiles car ils s'appuient en général sur des expressions assez difficiles à comprendre.

Pour remplacer une partie d'une chaîne de caractères sur la base d'une regex, nous allons utiliser la fonction `sub` du module `re`.

Elle prend trois paramètres :

- l'expression à rechercher ;
- par quoi remplacer cette expression ;
- la chaîne d'origine.

Elle renvoie la chaîne modifiée.

Des groupes numérotés

Pour remplacer une partie de l'expression, on doit d'abord utiliser des groupes. Si vous vous rappelez, les groupes sont indiqués entre parenthèses.

```
(a)b(cd)
```

Dans cet exemple, `(a)` est le premier groupe et `(cd)` est le second.

L'ordre des groupes est important dans cet exemple. Dans notre expression de remplacement, nous pouvons appeler nos groupes grâce à `\<numéro du groupe>`. Pour une fois, on compte à partir de 1.

Ce n'est pas très clair ? Regardez cet exemple simple :

```
>>> re.sub(r"(ab)", r" \1 ", "abcdef")
' ab cdef'
>>>
```

On se contente ici de remplacer `'ab'` par `' ab '`.

Je vous l'accorde, on serait parvenu au même résultat en utilisant la méthode `replace` de notre chaîne. Cependant, les expressions régulières sont bien plus précises que cela : vous commencez à vous en rendre compte, je pense.

Je vous laisse le soin de creuser la question, je préfère ne pas vous présenter tout de suite des expressions trop complexes.

Donner des noms à nos groupes

Nous pouvons également donner des noms à nos groupes. C'est parfois plus clair que de compter sur des numéros. Pour cela, il faut faire suivre la parenthèse ouvrant le groupe d'un point d'interrogation, d'un P majuscule et du nom du groupe entre chevrons <>.

```
(?P<id>[0-9]{2})
```

Dans l'expression de remplacement, on utilisera l'expression `\g<nom du groupe>` pour symboliser le groupe. Prenons un exemple :

```
>>> texte = """
... nom='Task1', id=8
... nom='Task2', id=31
... nom='Task3', id=127"""
... ...
...
>>> print(re.sub(r"id=(?P<id>[0-9]+)", r"id[\g<id>]", texte))
nom='Task1', id[8]
nom='Task2', id[31]
nom='Task3', id[127]
...
>>>
```

Des expressions compilées

Si, dans votre programme, vous utilisez plusieurs fois les mêmes expressions régulières, il est utile de les compiler. Le module `re` propose en effet de conserver votre expression régulière sous la forme d'un objet à stocker dans votre programme. Si vous devez chercher cette expression dans une chaîne, vous passez par des méthodes de l'expression. Cela vous fait gagner en performances si vous faites souvent appel à cette expression.

Par exemple, j'ai une expression qui est appelée quand l'utilisateur saisit son mot de passe. Je veux vérifier que son mot de passe fait bien six caractères au minimum et qu'il ne contient que des lettres majuscules, minuscules et des chiffres. Voici l'expression à laquelle j'arrive :

```
^[A-Za-z0-9]{6,}$
```

À chaque fois qu'un utilisateur saisit un mot de passe, le programme va appeler `re.search` pour vérifier que celui-ci respecte bien les critères de l'expression. Il serait plus judicieux de conserver l'expression en mémoire.

On utilise pour ce faire la méthode `compile` du module `re`. On stocke la valeur renvoyée (une expression régulière compilée) dans une variable, c'est un objet standard pour le reste.

```
chn_mdp = r"^[A-Za-z0-9]{6,}$"
exp_mdp = re.compile(chn_mdp)
```

Ensuite, vous pouvez utiliser directement cette expression compilée. Elle possède plusieurs méthodes utiles, dont `search` et `sub` que nous avons vu plus haut. À la différence des fonctions du module `re` portant les mêmes noms, elles ne prennent pas en premier paramètre l'expression (celle-ci se trouve directement dans l'objet).

Voyez plutôt :

```
chn_mdp = r"^[A-Za-z0-9]{6,}$"
exp_mdp = re.compile(chn_mdp)
mot_de_passe = ""
while not exp_mdp.search(mot_de_passe):
    mot_de_passe = input("Tapez votre mot de passe : ")
```

Vous en apprendrez plus, à la fois sur le module `re` et sur la syntaxe des expressions régulières, en lisant la documentation officielle du module : `https://docs.python.org/fr/3/library/re.html`

En résumé

- Les expressions régulières permettent de chercher et remplacer certaines expressions dans des chaînes de caractères.
- Le module `re` de Python sert à manipuler des expressions régulières en Python.
- La fonction `search` du module `re` cherche une expression dans une chaîne.
- Pour remplacer une certaine expression dans une chaîne, on utilise la fonction `sub` du module `re`.
- On peut également compiler les expressions régulières grâce à la fonction `compile` du module `re`.

Chapitre 28

Le temps

Difficulté :

Exprimer un temps en informatique, cela soulève quelques questions. Disposer d'une mesure du temps dans un programme autorise des applications variées : connaître la date et l'heure actuelles quand on remonte une erreur, calculer depuis combien de temps le programme a été lancé, gérer des alarmes programmées, faire des tests de performance... et j'en passe !

Il existe plusieurs façons de représenter des temps, que nous allons découvrir maintenant.

Pour bien suivre ce chapitre, vous aurez besoin de maîtriser l'objet : savoir ce qu'est un objet et comment en créer un.

Le module `time`

Le module `time` est sans doute le premier à être utilisé quand on souhaite manipuler des temps de façon simple.

Notez que, dans la documentation de la bibliothèque standard, ce module est classé dans la rubrique **Generic Operating System Services** [1]. Ce n'est pas un hasard : `time` est un module très proche du système. Cela signifie que certaines de ses fonctions risquent de donner des résultats différents sur des systèmes différents. Je vais surtout m'attarder sur les fonctionnalités les plus génériques possibles afin de ne perdre personne.

Comme toujours, je vous conseille de consulter la documentation officielle (en français) du module : `https://docs.python.org/fr/3/library/time.html`

Représenter une date et une heure dans un nombre unique

Comment représenter un temps ? Il existe, naturellement, plusieurs réponses à cette question. Celle que nous allons découvrir ici est sans doute la moins compréhensible pour un humain, mais la plus adaptée à un ordinateur : on stocke la date et l'heure dans un seul nombre.

Comment représenter une date et une heure dans un unique nombre ?

L'idée retenue a été de représenter une date et une heure en fonction du nombre de secondes écoulées depuis une date précise. La plupart du temps, cette date est l'`Epoch Unix`, soit le 1er janvier 1970 à 00 : 00 : 00.

Pourquoi cette date plutôt qu'une autre ?

Il fallait bien choisir une date de début. L'année 1970 a été considérée comme un bon départ, compte tenu de l'essor qu'a pris l'informatique à partir de cette époque. Par ailleurs, un ordinateur est inévitablement limité quand il traite des nombres ; dans les langages de l'époque, il fallait tenir compte de ce fait tout simple : on ne pouvait pas compter un nombre de secondes trop important. La date de l'**Epoch** ne pouvait donc pas être trop reculée dans le temps.

Voyons dans un premier temps comment afficher ce fameux nombre de secondes écoulées depuis le 1er janvier 1970 à 00 :00 :00. On utilise la fonction `time` du module.

```
>>> import time
>>> time.time()
1637692462.75781
>>>
```

1. C'est-à-dire les services communs aux différents systèmes d'exploitation.

Cela fait beaucoup ! D'un autre côté, songez quand même que cela représente le nombre de secondes écoulées depuis plus de cinquante ans à présent.

La précision de ce nombre flottant dépend hélas du système d'exploitation utilisé. La partie entière est bien le nombre de secondes et la partie flottante représente une fraction d'une seconde. Cela permet de réaliser, prudemment, des tests de performance, ou de faire remonter les erreurs très précisément (non pas juste à la seconde, mais à la fraction de seconde).

Maintenant, je vous l'accorde, ce nombre n'est pas très compréhensible pour un humain. À l'inverse, pour un ordinateur, c'est l'idéal : les durées calculées en nombre de secondes sont faciles à additionner, soustraire, multiplier… bref, l'ordinateur se débrouille bien mieux avec ce nombre de secondes, ce *timestamp* comme on l'appelle généralement.

Faites un petit test : stockez la valeur renvoyée par `time.time()` dans une première variable puis, quelques secondes plus tard, renouvelez l'opération dans une autre variable. Comparez-les, soustrayez-les, vous verrez que cela se fait tout seul :

```
>>> début = time.time()
>>> # On attend quelques secondes avant de taper la commande
 ↪  suivante
... fin = time.time()
>>> print(début, fin)
1637692498.2990806 1637692503.5138803
>>> début < fin
True
>>> fin - début # Combien de secondes entre début et fin ?
5.214799642562866
>>>
```

Notez bien que la valeur renvoyée par `time.time()` n'est pas un entier mais bien un flottant. Le temps ainsi donné est plus précis qu'à une seconde près. Pour des calculs de performance, ce n'est en général pas cette fonction que l'on utilise, mais c'est bien suffisant la plupart du temps.

La date et l'heure de façon plus présentable

Nous allons découvrir tout au long de ce chapitre des moyens d'afficher nos temps de façon plus élégante et d'obtenir les diverses informations relatives à une date et une heure. Je vous propose ici un premier moyen : une sortie sous la forme d'un objet contenant déjà beaucoup d'informations.

Nous allons utiliser la fonction `localtime` du module `time`.

```
time.localtime()
```

Elle renvoie un objet contenant, dans l'ordre :

1. `tm_year` : l'année sous la forme d'un entier ;
2. `tm_mon` : le numéro du mois (entre 1 et 12) ;

3. `tm_mday` : le numéro du jour du mois (entre 1 et 31, variant d'un mois et d'une année à l'autre) ;
4. `tm_hour` : l'heure du jour (entre 0 et 23) ;
5. `tm_min` : le nombre de minutes (entre 0 et 59) ;
6. `tm_sec` : le nombre de secondes (entre 0 et 61, même si on n'utilisera ici que les valeurs de 0 à 59) ;
7. `tm_wday` : un entier représentant le jour de la semaine (entre 0 et 6, 0 correspond par défaut au lundi) ;
8. `tm_yday` : le jour de l'année, entre 1 et 366 ;
9. `tm_isdst` : un entier représentant le changement d'heure local.

Comme toujours, si vous voulez en apprendre plus, je vous renvoie à la documentation officielle du module `time`.

La fonction `localtime` prend un paramètre optionnel : le *timestamp* tel que nous l'avons découvert précédemment. Si ce paramètre n'est pas précisé, `localtime` utilisera automatiquement `time.time()` et renverra donc la date et l'heure actuelles.

```
>>> time.localtime()
time.struct_time(tm_year=2021, tm_mon=11, tm_mday=23,
 ↪  tm_hour=19, tm_min=36, tm_sec=8, tm_wday=1, tm_yday=327,
 ↪  tm_isdst=0)
>>> time.localtime(début)
time.struct_time(tm_year=2021, tm_mon=11, tm_mday=23,
 ↪  tm_hour=19, tm_min=34, tm_sec=58, tm_wday=1, tm_yday=327,
 ↪  tm_isdst=0)
>>> time.localtime(fin)
time.struct_time(tm_year=2021, tm_mon=11, tm_mday=23,
 ↪  tm_hour=19, tm_min=35, tm_sec=3, tm_wday=1, tm_yday=327,
 ↪  tm_isdst=0)
>>>
```

Pour l'essentiel, c'est assez clair. Malgré tout, la date et l'heure renvoyées ne sont toujours pas des plus lisibles. L'avantage de les avoir sous cette forme, c'est qu'on peut facilement extraire une information si on a juste besoin, par exemple, de l'année et du numéro du jour.

Récupérer un timestamp depuis une date

Je vais passer plus vite sur cette fonction car, selon toute vraisemblance, vous l'utiliserez moins souvent. L'idée est, à partir d'une structure représentant les date et heure telles que renvoyées par `localtime`, de récupérer le *timestamp* correspondant. On utilise pour ce faire la fonction `mktime`.

```
>>> print(début)
1637692498.2990806
```

```
>>> temps = time.localtime(début)
>>> print(temps)
time.struct_time(tm_year=2021, tm_mon=11, tm_mday=23,
↪   tm_hour=19, tm_min=34, tm_sec=58, tm_wday=1, tm_yday=327,
↪   tm_isdst=0)
>>> ts_début = time.mktime(temps)
>>> print(ts_début)
1637692498.0
>>> début == ts_début
False
>>> début - ts_début
0.29908061027526855
>>>
```

Vous constatez que l'on ne retombe pas exactement sur le même *timestamp*. La différence entre les deux est minime et est dû au fait que notre structure de temps ne conserve pas autant de précision dans les secondes.

Mettre en pause l'exécution du programme pendant un temps déterminé

C'est également une fonctionnalité intéressante. La fonction s'appelle `sleep` et elle prend en paramètre un nombre de secondes qui peut être sous la forme d'un entier ou d'un flottant. Pour vous rendre compte de l'effet, testez par vous-mêmes :

```
>>> time.sleep(3.5) # Faire une pause pendant 3,5 secondes
>>>
```

Comme vous pouvez le constatez, Python se met en pause et vous devez attendre 3,5 secondes avant que les trois chevrons s'affichent à nouveau.

Formater un temps

Intéressons-nous maintenant à la fonction `strftime`. Elle formate une date/heure en la représentant dans une chaîne de caractères.

Elle prend deux paramètres :

— la chaîne de formatage (nous verrons plus loin comment la former) ;
— un temps optionnel tel que le renvoie `localtime` ; s'il n'est pas précisé, c'est la date et l'heure courantes qui sont utilisées par défaut.

Pour construire notre chaîne de formatage, nous allons utiliser plusieurs caractères spéciaux. Python va les remplacer par leur valeur (le temps passé en second paramètre ou le temps actuel sinon). Exemple :

```
time.strftime('%Y')
```

Voici comment afficher la date telle qu'on y est habitué en France :

Voici un tableau récapitulatif des quelques symboles utilisables dans cette chaîne.

Symbole	Signification
%A	Nom du jour de la semaine
%B	Nom du mois
%d	Jour du mois (de 01 à 31)
%H	Heure (de 00 à 23)
%M	Minute (entre 00 et 59)
%S	Seconde (de 00 à 59)
%Y	Année

```
1 time.strftime("%A %d %B %Y %H:%M:%S")
```

Mais... c'est en anglais !

Eh oui, mais avec ce que vous savez déjà et ce que vous allez voir par la suite, vous n'aurez pas de difficulté à personnaliser tout cela !

Bien d'autres fonctions

Le module `time` propose bien d'autres fonctions, à consulter dans la documentation du module `time` : `https://docs.python.org/fr/3/library/time.html`

Le module `datetime`

Le module `datetime` propose plusieurs classes pour représenter des dates et heures. Vous n'allez rien découvrir d'absolument spectaculaire dans cette section, mais nous nous avançons petit à petit vers une façon de gérer les dates et heures qui est davantage orientée objet.

Vous trouverez la documentation du module à l'adresse suivante : `https://docs.python.org/fr/3/library/datetime.html`

Représenter une date

Vous le reconnaîtrez probablement avec moi, c'est bien d'avoir accès au temps actuel avec une précision d'une seconde sinon plus... mais parfois, cette précision est inutile. Dans certains cas, on a juste besoin d'une date, c'est-à-dire un jour, un mois et une année. Il est naturellement possible d'extraire cette information de notre *timestamp*. Cependant, le module `datetime` propose une classe `date`, représentant une date, rien qu'une date.

L'objet possède trois attributs :

— `year` : l'année ;
— `month` : le mois ;
— `day` : le jour du mois.

Comment construit-on notre objet `date` ?

Il y a plusieurs façons de procéder. Le constructeur de cette classe prend trois arguments qui sont, dans l'ordre, l'année, le mois et le jour du mois.

```
>>> import datetime
>>> date = datetime.date(2019, 6, 5)
>>> print(date)
2019-06-05
>>>
```

Deux méthodes de classe vous intéresseront :

— `date.today()` : renvoie la date d'aujourd'hui ;
— `date.fromtimestamp(timestamp)` : renvoie la date correspondant au *timestamp* passé en argument.

```
>>> import time
>>> import datetime
>>> aujourdhui = datetime.date.today()
>>> aujourdhui
datetime.date(2021, 11, 23)
>>> datetime.date.fromtimestamp(time.time()) # Équivalent à
 ↪  date.today
datetime.date(2021, 11, 23)
>>>
```

Et, bien entendu, il est facile de manipuler ces dates et de les comparer grâce aux opérateurs usuels ; je vous laisse essayer !

Représenter une heure

C'est moins courant, mais on est quelquefois amené à manipuler une heure, indépendemment de toute date. La classe `time` du module `datetime` est là pour cela.

On construit une heure avec non pas trois mais cinq paramètres, tous optionnels :

— `hour` (0 par défaut) : les heures, valeur comprise entre 0 et 23 ;
— `minute` (0 par défaut) : les minutes, valeur comprise entre 0 et 59 ;
— `second` (0 par défaut) : les secondes, valeur comprise entre 0 et 59 ;
— `microsecond` (0 par défaut) : la précision de l'heure en micro-secondes, entre 0 et 1.000.000 ;

— `tzinfo` (`None` par défaut) : l'information de fuseau horaire (je ne détaillerai pas cette information ici).

Cette classe est moins utilisée que `datetime.date`, mais se révèle utile dans certains cas. Je vous laisse faire quelques tests, n'oubliez pas de vous reporter à la documentation du module `datetime` pour plus d'informations.

Représenter des dates et heures

Et nous y voilà ! Vous n'allez pas être bien surpris par ce que nous allons aborder. Nous avons vu une manière de représenter une date, une manière de représenter une heure, mais on peut naturellement représenter une date et une heure dans le même objet. Ce sera probablement la classe que vous utiliserez le plus souvent ; elle s'appelle `datetime`, comme son module.

Elle prend d'abord les paramètres de `datetime.date` (année, mois, jour) et ensuite ceux de `datetime.time` (heures, minutes, secondes, micro-secondes et fuseau horaire).

Voyons à présent les deux méthodes de classe que vous utiliserez le plus souvent :

— `datetime.now()` : renvoie l'objet `datetime` avec la date et l'heure actuelles ;
— `datetime.fromtimestamp(timestamp)` : renvoie la date et l'heure d'un *time-stamp* précis.

```
>>> import datetime
>>> datetime.datetime.now()
datetime.datetime(2021, 11, 23, 19, 40, 32, 907555)
>>>
```

Il y a bien d'autres choses à découvrir dans ce module `datetime`. Si vous êtes curieux ou si vous avez des besoins plus spécifiques, que je n'aborde pas ici, référez-vous à la documentation officielle du module.

Différence de dates et durées

À première vue, il semble que `datetime.datetime` ne soit qu'une enveloppe un peu plus agréable que celle proposée par `time.struct_time`. Il y a cependant une autre classe du module `datetime` qui s'avère très utile : `timedelta`.

Elle est utilisée pour représenter une durée, une différence entre deux dates et heures. Voyons un exemple :

```
>>> import datetime
>>> d1 = datetime.datetime.now() # première date
>>> # Attendons quelques secondes avant d'exécuter la ligne
 ↪  suivante
... d2 = datetime.datetime.now() # seconde date
>>> # Voyons la différence entre d2 et d1
```

```
... d2 - d1
datetime.timedelta(seconds=14, microseconds=791872)
>>>
```

La classe `datetime.timedelta` mesure la différence, ici, entre deux dates et heures. Vous voyez que la différence est de `14` secondes et `791872` micro-secondes.

On a fait la même chose en comparant les résultats de `time.time()` non ?

En effet. C'est juste un affichage plus agréable. Toutefois, voyons quelque chose de plus intéressant :

```
>>> jour = datetime.timedelta(days=1) # on crée un timedelta
 ↪  représentant une durée d'un jour
>>> d1 + jour # on cherche la date et heure dans un jour
datetime.datetime(2021, 11, 24, 19, 40, 53, 921282)
>>>
```

Il est très utile de créer une durée et, comme vous le voyez, de l'ajouter ou la soustraire de dates et heures. Cela permet de se projeter, dans le passé ou dans le futur. La manipulation du temps en informatique (en particulier celle des fuseaux horaires, heures d'été et heures d'hiver) n'est pas simple, mais vous constatez à nouveau l'intérêt d'encapsuler des données assez abstraites dans de beaux objets ; le code résultant est tellement plus lisible.

En résumé

- Le module `time` fournit, entre autres, la date et l'heure de votre système.
- La fonction `time` du module `time` renvoie le *timestamp* actuel.
- La fonction `localtime` du module `time` renvoie un objet isolant les informations d'un *timestamp* (la date et l'heure).
- Le module `datetime` sert à représenter des dates et heures.
- Les classes `date`, `time` et `datetime` représentent respectivement des dates, des heures, ainsi que des ensembles « date et heure ».

Chapitre 29

Un peu de programmation système

Difficulté :

Dans ce chapitre, nous allons découvrir plusieurs modules et fonctionnalités utiles pour interagir avec le système. Python peut servir à créer bien des choses, des jeux, des interfaces, mais également des scripts système, ce que nous allons découvrir dans ce chapitre.

Les concepts que je vais présenter ici risquent d'être plus familiers aux utilisateurs de Linux. Toutefois, pas de panique si vous êtes sur Windows : je vais prendre le temps de vous expliquer à chaque fois tout le nécessaire.

Les flux standards

Pour commencer, nous allons voir comment accéder aux flux standards (entrée standard et sortie standard) et de quelle façon nous devons les manipuler.

À quoi cela ressemble-t-il ?

Vous vous êtes sûrement habitués, quand vous utilisez la fonction `print`, à ce qu'un message s'affiche sur votre écran. Je pense que cela vous paraît même assez logique à présent.

Néanmoins, comme pour la plupart de nos manipulations en informatique, le mécanisme qui se cache derrière nos fonctions est plus complexe et puissant qu'il y paraît. Sachez que vous pourriez très bien faire en sorte qu'en utilisant `print`, le texte s'écrive dans un fichier plutôt qu'à l'écran.

Quel intérêt ? `print` est fait pour afficher à l'écran non ?

Pas seulement, non. Nous en discuterons un peu plus loin. Pour l'instant, voilà ce que l'on peut dire : quand vous appelez la fonction `print`, si le message s'affiche à l'écran, c'est parce que la sortie standard de votre programme est redirigée vers votre écran.

On distingue trois flux standards :

— L'**entrée standard** : elle est appelée quand vous utilisez `input`. C'est elle qui est utilisée pour demander des informations à l'utilisateur. Par défaut, l'entrée standard est votre clavier.
— La **sortie standard** : comme on l'a vu, c'est elle qui est utilisée pour afficher des messages. Par défaut, elle redirige vers l'écran.
— L'**erreur standard** : elle est notamment utilisée quand Python vous affiche le `traceback` d'une exception. Par défaut, elle redirige également vers votre écran.

Accéder aux flux standards

On accède aux objets représentant ces flux standards grâce au module `sys` qui propose plusieurs fonctions et variables pour interagir avec le système.

```
>>> import sys
>>> sys.stdin # L'entrée standard (standard input)
<_io.TextIOWrapper name='<stdin>' mode='r' encoding='utf-8'>
>>> sys.stdout # La sortie standard (standard output)
<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>
>>> sys.stderr # L'erreur standard (standard error)
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='utf-8'>
>>>
```

Ces objets ne vous rappellent-ils rien ? Vraiment ?

Ils sont de la même classe que les fichiers ouverts grâce à la méthode `pathlib.Path.open`. Et il n'y a aucun hasard derrière cela.

En effet, pour lire ou écrire dans les flux standards, on utilise les méthodes `read` et `write`.

Naturellement, l'entrée standard `stdin` va lire (méthode `read`) et les deux sorties `stdout` et `stderr` servent pour écrire (méthode `write`).

Essayons quelque chose :

```
>>> sys.stdout.write("un test")
un test7
>>>
```

Il n'y a pas trop de surprise, sauf que ce serait mieux avec un saut de ligne à la fin. Là, ce que renvoie la méthode (le nombre de caractères écrits) est affiché juste après notre message.

```
>>> sys.stdout.write("Un test\n")
Un test
8
>>>
```

Modifier les flux standards

Vous pouvez modifier `sys.stdin`, `sys.stdout` et `sys.stderr`.

```
>>> from pathlib import Path
>>> chemin = Path('sortie.txt')
>>> fichier = chemin.open('w')
>>> sys.stdout = fichier
>>> print("Quelque chose...")
>>>
```

Ici, rien ne s'affiche à l'écran. En revanche, si vous ouvrez le fichier `sortie.txt`, vous verrez le message que vous avez passé à `print`.

Je ne trouve pas le fichier `sortie.txt`, où est-il ?

Il doit se trouver dans le répertoire courant de Python. Pour connaître l'emplacement de ce répertoire, utilisez le module `os` et la fonction `getcwd`[1].

1. *Get Current Working Directory*

Une petite subtilité ici : si vous essayez de faire appel à `getcwd` directement, le résultat ne va pas s'afficher à l'écran… mais dans le fichier. Pour rétablir l'ancienne sortie standard, tapez la ligne suivante :

```
sys.stdout = sys.__stdout__
```

Vous pouvez ensuite faire appel à la fonction `getcwd` :

```
import os
os.getcwd()
```

Dans ce répertoire, vous devriez trouver votre fichier `sortie.txt`.

Lorsque vous avez modifié les flux standards et que vous cherchez les objets d'origine, ceux redirigeant vers le clavier (pour l'entrée) et vers l'écran (pour les sorties), vous les retrouvez dans `sys.__stdin__`, `sys.__stdout__` et `sys.__stderr__`.

La documentation de Python nous conseille malgré tout de garder de préférence les objets d'origine sous la main plutôt que d'aller les chercher dans `sys.__stdin__`, `sys.__stdout__` et `sys.__stderr__`.

Voilà qui conclut notre bref aperçu des flux standards. Là encore, si vous ne voyez pas d'application pratique à ce que je viens de vous montrer, cela viendra certainement par la suite.

Les signaux

Les signaux sont un des moyens dont dispose le système pour communiquer avec votre programme. Typiquement, si le système doit arrêter votre programme, il va lui envoyer un signal.

Les signaux peuvent être interceptés dans votre programme. Cela vous permet notamment de déclencher une certaine action si le programme doit se fermer (par exemple, enregistrer des objets dans des fichiers ou fermer les connexions réseau établies avec des clients éventuels).

Les signaux sont également utilisés pour faire communiquer des programmes entre eux. Si votre code est décomposé en plusieurs parties s'exécutant indépendamment les unes des autres, cela sert pour les synchroniser à certains moments clés. Nous ne verrons pas cette dernière fonctionnalité ici car elle mériterait un cours à elle seule tant il y aurait de choses à dire !

Les différents signaux

Le système dispose de plusieurs signaux génériques qu'il envoie aux programmes quand cela est nécessaire. Par exemple, si vous demandez l'arrêt du programme, un signal particulier lui sera envoyé.

Tous les signaux ne se retrouvent pas sur tous les systèmes d'exploitation ; c'est pourquoi je vais surtout m'attacher à un seul : le signal `SIGINT` envoyé à l'arrêt du programme.

Pour plus d'informations, un petit détour par la documentation s'impose, notamment du côté du module `signal` : `https://docs.python.org/fr/3/library/signal.html`

Intercepter un signal

Commencez par importer le module.

```
import signal
```

Le signal qui nous intéresse est donc `SIGINT`.

```
>>> signal.SIGINT
2
>>>
```

Pour l'intercepter, il va falloir créer une fonction, qui prend deux paramètres :

— le signal (plusieurs signaux peuvent être envoyés à la même fonction) ;
— le `frame` qui ne nous intéresse pas ici.

Ensuite, il faudra la connecter avec le signal `SIGINT`.

```
import sys

def fermer_programme(signal, frame):
    """Fonction appelée quand vient l'heure de fermer notre
    ↪  programme"""
    print("C'est l'heure de la fermeture !")
    sys.exit(0)
```

Que signifie la dernière ligne ?

On demande simplement à notre programme Python de se fermer. C'est le comportement standard quand on reçoit un tel signal et notre programme doit bien s'arrêter à un moment ou à un autre.

Pour ce faire, on utilise la fonction `exit` (sortir, en anglais) du module `sys`. Elle prend en paramètre le code de retour du programme.

Pour simplifier, la plupart du temps, si votre programme renvoie `0`, le système comprendra que tout s'est bien passé. Si c'est un entier autre que `0`, le système interprétera cela comme une erreur ayant eu lieu pendant l'exécution de votre programme.

Ici, notre programme s'arrête normalement, on passe donc `0` à `exit`.

Connectons à présent notre fonction au signal `SIGINT`, sans quoi elle ne serait jamais appelée.

On utilise pour cela la fonction `signal`, qui prend en paramètre :

— le signal à intercepter ;
— la fonction que l'on doit connecter à ce signal.

```
signal.signal(signal.SIGINT, fermer_programme)
```

N'écrivez pas les parenthèses à la fin du nom de la fonction. On envoie la référence vers la fonction, on ne l'exécute pas.

Cette ligne va connecter le signal à la fonction. Dès que le système enverra `SIGINT`, `fermer_programme` sera appelée.

Pour vérifier que tout fonctionne bien, lancez une boucle infinie dans votre programme :

```
import signal
import sys

def fermer_programme(signal, frame):
    """Fonction appelée quand vient l'heure de fermer notre
    ↪  programme"""
    print("C'est l'heure de la fermeture !")
    sys.exit(0)

# Connexion du signal à notre fonction
signal.signal(signal.SIGINT, fermer_programme)

# Notre programme...
print("Le programme va boucler...")
while True:
    continue
```

Quand vous lancez ce programme, un message vous informe que ce dernier va boucler… et il continue en effet de tourner tant que son exécution n'est pas interrompue.

Dans la fenêtre du programme, tapez CTRL + C sur Windows ou Linux, Cmd + C sur macOS.

Cette combinaison de touches demande au programme de s'arrêter. Après l'avoir saisie, vous constatez qu'effectivement, votre fonction `fermer_programme` est bien appelée et s'occupe de terminer correctement l'exécution.

Pour aller plus loin sur les signaux, consultez la documentation : `https://docs.python.org/fr/3/library/signal.html`

Interpréter les arguments de la ligne de commande

Python nous offre plusieurs moyens, en fonction de nos besoins, pour interpréter les arguments de la ligne de commande. Succinctement, ces arguments seront des paramètres passés au lancement de votre programme et qui influeront sur son exécution.

Ceux qui travaillent sur Linux n'auront, je pense, aucun mal à me suivre, mais je vais faire une petite présentation pour ceux qui viennent de Windows, afin qu'ils suivent sans difficulté.

Allergiques à la console, passez votre chemin !

Accéder à la console de Windows

Il existe plusieurs moyens d'accéder à la console de Windows. En voici un : ouvrez le menu `Démarrer` et cliquez sur `exécuter...`. Dans la fenêtre qui s'ouvre, tapez `cmd` puis appuyez sur `Entrée`.

Vous devriez vous retrouver dans une fenêtre en console, vous donnant plusieurs informations propres au système.

```
Microsoft Windows [Version 10.0.17134.765]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Vincent>
```

Accéder aux arguments de la ligne de commande

Nous allons à nouveau appeler notre module `sys`. Cette fois, nous nous intéressons à sa variable `argv`.

Créez un nouveau fichier Python.

Placez-y le code suivant :

```
import sys
print(sys.argv)
```

`sys.argv` contient une liste des arguments que vous passez en ligne de commande, au moment de lancer le programme. Essayez donc d'appeler votre programme depuis la ligne de commande en lui passant des arguments.

Sur Windows :

```
C:\Users\Vincent>python test_console.py
['test_console.py']
C:\Users\Vincent>python test_console.py arguments
['test_console.py', 'arguments']
C:\Users\Vincent>python test_console.py argument1 argument2
 ↪  argument3
['test_console.py', 'argument1', 'argument2', 'argument3']
C:\Users\Vincent>
```

Comme vous le constatez, le premier élément de `sys.argv` contient le nom du programme, de la façon dont vous l'avez appelé. Le reste de la liste contient vos arguments (s'il y en a).

Note : si le nom d'un argument contient des espaces, encadrez-le de guillemets :

```
C:\Users\Vincent>python test_console.py "un argument avec des espaces"
['test_console.py', 'un argument avec des espaces']
C:\Users\Vincent>
```

Interpréter les arguments de la ligne de commande

Accéder aux arguments, c'est bien, mais les interpréter c'est mieux.

Des actions simples

Parfois, votre programme devra déclencher plusieurs actions en fonction du premier paramètre fourni. Par exemple, en premier argument, vous pourriez préciser l'une des valeurs suivantes : `start` pour démarrer une opération, `stop` pour l'arrêter, `restart` pour la redémarrer, `status` pour connaître son état… bref, les utilisateurs de Linux ont sûrement bien plus d'exemples à l'esprit.

Dans ce cas de figure, il n'est pas vraiment nécessaire d'interpréter les arguments de la ligne de commande, comme on va le voir. Notre programme Python ressemblerait simplement à ce qui suit :

```
import sys

match sys.argv[1:]: # On ignore le premier argument.
    case ["start"]:
        print("On démarre l'opération")
    case ["stop"]:
        print("On arrête l'opération")
    case ["restart"]:
        print("On redémarre l'opération")
    case ["status"]:
        print("On affiche l'état (démarré ou arrêté ?) de l'opération")
    case [autre_chose]:
        print(f"Je ne connais pas cette action: {autre_chose}")
    case _: # Rien ne correspond.
        print("Précisez une action en paramètre")
        sys.exit(1)
```

Il est nécessaire de passer par la ligne de commande pour tester ce programme.

Des options plus complexes

La ligne de commande permet également de transmettre des arguments plus complexes comme des options. La plupart du temps, nos options sont sous la forme : `-option_courte` (une seule lettre) ou `--option_longue`, suivie d'un argument ou non.

Souvent, une option courte est accessible aussi depuis une option longue.

Ici, mon exemple va être tiré de Linux. La commande `ls` affiche le contenu d'un répertoire. Elle accepte en paramètres plusieurs options qui influent sur ce qu'elle va afficher au final.

Par exemple, pour afficher tous les fichiers (cachés ou non) du répertoire, on utilise l'option courte `a`.

```
$ ls -a
.  ..  fichier1.txt  .fichier_cache.txt  image.png
$
```

Cette option courte est accessible depuis une option longue, `all`. Vous arrivez donc au même résultat en tapant :

```
$ ls --all
.  ..  fichier1.txt  .fichier_cache.txt  image.png
$
```

Pour récapituler, nos options courtes sont précédées d'un seul tiret et composées d'une seule lettre. Les options longues sont précédées de deux tirets et composées de plusieurs lettres.

Certaines options attendent un argument, à préciser juste après l'option.

Par exemple (toujours sur Linux), pour afficher les X premières lignes d'un fichier, vous appelez la commande `head -n X`.

```
$ head -n 5 fichier.txt
ligne 1
ligne 2
ligne 3
ligne 4
ligne 5
$
```

Dans ce cas, l'option `-n` attend un argument qui est le nombre de lignes à afficher.

Interpréter ces options grâce à Python

Nous allons nous intéresser au module `argparse` qui est utile, justement, pour interpréter les arguments de la ligne de commande selon un certain schéma. La base du code est la suivante :

```
import argparse
parser = argparse.ArgumentParser()
parser.parse_args()
```

1. D'abord, on importe le module `argparse`.
2. On crée ensuite un `argparse.ArgumentParser` qui va être utile pour configurer nos options à interpréter.
3. Enfin, on appelle la méthode `parse_args()` sur notre parseur ; elle retourne les arguments interprétés. Nous allons voir comment préciser des options dans notre parseur, pour rendre les choses plus intéressantes. Notez que, par défaut, l'interprétation des arguments se fait depuis `sys.argv[1:]` (c'est-à-dire la liste des arguments sans le nom du script).

En fait, notre parseur n'est pas tout à fait vide. Exécutez le script précédent avec l'option `-help` :

```
> python code.py --help
usage: code.py [-h]

optional arguments:
  -h, --help  show this help message and exit

>
```

Cela vous donne un petit aperçu de la façon d'utiliser notre programme. L'aide (option `-h` ou `-help`) est générée par défaut. Et si vous n'utilisez pas le script convenablement, une erreur vous est retournée :

```
> python code.py --inexistante
usage: code.py [-h]
code.py: error: unrecognized arguments: --inexistante

>
```

Ici, nous avons simplement spécifié une option qui n'a pas été définie. Essayons d'en définir une :

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", help="le nombre à élever au carré")
parser.parse_args()
```

Nous avons ajouté une option grâce à la méthode `add_argument()`. Elle prend plusieurs paramètres (de nombreux paramètres optionnels, en fait) mais nous n'en avons précisé que deux ici : l'option et le message d'aide lié.

Demandez l'aide du script :

```
> python code.py --help
usage: code.py [-h] x
```

```
positional arguments:
  x           le nombre à élever au carré

optional arguments:
  -h, --help  show this help message and exit

>
```

Nous devons maintenant préciser un nombre `x` en paramètre. Essayons de récupérer sa valeur :

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", help="le nombre à élever au carré")
args = parser.parse_args()
print("Vous avez précisé X =", args.x)
```

Pour connaître les options (ce que nous voudrons faire la plupart du temps), on récupère le retour de la méthode `parse_args()`. Elle retourne un objet `namespace` avec nos options en attribut. Accéder à `args.x` retourne donc le nombre précisé par l'utilisateur :

```
> python code.py 5
Vous avez précisé X = 5

>
```

Dans ce contexte, on veut un nombre... mais l'utilisateur peut entrer n'importe quoi. Modifions donc notre méthode `add_argument` pour que l'utilisateur ne puisse saisir que des nombres :

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="le nombre à élever au
↪  carré")
args = parser.parse_args()
x = args.x
retour = x ** 2
print(retour)
```

Comme vous le voyez, la méthode `add_argument` est précisée ici avec un nouvel argument : `type`. On lui précise `int`, ce qui veut dire que l'on attend un nombre (l'entrée de l'utilisateur sera automatiquement convertie).

Vous constatez aussi que notre programme fait maintenant quelque chose de concret :

```
> python code.py 5
25

> python code.py -8
```

```
64

> python code.py test
usage: code.py [-h] x
code.py: error: argument x: invalid int value: 'test'

>
```

Vous observez que la conversion se passe bien, jusqu'au message d'erreur affiché si l'utilisateur n'entre pas un nombre.

Jusqu'ici, nous avons créé des « positional arguments », qui doivent être précisés sans option. Voyons comment ajouter des options facultatives :

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="le nombre à élever au carré")
parser.add_argument("-v", "--verbose", action="store_true",
        help="augmente le niveau de détail")
args = parser.parse_args()

x = args.x
retour = x ** 2
if args.verbose:
    print(f"{x} ^ 2 = {retour}")
else:
    print(retour)
```

Nous avons ajouté une nouvelle option : `-v` ou `-verbose`. Le nom commençant par un tiret, `argparse` suppose qu'il s'agit d'une option facultative, même si cela peut être modifié.

Notez que l'on appelle la méthode `add_argument` avec l'argument `action`. L'action précisée, « `store_true` », sert à convertir l'option précisée en booléen :

— Si l'option est précisée, alors `args.verbose` vaudra `True`.
— Si l'option n'est pas précisée, alors `args.verbose` vaudra `False`.

Le résultat affiché est différent en fonction de l'option ; si elle est précisée, le message de retour est un peu plus détaillé :

```
> python code.py -h
usage: code.py [-h] [-v] x

positional arguments:
  x              le nombre à élever au carré

optional arguments:
  -h, --help     show this help message and exit
  -v, --verbose  augmente le niveau de détail
```

```
> python code.py 5
25

> python code.py 5 --verbose
5 ** 2 = 25

> python code.py -v 5
5 ** 2 = 25

>
```

Vous voyez que le retour est différent en fonction du niveau de détail. Notez aussi que le message d'aide intègre bien notre nouvelle option. C'est l'une des raisons (il y en a beaucoup) qui rendent l'utilisation de `argparse` si pratique.

Nous n'avons vu que le tout début des fonctionnalités de ce module. Si vous voulez en apprendre plus, je vous conseille un tutoriel sur `argparse` (français), qui présente les fonctionnalités les plus couramment utilisées du module : `https://docs.python.org/fr/3/howto/argparse.html`

Et bien évidemment, tout est décrit en détail dans la documentation officielle en français (il est préférable de lire le cours avant) : `https://docs.python.org/fr/3/library/argparse.html`

Exécuter une commande système depuis Python

Nous allons ici nous intéresser à la façon d'exécuter des commandes depuis Python. Nous étudierons deux moyens, mais il en existe d'autres.

Ceux que je vais présenter ont l'avantage de fonctionner sur Windows.

La fonction `system`

Vous vous souvenez peut-être de cette fonction du module `os`. Elle prend en paramètre une commande à exécuter, en affiche le résultat et renvoie son code de retour.

```
os.system("ls") # Sur Linux
os.system("dir") # Sur Windows
```

Vous pouvez capturer le code de retour de la commande mais pas ce qu'elle affiche.

En outre, la fonction `system` exécute un environnement particulier rien que pour votre commande. Cela veut dire, entre autres, que `system` retournera tout de suite même si la commande s'exécute toujours.

En gros, si vous écrivez `os.system(''sleep~5'')`, le programme ne s'arrêtera pas pendant cinq secondes.

La fonction **popen**

Cette fonction se trouve également dans le module `os` et prend une commande en paramètre.

Toutefois, au lieu de renvoyer le code de retour de la commande, elle renvoie un objet, un `pipe`[2] contenant son retour.

Voici un exemple sur Linux :

```
>>> import os
>>> cmd = os.popen("ls")
>>> cmd
<os._wrap_close object at 0x7f81d16554d0>
>>> cmd.read()
'fichier1.txt\nimage.png\n'
>>>
```

Le fait de lire le *pipe* bloque le programme jusqu'à ce que la commande ait fini de s'exécuter.

Je vous ai dit qu'il existait d'autres moyens. Et au-delà de cela, vous avez beaucoup d'autres choses intéressantes dans le module `os` pour interagir avec le système…

En résumé

- Le module `sys` propose trois objets pour accéder aux flux standards : `stdin`, `stdout` et `stderr`.
- Le module `signal` sert à intercepter les signaux envoyés à notre programme.
- Le module `argparse` interprète les arguments passés en console à notre programme.
- Enfin, le module `os` possède, entre autres, plusieurs fonctions pour envoyer des commandes au système.

2. Mot anglais pour un « tuyau ».

Chapitre 30

Un peu de mathématiques

Difficulté :

Dans ce chapitre, nous allons découvrir trois modules. Je vous ai déjà fait utiliser certains d'entre eux ; ce sera ici l'occasion de revenir dessus plus en détail.

— Le module `math` propose un bon nombre de fonctions mathématiques.
— Le module `fractions`, dont nous allons surtout voir la classe éponyme, permet... vous l'avez deviné ? De modéliser des fractions.
— Et enfin, nous découvrirons en détail le module `random` que vous connaissez de par nos TP.

Pour commencer, le module `math`

Le module `math`, vous le connaissez déjà : nous l'avons utilisé comme premier exemple de module créé par Python. Vous avez peut-être eu la curiosité de regarder son aide pour voir quelles fonctions y étaient définies. Dans tous les cas, je fais un petit point sur certaines de ces fonctions.

Je ne vais pas m'attarder très longtemps sur ce module en particulier car il est plus vraisemblable que vous cherchiez une fonction précise et que la documentation sera, dans ce cas, plus accessible et explicite.

Fonctions usuelles

Vous vous souvenez des opérateurs `+`, `-`, `*`, `/` et `%`. Le module `math` propose plusieurs fonctions pour les compléter :

```
>>> math.pow(5, 2) # 5 au carré
25.0
>>> 5 ** 2 # Pratiquement identique à pow(5, 2)
25
>>> math.sqrt(25) # Racine carrée de 25 (square root)
5.0
>>> math.exp(5) # Exponentielle
148.4131591025766
>>> math.fabs(-3) # Valeur absolue
3.0
>>>
```

Il y a bel et bien une différence entre l'opérateur `**` et la fonction `math.pow` : cette dernière renvoie toujours un flottant alors que l'opérateur renvoie un entier quand cela est possible.

Un peu de trigonométrie

Avant de présenter les fonctions usuelles en trigonométrie, j'attire votre attention sur le fait que les angles, en Python, sont donnés et renvoyés en radians (rad).

Pour rappel :

$$1 \text{ rad} = 57{,}29 \text{ degrés}$$

Cela étant dit, il existe déjà dans le module `math` les fonctions qui convertissent simplement nos angles.

```
math.degrees(angle_en_radians) # Convertit en degrés
math.radians(angle_en_degrés) # Convertit en radians
```

Les fonctions trigonométriques se nomment, sans surprise :

— `cos` : cosinus ;
— `sin` : sinus ;
— `tan` : tangente ;
— `acos` : arc cosinus ;
— `asin` : arc sinus ;
— `atan` : arc tangente.

Arrondir un nombre

Le module `math` nous propose plusieurs fonctions pour arrondir un nombre selon différents critères :

```
>>> math.ceil(2.3) # Renvoie le plus petit entier >= 2.3
3
>>> math.floor(5.8) # Renvoie le plus grand entier <= 5.8
5
>>> math.trunc(9.5) # Tronque 9.5
9
>>>
```

Arrondir au plus proche ne se fait pas grâce à une fonction du module `math`, mais à une qui est toujours présente dans Python, appelée `round`. Elle prend au moins un paramètre : le nombre à arrondir :

```
>>> round(3.1)
3
>>> round(4.8)
5
>>> round(-5.5)
-6
>>>
```

On peut également préciser en second paramètre un nombre indiquant la précision. S'il est positif, il correspond au nombre de chiffres après la virgule qui seront retournés. S'il est négatif, il donne la précision avant la virgule :

```
>>> round(25.25589, 2) # Arrondit à 2 chiffres après la
 ↪  virgule
25.26
>>> round(3.3333333334, 1)
3.3
>>> round(192, -1) # Arrondit à la dizaine (un chiffre avant
 ↪  la virgule)
190
>>> round(198, -1)
200
>>>
```

Quant aux constantes du module, elles ne sont pas nombreuses : `math.pi` naturellement, ainsi que `math.e`.

Si vous cherchez un renseignement précis, consultez la documentation officielle du module : `https://docs.python.org/fr/3/library/math.html`

Des fractions avec le module `fractions`

Ce module propose, entre autres, de manipuler des objets modélisant des fractions. C'est la classe `Fraction` du module qui nous intéresse :

```
from fractions import Fraction
```

Créer une fraction

Le constructeur de la classe `Fraction` accepte plusieurs types de paramètres :

- Deux entiers, le numérateur et le dénominateur (par défaut le numérateur vaut `0` et le dénominateur `1`). Si le dénominateur est `0`, une exception `ZeroDivisionError` est levée.
- Une autre fraction.
- Une chaîne sous la forme `'numérateur/dénominateur'`.

```
>>> un_demi = Fraction(1, 2)
>>> un_demi
Fraction(1, 2)
>>> un_quart = Fraction('1/4')
>>> un_quart
Fraction(1, 4)
>>> autre_fraction = Fraction(-5, 30)
>>> autre_fraction
Fraction(-1, 6)
>>>
```

Ne peut-on pas créer des fractions depuis un flottant ?

Si, mais pas dans le constructeur. Pour créer une fraction depuis un flottant, on utilise la méthode de classe `from_float` :

```
>>> Fraction.from_float(0.5)
Fraction(1, 2)
>>>
```

Et pour retomber sur un flottant, rien n'est plus simple :

```
>>> float(un_quart)
0.25
>>>
```

Manipuler les fractions

Maintenant, quel est l'intérêt d'écrire nos nombres sous cette forme ? C'est surtout pour la précision des calculs. Les fractions que nous venons de voir acceptent naturellement les opérateurs usuels :

```
>>> un_dixième = Fraction(1, 10)
>>> un_dixième + un_dixième + un_dixième
Fraction(3, 10)
>>>
```

Alors que la forme courante conduit à des erreurs de précision :

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
>>>
```

Bien sûr, la différence n'est pas énorme, mais elle est là. Tout dépend de vos besoins en termes de précision.

D'autres calculs ?

```
>>> un_dixième * un_quart
Fraction(1, 40)
>>> un_dixième + 5
Fraction(51, 10)
>>> un_demi / un_quart
Fraction(2, 1)
>>> un_quart / un_demi
Fraction(1, 2)
>>>
```

Cette petite démonstration suffira pour démarrer. Si ce module vous intéresse, rendez-vous sur sa documentation officielle : `https://docs.python.org/fr/3/library/fractions.html`

Du pseudo-aléatoire avec `random`

Vous avez également utilisé le module `random` pendant nos TP. Nous allons en présenter quelques fonctions.

Du pseudo-aléatoire

L'ordinateur est une machine puissante, qui n'a aucune difficulté à additionner, soustraire, multiplier ou diviser des nombres et même réaliser des calculs bien plus complexes. Toutefois, choisir un nombre au hasard est pour lui bien plus compliqué qu'il n'y paraît.

Ce qu'il faut bien comprendre, c'est que derrière notre appel à `random.randint` par exemple, Python va lancer un véritable calcul pour trouver un nombre. Ce dernier n'est donc pas réellement aléatoire puisqu'un calcul identique, effectué dans les mêmes conditions, donnera la même valeur. Cependant, les algorithmes mis en place pour générer de l'aléatoire sont maintenant suffisamment complexes pour que les nombres obtenus ressemblent bien à une série aléatoire. Souvenez-vous toutefois que, pour un ordinateur, le véritable hasard ne peut pas exister.

La fonction `random`

Cette fonction ne servira peut-être pas souvent de manière directe, mais elle est implicitement utilisée par le module quand on fait appel à `randint` ou `choice`.

Elle génère un nombre pseudo-aléatoire compris entre 0 et 1. Ce sera donc naturellement un flottant :

```
>>> import random
>>> random.random()
0.9565461152605507
>>>
```

`randrange` et `randint`

La fonction `randrange` prend trois paramètres :

- la marge inférieure de l'intervalle ;
- la marge supérieure de l'intervalle ;
- l'écart entre chaque valeur de l'intervalle (`1` par défaut).

Que représente le dernier paramètre ?

Prenons un exemple, ce sera plus simple :

```
random.randrange(5, 10, 2)
```

Cette instruction va chercher à générer un nombre aléatoire entre `5` inclus et `10` non inclus, avec un écart de `2` entre chaque valeur possible. Elle va donc chercher dans la liste des valeurs `[5, 7, 9]`.

Si vous ne précisez pas de troisième paramètre, il vaudra `1` par défaut (c'est le comportement attendu la plupart du temps).

La fonction `randint` prend deux paramètres :

— là encore, la marge inférieure de l'intervalle ;
— la marge supérieure de l'intervalle, cette fois incluse.

Pour tirer au hasard un nombre entre `1` et `6`, il est donc plus intuitif d'écrire ce qui suit :

```
random.randint(1, 6)
```

Pour lancer plusieurs dés d'un seul coup, que penseriez-vous d'utiliser nos magnifiques compréhensions de liste ?

```
>>> [random.randint(1, 6) for étape in range(10)] # Génère une liste de 10 nombres pseudo-aléatoires entre 1 et 6 inclus
[2, 6, 4, 2, 4, 4, 5, 5, 1, 1]
>>>
```

Opérations sur des séquences

Une première fonction, `choice`, renvoie au hasard un élément d'une séquence passée en paramètre :

```
>>> random.choice(['a', 'b', 'k', 'p', 'i', 'w', 'z'])
'k'
>>>
```

La fonction `shuffle`, quant à elle, prend en paramètre une séquence et la mélange ; elle modifie donc la séquence qu'on lui passe et ne renvoie rien :

```
>>> liste = ['a', 'b', 'k', 'p', 'i', 'w', 'z']
>>> random.shuffle(liste)
>>> liste
['p', 'k', 'w', 'z', 'i', 'b', 'a']
>>>
```

Si vous voulez aller plus loin, consultez la documentation officielle de Python : `https://docs.python.org/fr/3/library/random.html`

Puis-je utiliser `random` pour générer un mot de passe ?

Surtout pas ! Enfin... techniquement, oui, mais surtout, surtout pas.

Comme je l'ai dit, le module `random` génère du **pseudo**-aléatoire. Si on l'utilise pour générer un mot de passe, il serait théoriquement possible à une personne malveillante d'utiliser le même calcul pour recréer votre mot de passe généré. Cela étant, il existe un autre module, `secrets`, que l'on peut utiliser car il comprend des algorithmes sécurisés au possible. Si ce sujet vous intéresse, je vous renvoie à la documentation officielle. Nous aurons l'occasion de reparler de mot de passe au prochain chapitre : `https://docs.python.org/fr/3/library/secrets.html`

En résumé

— Le module `math` propose plusieurs fonctions et constantes mathématiques usuelles.
— Le module `fractions` fournit le nécessaire pour manipuler des fractions, parfois utiles pour la précision des calculs.
— Le module `random` sert à générer des nombres pseudo-aléatoires.

Chapitre 31

Gestion des mots de passe

Difficulté :

Dans ce chapitre, nous allons nous intéresser aux mots de passe et à la façon de les gérer en Python, c'est-à-dire de les obtenir et de les protéger.

Nous allons découvrir trois modules : d'abord `getpass`, pour demander un mot de passe à l'utilisateur, puis `hashlib`, pour le chiffrer, et enfin `secrets`, pour générer un mot de passe aléatoire.

Obtenir un mot de passe saisi par l'utilisateur

Nous nous sommes déjà largement servi d'une forme d'interaction avec l'utilisateur : il s'agit naturellement de la fonction `input`.

Cependant, elle n'est pas très discrète. Si vous saisissez un mot de passe confidentiel, il apparaît de manière visible à l'écran, ce qui n'est pas souhaitable.

C'est ici qu'intervient le module `getpass`. La fonction qui nous intéresse porte le même nom que le module. Comme `input`, elle attend et capture une saisie de l'utilisateur, mais elle ne l'affiche pas.

```
>>> from getpass import getpass
>>> mot_de_passe = getpass()
Password:
>>> mot_de_passe
'un mot de passe'
>>>
```

Comme vous le voyez... bah justement on ne voit rien ! Le mot de passe que l'on tape est invisible. Vous appuyez sur les touches de votre clavier mais rien ne s'affiche. Cependant, vous écrivez bel et bien et, quand vous appuyez sur `Entrée`, la fonction `getpass` renvoie ce que vous avez saisi.

Ici, on le stocke dans la variable `mot_de_passe`. C'est plus discret qu'`input`, reconnaissez-le !

Bon, il reste un détail, mineur certes, mais un détail quand même : le `prompt` par défaut, c'est-à-dire le message qui vous invite à saisir votre mot de passe, est en anglais. Heureusement, il s'agit tout simplement d'un paramètre facultatif de la fonction :

```
>>> mot_de_passe = getpass("Tapez votre mot de passe : ")
Tapez votre mot de passe :
>>>
```

C'est mieux.

Bien entendu, tous les mots de passe que vous collecterez ne viendront pas forcément d'une saisie directe d'un utilisateur. Néanmoins, dans ce cas précis, la fonction `getpass` est bien utile. À la fin de ce chapitre, nous verrons une utilisation complète de cette fonction, incluant réception et chiffrement de notre mot de passe en prime, deux en un.

Chiffrer un mot de passe

Cette fois-ci, nous allons nous intéresser au module `hashlib`. Au préalable toutefois, quelques explications s'imposent.

Chiffrer un mot de passe ?

La première question qu'on pourrait légitimement se poser est « pourquoi protéger un mot de passe ? ». Je suis sûr que vous trouverez par vous-mêmes de nombreuses réponses : il est un peu trop facile de récupérer un mot de passe s'il est stocké ou transmis en clair. Et cela donne accès à beaucoup de choses. Ainsi, généralement, quand on a besoin de stocker un mot de passe ou de le transmettre, on le chiffre.

Maintenant, qu'est-ce que le chiffrement ? A priori, l'idée est assez simple : en partant d'un mot de passe, n'importe lequel, on arrive à une seconde chaîne de caractères, complètement incompréhensible.

Quel en est l'intérêt ?

Eh bien, si vous voyez une chaîne de caractères comme

`b47ea832576a75814e13351dcc97eaa985b9c6b7`,

vous ne pouvez pas vraiment deviner le mot de passe qui se cache derrière.

Et l'ordinateur ne peut pas le déchiffrer si facilement que cela non plus. Bien sûr, il existe des méthodes pour déchiffrer un mot de passe mais nous ne les verrons certainement pas ici. Nous, ce que nous voulons savoir, c'est comment protéger nos mots de passe, pas comment déchiffrer ceux des autres !

Comment fonctionne le chiffrement ?

Grave question. D'abord, il existe plusieurs techniques ou **algorithmes** de chiffrement. Chiffrer un mot de passe avec un certain algorithme ne donne pas le même résultat qu'avec un autre algorithme.

Ensuite, l'algorithme, quel qu'il soit, est assez complexe. Je serais bien incapable de vous expliquer en détail comment cela marche ; on fait appel à beaucoup de concepts mathématiques relativement poussés.

Toutefois, si vous voulez faire un exercice, je vous propose quelque chose d'amusant qui vous donnera une meilleure idée du chiffrement.

Commencez par numéroter toutes les lettres de l'alphabet (de **a** à **z**) de **1** à **26**. Représentez l'ensemble des valeurs dans un tableau, ce sera plus simple.

A (1)	B (2)	C (3)	D (4)	E (5)	F (6)	
G (7)	H (8)	I (9)	J (10)	K (11)	L (12)	M (13)
N (14)	O (15)	P (16)	Q (17)	R (18)	S (19)	
T (20)	U (21)	V (22)	W (23)	X (24)	Y (25)	Z (26)

Maintenant, supposons que nous allons chercher à chiffrer des prénoms. Pour cela, nous allons baser notre exemple sur un calcul simple : dans le tableau, prenez la valeur numérique de chaque lettre constituant le prénom et additionnez l'ensemble des valeurs obtenues.

Par exemple, partons du prénom *Eric*. Quatre lettres, cela ira vite. Oubliez les accents, les majuscules et minuscules. On a un **E (5)**, un **R (18)**, un **I (9)** et un **C (3)**. En ajoutant les valeurs de chaque lettre, on obtient donc **5**+**18**+**9**+**3**. Conclusion : en chiffrant *Eric* grâce à notre algorithme, on obtient le nombre **35**.

C'est l'idée sous-jacente du chiffrement même si, en réalité, les choses sont beaucoup plus complexes. En outre, au lieu d'avoir un nombre en sortie, on a généralement plutôt une chaîne de caractères.

Prenez cet exemple pour vous amuser, si vous voulez. Appliquez notre algorithme à plusieurs prénoms. Si vous vous sentez d'attaque, essayez de coder une fonction Python qui prenne en paramètre notre chaîne et la renvoie chiffrée ; ce n'est pas bien difficile.

Vous pouvez maintenant vous rendre compte que derrière un nombre tel que **35**, il est plutôt difficile de deviner que se cache le prénom *Eric* !

Maintenant, si vous faites le test sur les prénoms *Louis* et *Jacques*, vous vous rendrez compte… qu'ils produisent le même résultat, **76** :

— Louis = 12 + 15 + 21 + 9 + 19 = 76
— Jacques = 10 + 1 + 3 + 17 + 21 + 5 + 19 = 76

C'est ce qu'on appelle une **collision** : en prenant deux chaînes différentes, on obtient le même chiffrement au final.

Les algorithmes que nous allons découvrir dans le module `hashlib` essayent de minimiser, autant que possible, les collisions. À l'inverse, celui que nous venons juste de voir en est plein : il suffit de changer de place les lettres de notre prénom et nous retombons sur le même nombre, après tout.

Chiffrer un mot de passe

Commençons par importer le module `hashlib` :

```
1 import hashlib
```

On va maintenant choisir un algorithme. Pour nous aider dans notre choix, le module `hashlib` nous propose deux listes :

— `algorithms_guaranteed` : les algorithmes garantis par Python, les mêmes d'une plate-forme à l'autre. Si vous voulez que vos programmes soient portables, il est préférable d'utiliser l'un d'entre eux :

```
>>> hashlib.algorithms_guaranteed
{'sha384', 'sha3_224', 'md5', 'sha256', 'sha3_256',
 ↪  'shake_128', 'sha224',
  'sha3_512', 'blake2s', 'sha512', 'blake2b',
 ↪  'shake_256', 'sha3_384', 'sha1'}
>>>
```

— `algorithms_available` : les algorithmes disponibles sur votre plate-forme. Tous les algorithmes garantis s'y trouvent, plus quelques autres propres à votre système.

Dans ce chapitre, nous allons nous intéresser à `sha256`.

Pour commencer, nous allons créer notre objet `SHA256`. On va utiliser le constructeur `sha256` du module `hashlib`. Il prend en paramètre une chaîne, mais une chaîne de **bytes** (octets).

Pour obtenir une chaîne de **bytes** depuis une chaîne `str`, on peut utiliser la méthode `encode`. Je ne vais pas rentrer dans le détail des encodages ici. Pour écrire directement une chaîne **bytes** sans passer par une chaîne `str`, vous avez une autre possibilité consistant à mettre un `b` minuscule avant l'ouverture de votre chaîne :

```
>>> b'test'
b'test'
>>>
```

Générons notre mot de passe :

```
>>> mot_de_passe = hashlib.sha256(b"mot de passe")
>>> mot_de_passe
<sha256 _hashlib.HASH object @ 0x00DF1F80>
>>>
```

Pour obtenir le chiffrement associé à cet objet, on a deux possibilités :

— la méthode `digest`, qui renvoie un type **bytes** contenant notre mot de passe chiffré ;
— la méthode `hexdigest`, qui renvoie une chaîne `str` contenant une suite de symboles hexadécimaux (de 0 à 9 et de A à F).

C'est cette dernière méthode que je vais montrer ici, parce qu'elle est préférable pour un stockage en fichier si les fichiers doivent transiter d'une plate-forme à l'autre.

```
>>> mot_de_passe.hexdigest()
'b9e50e0e8b504aa57a1bb6711ee832ee4ce9c641a1618b91833582382
 ↪  c709023'
>>>
```

Et pour déchiffrer ce mot de passe ?

On ne le déchiffre pas. Si vous voulez vérifier le mot de passe saisi par l'utilisateur, chiffrez-le et comparez le résultat au mot de passe chiffré que vous aurez conservé :

```
import hashlib
from getpass import getpass

chaîne_mot_de_passe = b"azerty"
mot_de_passe_chiffré =
 ↪  hashlib.sha256(chaîne_mot_de_passe).hexdigest()

verrouillé = True
while verrouillé:
    entré = getpass("Tapez le mot de passe : ") # azerty
    # On encode la saisie pour avoir un type bytes
    entré = entré.encode()
    entré_chiffré = hashlib.sha256(entré).hexdigest()
    if entré_chiffré == mot_de_passe_chiffré:
        verrouillé = False
    else:
        print("Mot de passe incorrect")

print("Mot de passe accepté...")
```

Ne peut-on conserver le mot de passe de façon sécurisée pour le déchiffrer plus tard ?

Dans certains cas, on a besoin de protéger certaines informations... mais on doit se garder la possibilité de les éditer. `hashlib` (ou `hmac`) sont pratiques, mais en l'occurrence le plus simple est de faire appel à une bibliothèque externe, comme `cryptography`, qu'il vous faudra installer.

Un chapitre à la fin de ce livre est consacré à l'installation de dépendances en Python. Si ce sujet vous intéresse, n'hésitez pas à regarder la documentation du module `cryptography`.

Générer des mots de passe aléatoires

En explorant le module `random`, tu as bien dit qu'un ordinateur ne pouvait pas générer de véritable aléatoire ? C'est dangereux pour les mots de passe, non ?

En effet. Cependant, le module `secrets`, apparu avec Python 3.6, utilise des algorithmes de génération bien plus poussés et solides que le pseudo-aléatoire offert par `random`. La génération offerte par `secrets` utilise les algorithmes les plus sécurisés et solides offerts par votre système d'exploitation : vous pouvez toujours vous en prendre à lui si on vous vole votre mot de passe.

Le module `secrets` est conseillé, entre autres, pour générer des mots de passe aléatoires. Je vais surtout vous montrer deux de ses fonctions :

— `choice` : retourne un élément aléatoire d'une séquence ;
— `token_hex` : génère une chaîne aléatoire de nombres hexadécimaux.

La première fonction ne semble pas très intéressante à première vue :

```
>>> import secrets
>>> secrets.choice("abcdefghij")
'd'
>>> secrets.choice("abcdefghij")
'f'
>>>
```

Pour l'instant, rien n'est vraiment nouveau, en apparence tout au moins. On utilise `secrets.choice` à la place de `random.choice`, mais sinon, le résultat semble assez proche.

Et pourtant, avec cette fonction, on peut générer un mot de passe de 8 lettres au minimum, comportant les lettres de l'alphabet (majuscules et minuscules) et les chiffres. Sauriez-vous comment ? C'est un exercice intéressant, je vous le recommande :

```
>>> mot_de_passe = ""
>>> alphabet = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP
 ↪  QRSTUVWXYZ0123456789"
>>> for i in range(8):
...     mot_de_passe += secrets.choice(alphabet)
...
>>> print(mot_de_passe)
IEdqJtwq
>>>
```

En utilisant les compréhensions de liste, on peut même écrire plus simplement :

```
>>> alphabet = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP
 ↪  QRSTUVWXYZ0123456789"
>>> caractères = [secrets.choice(alphabet) for i in range(8)]
```

```
>>> caractères
['A', 'J', 'T', 'a', 'k', 'y', 'p', 'L']
>>> mot_de_passe = "".join(caractères) # coller tous les
 ↪  caractères et les grouper dans une chaîne
>>> mot_de_passe
'AJTakypL'
>>>
```

L'utilisation de `str.join` dans une chaîne vide a pu vous surprendre. En fait il s'agit d'une manière simple de regrouper notre liste en une chaîne, sans aucun séparateur entre chaque caractère.

Notre mot de passe est composé de huit lettres, ni plus, ni moins. On pourrait vouloir avoir une taille aléatoire pour des raisons de sécurité. Certes, on pourrait utiliser `random.randint`, mais autant continuer avec `secrets`. En théorie, c'est peu utile : même si une personne malveillante trouve la taille de votre mot de passe, ça ne l'avance pas forcément beaucoup. Cependant, autant jouer la prudence :

```
>>> taille_mot_de_passe = secrets.choice(range(8, 16))
>>> mot_de_passe = "".join([secrets.choice(alphabet) for _ in
 ↪  range(taille_mot_de_passe + 1)])
>>> print(mot_de_passe)
jdVQkGQXzcjPmo
>>>
```

Ce code est un peu plus compressé, n'hésitez pas à le décomposer en plusieurs lignes si ça vous aide. D'abord, on génère une taille aléatoire pour notre mot de passe. On utilise le retour de `range` qui est une séquence (et dans ce cas précis, c'est tout ce qu'il faut pour choisir dans un nombre aléatoire). Ensuite, on crée notre mot de passe en utilisant la même syntaxe qu'avant. Notez qu'on utilise `_` comme nom de variable, ce qui veut dire qu'on n'en a pas vraiment besoin ici (on aurait pu l'appeler `i`, `_` est le nom conventionnel... d'une variable qui n'a pas de nom).

Voyons maintenant la fonction `token_hex`. Ce sera bien plus rapide :

```
>>> secrets.token_hex(16)
'835b4531d1a4624dc9fdebd80395c620'
>>>
```

L'argument donné à la fonction est le nombre d'octets générés. Le retour de la fonction est une chaîne comprenant deux fois plus de caractères (32 dans notre cas). Chaque caractère généré apparaît sur deux chiffres hexadécimaux, d'où ce résultat.

Générer un jeton (token) est une opération assez spécifique. Il est plus probable que vous utilisiez `secrets.choice` au final. Il existe d'autres fonctions dans ce module que je vous laisse découvrir, si vous êtes intéressé.

Si vous voulez aller plus loin, tournez-vous vers la documentation officielle :

— Module `getpass` : `https://docs.python.org/fr/3/library/getpass.html`
— Module `hashlib` : `https://docs.python.org/fr/3/library/hashlib.html`
— Module `secrets` : `https://docs.python.org/fr/3/library/secrets.html`

En résumé

— Pour demander à l'utilisateur de saisir un mot de passe, il vaut mieux passer par le module `getpass`.
— La fonction `getpass` fonctionne de la même façon que `input`, sauf qu'elle n'affiche pas ce que l'utilisateur saisit.
— Pour chiffrer un mot de passe, on va utiliser le module `hashlib`.
— Ce module contient en attributs les différents algorithmes utilisables pour chiffrer nos mots de passe.
— Pour générer des mots de passe aléatoires, on utilise le module `secrets`.

Chapitre 32

Le réseau

Difficulté :

Vaste sujet que le réseau ! Si je devais faire une présentation détaillée, ou même parler des réseaux en général, il me faudrait bien plus d'un chapitre rien que pour la théorie. Dans ce chapitre, nous allons donc apprendre à faire communiquer deux applications grâce aux *sockets*, des objets qui connectent un client à un serveur et transmettent des données de l'un à l'autre.

Si cela ne vous met pas l'eau à la bouche...

Brève présentation du réseau

On va s'attacher ici à comprendre comment faire communiquer deux applications, qui peuvent être sur la même machine mais aussi sur des machines distantes. Dans ce cas, elles se connectent grâce au réseau local ou à Internet.

Il existe plusieurs protocoles de communication en réseau. C'est un peu comme la communication orale : pour que les échanges se passent correctement, les deux (ou plus) parties en présence doivent parler la même langue. Nous allons ici parler du protocole **TCP**.

Le protocole TCP

Le sigle TCP signifie *Transmission Control Protocol*, soit « protocole de contrôle de transmission ». Concrètement, il sert à connecter deux applications et à leur faire échanger des informations.

Ce protocole est dit « orienté connexion », c'est-à-dire que les applications sont connectées pour communiquer et que l'on peut être sûr, quand on envoie une information au travers du réseau, qu'elle a bien été reçue par l'autre application. Si la connexion est rompue pour une raison quelconque, les applications doivent la rétablir pour communiquer de nouveau.

Cela vous paraît peut-être évident mais le protocole **UDP** [1], par exemple, n'est pas connecté : une application envoie des informations au travers du réseau sans se soucier de savoir si sa cible les reçoit correctement.

Ce type de protocole est utile si vous avez besoin de faire transiter beaucoup d'informations au travers du réseau sans qu'une petite perte occasionnelle d'informations ne soit très handicapante. Par exemple, dans des jeux graphiques en réseau, le serveur envoie très fréquemment des informations au client pour qu'il actualise sa fenêtre. Cela fait beaucoup à transmettre mais ce n'est pas dramatique s'il y a une petite perte d'informations de temps à autre puisque, quelques millisecondes plus tard, le serveur les renverra de nouveau.

En attendant, c'est le protocole **TCP** qui nous intéresse. Il est un peu plus lent que le protocole **UDP** mais plus sûr, ce qui est préférable dans notre cas.

Clients et serveur

Dans l'architecture que nous allons étudier, on trouve en général un serveur et plusieurs clients. Le serveur, c'est une machine qui va traiter les requêtes des clients.

Si vous accédez par exemple à OpenClassrooms, c'est parce que votre navigateur, faisant office de client, se connecte au serveur d'OpenClassrooms. Il lui envoie un message en lui demandant la page que vous souhaitez afficher et le serveur d'OpenClassrooms la lui fournit.

1. *User Datagram Protocol*

Cette architecture est très fréquente, même si ce n'est pas la seule envisageable.

Dans les exemples, nous allons créer deux applications : l'application **serveur** et l'application **client**. Le serveur *écoute* donc en attendant des connexions et les clients se connectent au serveur.

Les différentes étapes

Nos applications vont fonctionner selon un schéma assez similaire. Voici dans l'ordre les étapes du client et du serveur. Elles sont très simplifiées, les serveurs communiquant pour la plupart avec plusieurs clients, mais nous ne verrons pas cela tout de suite.

Le serveur :

1. attend une connexion de la part du client ;
2. accepte la connexion ;
3. échange des informations avec le client ;
4. ferme la connexion.

Le client :

1. se connecte au serveur ;
2. échange des informations avec le serveur ;
3. ferme la connexion.

Comme on l'a vu, le serveur peut dialoguer avec plusieurs clients : c'est tout l'intérêt. Si le serveur d'OpenClassrooms ne pouvait dialoguer qu'avec un seul client à la fois, il faudrait attendre votre tour, peut-être assez longtemps, avant d'avoir accès à vos pages. Et, pour les mêmes raisons, les jeux en réseau ou les logiciels de messagerie instantanée seraient bien plus complexes.

Établir une connexion

Pour que le client se connecte au serveur, il nous faut deux informations :

— Le **nom d'hôte** (*host name* en anglais), qui identifie une machine sur Internet ou sur un réseau local. Les noms d'hôtes représentent des adresses IP de façon plus claire (un nom comme `google.fr` est plus facile à retenir que l'adresse IP correspondante `74.125.224.84`).
— Un numéro de port, qui est souvent propre au type d'information que l'on va échanger. Si on demande une connexion web, le navigateur va en général interroger le port `80` si c'est en `http` ou le port `443` si c'est en connexion sécurisée (`https`). Le port est compris entre 0 et 65 535 et les numéros entre 0 et 1 023 sont réservés par le système. On peut les utiliser, mais ce n'est pas une très bonne idée.

Pour résumer, quand votre navigateur tente d'accéder à OpenClassrooms, il établit une connexion avec le serveur dont le nom d'hôte est `openclassrooms.com` sur le port `443`. Dans ce chapitre, nous allons plus volontiers travailler avec des noms d'hôtes qu'avec des adresses IP.

Les sockets

Comme on va le voir, les *sockets* sont des objets qui ouvrent une connexion avec une machine locale ou distante et échangent avec elle.

Ces objets sont définis dans le module `socket`, que nous allons avant tout importer.

```
1 import socket
```

Nous allons d'abord créer notre serveur puis, en parallèle, un client. Nous ferons communiquer les deux. Commençons par le serveur.

Construire notre socket

Nous allons pour cela faire appel au constructeur `socket`. Dans le cas d'une connexion **TCP**, il prend les deux paramètres suivants, dans l'ordre :

- `socket.AF_INET` : la famille d'adresses, ici ce sont des adresses Internet ;
- `socket.SOCK_STREAM` : le type du *socket*, `SOCK_STREAM` pour le protocole TCP.

```
1 >>> connexion_principale = socket.socket(socket.AF_INET,
  ↪  socket.SOCK_STREAM)
2 >>>
```

Connecter le socket

Pour que le serveur attende les connexions de clients, on utilise la méthode `bind`. Elle prend un paramètre : le tuple `(nom_hôte, port)`.

Attends un peu, je croyais que c'était notre client qui se connectait à notre serveur, pas l'inverse...

Oui mais, pour que notre serveur écoute sur un port, il faut le configurer en conséquence. Donc, dans notre cas, le nom de l'hôte sera vide et le port sera celui que vous voulez, entre 1024 et 65535.

```
1 >>> connexion_principale.bind(('', 12800))
2 >>>
```

Faire écouter notre socket

Bien. Notre *socket* est prêt à écouter sur le port `12 800`, mais il n'écoute pas encore. On va avant tout lui préciser le nombre maximum de connexions qu'il peut recevoir sur ce port sans les accepter. On utilise pour cela la méthode `listen`. On lui passe généralement `5` en paramètre.

Cela veut dire que notre serveur ne pourra dialoguer qu'avec 5 clients maximum ?

Non. Cela veut dire que si 5 clients tentent de se connecter sans que le serveur l'accepte, aucun autre client ne pourra se connecter. Généralement, très peu de temps après que le client a demandé la connexion, le serveur l'accepte. Vous pouvez donc avoir bien plus de clients connectés.

```
>>> connexion_principale.listen(5)
>>>
```

Accepter une connexion venant du client

Enfin, dernière étape, on va accepter une connexion. Aucune ne s'est encore présentée mais la méthode `accept` va bloquer le programme tant qu'aucun client ne s'est connecté.

Il est important de noter que la méthode `accept` renvoie deux informations :

- le *socket* connecté qui vient de se créer, celui qui va nous permettre de dialoguer avec notre client ;
- un tuple représentant l'adresse IP et le port de connexion du client.

Le port de connexion du client... ce n'est pas le même que celui du serveur ?

Non car votre client, en ouvrant une connexion, passe par un port dit « de sortie » qui va être choisi par le système parmi les ports disponibles. Pour schématiser, quand un client se connecte à un serveur, il emprunte un port (une forme de porte, si vous voulez) puis établit la connexion sur le port du serveur. Il y a donc deux ports dans notre histoire mais celui qu'utilise le client pour ouvrir sa connexion ne va pas nous intéresser.

```
>>> connexion_avec_client, infos_connexion =
↪   connexion_principale.accept()
```

Cette méthode, comme vous le voyez, bloque le programme. Elle attend qu'un client se connecte. Laissons cette fenêtre Python ouverte et, à présent, ouvrons-en une nouvelle pour construire notre client.

Création du client

Commencez par construire votre *socket* de la même façon :

```
>>> import socket
>>> connexion_avec_serveur = socket.socket(socket.AF_INET,
 ↪  socket.SOCK_STREAM)
>>>
```

Connecter le client

On va utiliser la méthode `connect`. Elle prend en paramètre un tuple, comme `bind`, contenant le nom d'hôte et le numéro du port identifiant le serveur.

Le numéro du port sur lequel on veut se connecter, vous le connaissez : c'est `12 800`. Puisque nos deux applications Python sont sur la même machine, le nom d'hôte va être `localhost`[2].

```
>>> connexion_avec_serveur.connect(('localhost', 12800))
>>>
```

Et voilà, notre serveur et notre client sont connectés !

Si vous retournez dans la console Python abritant le serveur, vous constatez que la méthode `accept` ne bloque plus, puisqu'elle vient d'accepter la connexion demandée par le client. Vous pouvez donc de nouveau saisir du code côté serveur :

```
>>> print(infos_connexion)
('127.0.0.1', 2901)
>>>
```

La première information, c'est l'adresse IP du client. Ici, elle vaut `127.0.0.1` c'est-à-dire l'IP de l'ordinateur local. Dites-vous que l'hôte `localhost` redirige vers l'IP `127.0.0.1`.

Le second est le port de sortie du client, qui ne nous intéresse pas ici.

Faire communiquer nos sockets

Bon, maintenant, comment faire communiquer nos *sockets* ? Eh bien, en utilisant les méthodes `send` pour envoyer et `recv` pour recevoir.

Les informations que vous transmettrez seront des chaînes de bytes, pas de caractères !

2. C'est-à-dire la machine locale.

Voici ce qu'on programme côté serveur :

```
>>> connexion_avec_client.send(b"Je viens d'accepter la
 ↪  connexion")
32
>>>
```

La méthode `send` vous affiche le nombre de caractères envoyés.

Maintenant, côté client, on va recevoir le message que l'on vient d'envoyer. La méthode `recv` prend en paramètre le nombre de caractères à lire. Généralement, on lui passe la valeur `1024`. Si le message est plus grand que 1024 caractères, on récupérera le reste après.

Dans la fenêtre Python côté client, on écrit donc :

```
>>> msg_reçu = connexion_avec_serveur.recv(1024)
>>> msg_reçu
b"Je viens d'accepter la connexion"
>>>
```

Magique, non ? Songez que ce petit mécanisme peut servir à faire communiquer des applications entre elles non seulement sur la machine locale, mais aussi sur des machines distantes et reliées par Internet.

Le client a également la possibilité d'envoyer des informations au serveur et le serveur peut les recevoir, tout cela grâce aux méthodes `send` et `recv` que nous venons de voir.

Fermer la connexion

Pour fermer la connexion, il faut appeler la méthode `close` de notre *socket*.

Voici le code côté serveur :

```
>>> connexion_avec_client.close()
>>>
```

Et voilà celui côté client :

```
>>> connexion_avec_serveur.close()
>>>
```

Voilà ! Je vais récapituler en vous présentant dans l'ordre un petit serveur et un client. Et pour finir, je vous montrerai une façon d'optimiser un peu notre serveur pour qu'il gère plusieurs clients à la fois.

Le serveur

Pour éviter les confusions, je vous remets ici le code du serveur, légèrement amélioré. Il n'accepte qu'un seul client (nous verrons plus bas comment en accepter plusieurs) et il tourne jusqu'à recevoir du client le message `fin`.

À chaque fois que le serveur reçoit un message, il envoie en retour le message `'5/5'`.

```
import socket

hôte = ''
port = 12800

connexion_principale = socket.socket(socket.AF_INET,
↪  socket.SOCK_STREAM)
connexion_principale.bind((hôte, port))
connexion_principale.listen(5)
print(f"Le serveur écoute à présent sur le port {port}")

connexion_avec_client, infos_connexion =
↪  connexion_principale.accept()

msg_reçu = b""
while msg_reçu != b"fin":
    msg_reçu = connexion_avec_client.recv(1024)
    # L'instruction ci-dessous peut lever une exception si le
    ↪  message
    # reçu comporte des accents
    print(msg_reçu.decode())
    connexion_avec_client.send(b"5 / 5")

print("Fermeture de la connexion")
connexion_avec_client.close()
connexion_principale.close()
```

Voilà pour le serveur. Il est minimal, vous en conviendrez, mais il est fonctionnel. Nous l'améliorerons plus loin.

Le client

Le client tente de se connecter sur le port `12 800` de la machine locale. Il demande à l'utilisateur de saisir quelque chose au clavier et l'envoie au serveur, puis attend sa réponse.

```
import socket

hôte = "localhost"
port = 12800

connexion_avec_serveur = socket.socket(socket.AF_INET,
↪  socket.SOCK_STREAM)
connexion_avec_serveur.connect((hôte, port))
print(f"Connexion établie avec le serveur sur le port {port}")

```

```
msg_à_envoyer = b""
while msg_à_envoyer != b"fin":
    msg_à_envoyer = input("> ")
    # Peut planter si vous tapez des caractères spéciaux
    msg_à_envoyer = msg_à_envoyer.encode()
    # On envoie le message
    connexion_avec_serveur.send(msg_à_envoyer)
    msg_reçu = connexion_avec_serveur.recv(1024)
    print(msg_reçu.decode()) # Là encore, peut planter s'il y
    ↪  a des accents

print("Fermeture de la connexion")
connexion_avec_serveur.close()
```

Que font les méthodes `encode` et `decode` ?

`encode` est une méthode de chaîne de caractères. Elle accepte un nom d'encodage en paramètre et permet de transformer une chaîne de caractères en chaîne de `bytes` (octets). C'est, comme vous le savez, ce type de chaîne que `send` accepte. En fait, cette méthode *encode* la chaîne `str` en fonction d'un encodage précis (par défaut, **UTF-8**).

`decode`, à l'inverse, est une méthode de chaîne de `bytes`. Elle aussi prend un encodage en paramètre et elle renvoie une chaîne de caractères décodée grâce à l'encodage (par défaut `UTF-8`).

Si l'encodage de votre console est différent d'`UTF-8` (ce sera souvent le cas sur Windows), des erreurs risquent de se produire si les messages que vous encodez ou décodez comportent des accents.

Un serveur plus élaboré

Quel sont les problèmes de notre serveur ?

Si vous y réfléchissez, il y en a plusieurs !

— D'abord, il ne sait accepter qu'un seul client. Si d'autres clients veulent se connecter, ils n'en reçoivent pas l'autorisation.
— Ensuite, on part du principe qu'on attend le message d'un client et qu'on lui répond immédiatement après réception. Or, ce n'est pas toujours le cas : parfois vous envoyez un message au client alors qu'il ne vous a rien demandé, parfois vous recevez des informations de sa part sans lui avoir rien envoyé.

> Prenez un logiciel de messagerie instantanée : est-ce que, pour dialoguer, vous êtes obligés d'attendre que votre interlocuteur vous réponde ? Ce n'est pas « j'envoie un message, il me répond, je lui réponds, il me répond »... Parfois, souvent même, vous enverrez deux messages à la suite, peut-être même trois, ou l'inverse, qui sait ?
> Bref, on doit pouvoir envoyer et recevoir plusieurs messages dans un ordre inconnu. Avec notre technique, c'est impossible (faites le test si vous voulez).

En outre, les erreurs sont assez mal gérées, vous en conviendrez.

Le module `select`

Le module `select` va nous autoriser à interroger plusieurs clients dans l'attente d'un message à réceptionner, sans paralyser notre programme.

Pour schématiser, `select` va écouter sur une liste de clients et retourner au bout d'un temps précisé. Ce que renvoie `select`, c'est la liste des clients qui ont un message à recevoir. Il suffit de parcourir ces clients, de lire les messages en attente (grâce à `recv`) et le tour est joué.

Sur Linux, `select` est utilisable sur autre chose que des *sockets* mais, cette fonctionnalité n'étant pas portable, je ne fais que la mentionner ici.

En théorie

La fonction qui nous intéresse porte le même nom que le module associé, `select`. Elle prend trois ou quatre arguments :

- `rlist` : la liste des *sockets* en attente d'être lus ;
- `wlist` : la liste des *sockets* en attente d'être écrits ;
- `xlist` : la liste des *sockets* en attente d'une erreur (je ne m'attarderai pas sur cette liste) ;
- `timeout` : le délai pendant lequel la fonction attend avant de retourner. Si vous spécifiez `0`, la fonction retourne immédiatement. Si ce paramètre n'est pas précisé, la fonction retourne dès qu'un des *sockets* change d'état (est prêt à être lu s'il est dans `rlist` par exemple) mais pas avant.

Concrètement, nous allons surtout nous intéresser au premier et au quatrième paramètres.

On cherche à écrire des *sockets* dans une liste et que `select` les surveille, en retournant dès que l'un d'eux est prêt à être lu. Comme cela, notre programme ne bloque pas et sait recevoir des messages de plusieurs clients dans un ordre complètement inconnu.

Maintenant, parlons du `timeout` : comme je vous l'ai dit, si vous ne le précisez pas, `select` bloque jusqu'au moment où l'un des *sockets* que nous écoutons est prêt à être lu. Si vous précisez un `timeout` de `0`, `select` retournera tout de suite. Sinon, `select` retournera au bout du temps que vous indiquez en secondes, ou plus tôt si un *socket* est prêt à être lu.

Concrètement, si vous précisez un `timeout` de `1`, la fonction va bloquer pendant une seconde maximum et retournera prématurément si un des *sockets* en écoute est prêt à être lu dans l'intervalle (c'est-à-dire si un des clients envoie un message au serveur).

`select` renvoie trois listes, là encore `rlist`, `wlist` et `xlist`, sauf qu'il ne s'agit pas des listes fournies en entrée mais uniquement des *sockets* « à lire » dans le cas de `rlist`.

Ce n'est pas clair ? Considérez cette ligne (ne l'essayez pas encore) :

```
rlist, wlist, xlist = select.select(clients_connectés, [], [],
↪  2)
```

Cette instruction va écouter les sockets contenus dans la liste `clients_connectés`. Elle retournera au plus tard dans 2 secondes, sauf si un client envoie un message. La liste des clients ayant envoyé un message se retrouve dans notre variable `rlist`. On la parcourt ensuite et on peut appeler `recv` sur chacun des *sockets*.

Si ce n'est pas plus clair, rassurez-vous : nous allons voir `select` en action un peu plus loin. Référez-vous également à la documentation du module : `https://docs.python.org/fr/3/library/select.html`

`select` en action

Travaillons un peu sur notre serveur, tout en gardant le même client de test.

Le but est de créer un serveur capable d'accepter plusieurs clients, de recevoir leurs messages et de leur envoyer une confirmation à chaque réception. L'exercice ne change pas beaucoup mais on va utiliser `select`.

Cette fonction va non seulement écouter plusieurs clients déjà connectés, mais également nous indiquer si d'autres se connectent. Si vous vous souvenez, la méthode `accept` est aussi une fonction bloquante. On va du reste l'utiliser de la même façon que précédemment.

Voici le code :

```
import socket
import select

hôte = ''
port = 12800

connexion_principale = socket.socket(socket.AF_INET,
↪  socket.SOCK_STREAM)
connexion_principale.bind((hôte, port))
connexion_principale.listen(5)
print(f"Le serveur écoute à présent sur le port {port}")

serveur_lancé = True
clients_connectés = []
while serveur_lancé:
```

```
    # On va vérifier que de nouveaux clients ne demandent pas à se connecter
    # Pour cela, on écoute la connexion_principale en lecture
    # On attend maximum 50ms
    connexions_demandées, wlist, xlist = select.select([connexion_principale],
            [], [], 0.05)

    for connexion in connexions_demandées:
        connexion_avec_client, infos_connexion = connexion.accept()
        # On ajoute le socket connecté à la liste des clients
        clients_connectés.append(connexion_avec_client)

    # Maintenant, on écoute la liste des clients connectés
    # Les clients renvoyés par select sont ceux devant être lus (recv)
    # On attend là encore 50ms maximum
    # On enferme l'appel à select.select dans un bloc try
    # En effet, si la liste de clients connectés est vide, une exception
    # Peut être levée
    clients_à_lire = []
    try:
        clients_à_lire, wlist, xlist = select.select(clients_connectés,
                [], [], 0.05)
    except select.error:
        pass
    else:
        # On parcourt la liste des clients à lire
        for client in clients_à_lire:
            # Client est de type socket
            msg_reçu = client.recv(1024)
            # Peut planter si le message contient des caractères spéciaux
            msg_reçu = msg_reçu.decode()
            print(f"Reçu {msg_reçu}")
            client.send(b"5 / 5")
            if msg_reçu == "fin":
                serveur_lancé = False

print("Fermeture des connexions")
for client in clients_connectés:
    client.close()
```

```
53
54 connexion_principale.close()
```

C'est inévitablement plus long.

Maintenant, notre serveur sait accepter des connexions de plus d'un client. En outre, il ne se bloque pas dans l'attente d'un message, du moins pas plus de 50 millisecondes.

Je pense que les commentaires sont assez précis pour vous aider à aller plus loin. Ceci n'est naturellement pas encore une version complète mais, grâce à cette base, vous devriez facilement arriver à quelque chose. Pourquoi ne pas coder un mini tchat ?

Les déconnexions fortuites ne sont pas gérées non plus. Toutefois, vous avez assez d'éléments écrire des tests et améliorer notre serveur si cela vous tente.

Et encore plus

Je vous l'ai dit, le réseau est un vaste sujet et, même en se restreignant au sujet que j'ai choisi, il y aurait beaucoup d'autres choses à vous expliquer. Je ne peux tout simplement pas remplacer la documentation :

- Module `socket` : `https://docs.python.org/fr/3/library/socket.html`
- Module `select` : `https://docs.python.org/fr/3/library/select.html`
- Module `socketserver` qui est de plus en plus préféré pour le code serveur : `https://docs.python.org/fr/3/library/socketserver.html`
- Module `selectors` qui est préféré à `select` : `https://docs.python.org/fr/3/library/selectors.html`

En résumé

- Dans la structure réseau que nous avons décrite, on trouve un **serveur** capable de dialoguer avec plusieurs **clients**.
- Pour créer une connexion côté serveur ou client, on utilise le module `socket` et la classe `socket` de ce module.
- Pour se connecter à un serveur, le *socket* client utilise la méthode `connect`.
- Pour écouter sur un port précis, le serveur utilise d'abord la méthode `bind` puis la méthode `listen`.
- Pour s'échanger des informations, les *sockets* client et serveur utilisent les méthodes `send` et `recv`.
- Pour fermer une connexion, le *socket* serveur ou client utilise la méthode `close`.
- Le module `select` est utile si l'on souhaite créer un serveur capable de gérer plusieurs connexions simultanément ; toutefois, il en existe d'autres.

Chapitre 33

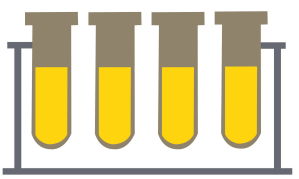
Les tests unitaires avec `unittest`

Difficulté : 

Tester ! Tout un monde. Vous allez découvrir dans ce chapitre comment tester le bon fonctionnement de votre programme et apprendre à le rendre aussi stable que possible au fur et à mesure que vous proposerez de nouvelles améliorations.

Si vous pensez que tester ne sert à rien ou n'est utile que quand tout le développement est fini, je vous encourage vivement à lire ce chapitre, ne serait-ce que pour information.

Pour suivre les explications, vous aurez besoin de savoir comment créer des classes et avoir une idée du fonctionnement de l'héritage.



Pourquoi tester ?

On va parler de tests... mais qu'est-ce qu'on entend par « tester » ?

C'est la première question et elle est très importante !

Dans ce chapitre, je vais parler de tests (principalement de tests unitaires), qui vérifient que votre code réagit comme il le devrait et qu'il continue à réagir comme il le devrait après de nouvelles améliorations.

Certains développeurs refusent de travailler sur du code qui n'est pas le leur s'il n'a pas de documentation. Pour ce que j'en ai vu, un nombre plus important encore de développeurs refuse de le faire si le code n'a pas de test.

Admettons que vous travaillez sur votre projet qui propose plusieurs fonctions, utilisées par d'autres développeurs ou utilisateurs. Vous pouvez être tout seul sur le projet et ne proposer qu'une dizaine de fonctions, c'est bien suffisant ; le plus important, c'est que votre code soit utilisé par d'autres.

Vous commencez à coder votre onzième fonction, qui utilise la troisième déjà développée. Cependant, vous vous heurtez à un problème : votre nouvelle fonction ne marche pas comme il faut.

Après enquête, vous vous rendez compte que ce n'est pas votre fonction 11 qui pose problème, mais la fonction 3 que la 11 appelle. Elle ne répond plus à votre besoin et vous vous dites, naturellement, « je vais la modifier ».

Vous modifiez donc votre fonction 3. La 11 marche, enfin, sans problème. Vous proposez votre nouvelle version à vos utilisateurs.

Et vous recevez un choeur de protestations : jugez donc ! Ils utilisaient votre fonction 3 sans problème, mais avec votre nouvelle version, rien ne marche plus.

Les tests sont une solution possible : pour chaque fonctionnalité, il y aura un test et ce dernier va s'assurer que votre programme reste valide même quand vous le modifierez, ce qui deviendra de plus en plus important au fur et à mesure qu'il gagnera en fonctionnalités, bien entendu.

Est-ce qu'on doit tester un code quand tout est développé ?

Non ! Si vous pouvez le faire dès le début, dès les premières lignes de code que vous écrivez, c'est mieux. Sachez qu'il est assez difficile d'écrire des tests quand votre programme comporte déjà plusieurs centaines de fonctionnalités ; il vaut mieux le faire petit à petit.

Il existe aussi plusieurs méthodes de développement, dont le TDD *(Test-Driven Development)*, qui veut que l'on écrive les tests avant d'écrire le code. Je ne rentrerai pas dans le détail ici, mais je vous conseille vivement d'écrire vos tests unitaires même si vous n'avez qu'un tout petit projet avec 4 ou 5 fonctions. Il y a une chance non négligeable que le petit projet devienne grand ; avec des tests à portée de main, vous dormirez plus tranquille.

Est-ce difficile de tester un programme ?

Une fois que vous maîtrisez une des méthodes de test et que vous l'appliquez à votre programme au fur et à mesure, non ce n'est absolument pas difficile. Vous allez voir dans ce chapitre comment utiliser des tests unitaires. Il existe d'autres méthodes de test proposées par Python, mais c'est celle-ci que je trouve la plus rapide à prendre en main ainsi que la plus flexible. Ce chapitre est là pour vous guider par à pas vers la création de vos premiers tests unitaires et même vers la gestion de nombreux tests quand votre projet sera plus grand.

Qui écrit les tests ?

Le développeur, la plupart du temps. Là encore, la méthode de test utilisée permet parfois à d'autres personnes de s'en charger, mais les tests unitaires sont souvent écrits par des développeurs (ou des utilisateurs sachant programmer). Comme vous allez le voir, ils ne sont pas très difficiles à écrire, mais vous passerez malgré tout par Python pour ce faire.

Passons à la pratique, la découverte du module `unittest` !

Premiers exemples de tests unitaires

Le module `unittest` de la bibliothèque standard de Python inclut le mécanisme des tests unitaires.

Voici la structure que vous rencontrerez le plus souvent :

— Pour chaque fonctionnalité, un ensemble de fonctions, de classes, de modules, de *packages* et autre. Tout ce cours est là pour vous montrer comment réaliser cette partie du développement.
— Pour chaque fonctionnalité, un test qui vérifie qu'elle fait bien ce qu'on lui demande (par exemple, que si elle est appelée avec certains paramètres, elle retourne telle valeur).

Nous allons nous intéresser ici à ce second point dans la liste : comment tester une fonctionnalité.

Tester une fonctionnalité existante

Pour commencer, nous allons tester une fonctionnalité déjà existante, proposée dans l'un des modules de Python. Je vais reprendre les exemples de la documentation officielle, qui sont assez faciles à comprendre : `https://docs.python.org/fr/3/library/unittest.html`

Pour cet exemple, nous allons nous intéresser au module `random` que nous avons déjà utilisé. Nous allons chercher à tester le fonctionnement en particulier de trois fonctions :

— `random.choice` : cette fonction retourne un élément au hasard de la séquence précisée en paramètre.

```
>>> liste = ["chat", "chien", "renard", "serpent", "cheval",
             "parapluie"]
>>> random.choice(liste)
'renard'
>>>
```

— `random.shuffle` : cette fonction mélange une liste. La liste d'origine est modifiée.

```
>>> liste = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> random.shuffle(liste)
>>> liste
[3, 4, 7, 1, 8, 6, 5, 9, 2]
>>>
```

— `random.sample` : cette fonction prend une séquence et un nombre en paramètres. Elle retourne une nouvelle séquence contenant autant d'éléments que le nombre indiqué, sélectionnés aléatoirement dans la séquence d'origine.

```
>>> liste = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
>>> random.sample(liste, 5)
['b', 'a', 'c', 'j', 'e']
>>> # Ou peut-être que cet exemple sera plus clair
... random.sample(range(1000), 10)
[389, 406, 890, 955, 837, 401, 971, 716, 954, 862]
>>>
```

Structure de base d'un test unitaire

Nous le verrons plus loin, un test unitaire peut être constitué de nombreux tests répartis dans plusieurs *packages* et modules. Pour l'instant, nous n'allons nous intéresser qu'à un **test case**, la forme la plus simple.

Pour créer un test unitaire, la première chose est de définir une classe héritant de `unittest.TestCase` :

```
1 import random
2 import unittest
3
4 class RandomTest(unittest.TestCase):
```

On peut définir ensuite un test dans une méthode dont le nom commence par `test`.

Test de la fonction `random.choice`

Voyons tout d'abord le test de la fonction `choice` :

```
class RandomTest(unittest.TestCase):

    """Test case utilisé pour tester les fonctions du module
    ↪  'random'."""

    def test_choice(self):
        """Teste le fonctionnement de la fonction
        ↪  'random.choice'."""
        liste = list(range(10))
        elt = random.choice(liste)
        # Vérifie que 'elt' est dans 'liste'
        self.assertIn(elt, liste)
```

Quelques explications s'imposent :

1. D'abord à la première ligne, on crée une liste de `0` à `9`.
2. Ensuite on appelle la fonction `random.choice` sur notre liste et on récupère le retour.
3. Enfin, on vérifie que l'élément retourné par `random.choice` se trouve bien dans notre liste. On utilise pour ce faire une méthode `assertIn` et pas le mot-clé `assert`. En fait, `unittest.TestCase` propose plusieurs méthodes d'assertion que nous utiliserons dans nos tests unitaires. Une assertion lève une exception qui serait considérée par `unittest` comme une erreur. Nous verrons plus loin comment les erreurs sont gérées.

Si vous exécutez ce code dans votre interpréteur… rien ne se passe ! Vous avez créé une classe mais vous n'avez pas demandé au test de se lancer. Pour ce faire, vous pouvez exécuter l'instruction suivante :

```
unittest.main()
```

Et vous devriez obtenir quelque chose comme ce qui suit :

```
.
----------------------------------------------------------------------
Ran 1 test in 0.003s

OK
```

L'appel à `unittest.main` ferme la console Python, soyez prévenu ; ce n'est pas une erreur, mais bien un comportement attendu.

Le retour affiché se décompose en trois parties :

— D'abord, la première ligne contient un caractère par test exécuté. Les principaux caractères sont un point si le test s'est validé, la lettre `F` si le test n'a pas obtenu le bon résultat et la lettre `E` si le test a rencontré une erreur (si une exception a été levée pendant l'exécution de la méthode).
— Ensuite se trouve une ligne récapitulative du nombre de tests exécutés.
— Enfin, la dernière ligne récapitule le nombre de réussites ou échecs ou erreurs. Si tout va bien, cette dernière ligne devrait être simplement `OK`.

Faisons échouer un test

Modifions notre test pour être sûr de provoquer un échec :

```
class RandomTest(unittest.TestCase):

    """Test case utilisé pour tester les fonctions du module
    ↪  'random'."""

    def test_choice(self):
        """Teste le fonctionnement de la fonction
        ↪  'random.choice'."""
        liste = list(range(10))
        elt = random.choice(liste)
        self.assertIn(elt, ('a', 'b', 'c'))
```

Voici ce qu'on obtient après un appel à `unittest.main()` :

```
F
======================================================================
FAIL: test_choice (__main__.RandomTest)
Teste le fonctionnement de la fonction 'random.choice'.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "code.py", line 13, in test_choice
    self.assertIn(elt, ('a', 'b', 'c'))
AssertionError: 0 not found in ('a', 'b', 'c')

----------------------------------------------------------------------
Ran 1 test in 0.004s

FAILED (failures=1)
```

Vous voyez que l'on obtient plusieurs informations sur les tests qui ne échouent. D'abord, notez qu'ici, on parle d'échec *(failure)* et non pas d'erreur *(error)*. Cela signifie que notre assertion ne s'est pas vérifiée, mais que notre test s'est correctement exécuté. Vous pouvez essayer de provoquer une erreur dans la méthode de test aussi, pour voir le résultat.

Le traceback est assez détaillée : elle donne la ligne de l'erreur avec les appels successifs, pour remonter la piste de l'erreur. Le message lui-même précise pourquoi le test a échoué (`0 not found in ('a', 'b', 'c')`).

Test de la fonction `random.shuffle`

Intéressons-nous maintenant à la fonction `random.shuffle`. Souvenez-vous : elle prend une liste en paramètre et la mélange aléatoirement.

En vous inspirant du premier exemple, essayez d'écrire la méthode de test correspondante. Il vous faut réfléchir à comment vérifier qu'une liste, après avoir été mélangée, correspond à une liste d'éléments entre `0` et `9`.

Je vous conseille d'utiliser cette fois la méthode d'assertion `assertEqual`, qui prend deux arguments en paramètres et vérifie s'ils sont identiques.

À vous de jouer !

```
class RandomTest(unittest.TestCase):

    """Test case utilisé pour tester les fonctions du module
    ↪ 'random'."""

    # Autres méthodes de test
    def test_shuffle(self):
        """Teste le fonctionnement de la fonction
        ↪ 'random.shuffle'."""
        liste = list(range(10))
        random.shuffle(liste)
        liste.sort()     # Supposée testée
        self.assertEqual(liste, list(range(10)))
```

Comme vous le voyez, on appelle la fonction `random.shuffle` avant de trier de nouveau notre liste, qui devrait alors être identique à notre liste d'origine (`list(range(10))`).

Ici, nous avons utilisé la méthode `assertEqual`, qui sera sans doute celle que vous utiliserez le plus souvent. Nous verrons un peu plus loin une liste des méthodes d'assertion proposées par `unittest.TestCase`.

Test de la fonction `random.sample`

Enfin, écrivons notre méthode de test de la fonction `random.sample`. Souvenez-vous qu'elle prend deux paramètres : une séquence et un nombre `K`. Elle retourne une liste contenant `K` éléments sélectionnés aléatoirement dans notre séquence de base.

Voyons une première approche :

```
class RandomTest(unittest.TestCase):

    """Test case utilisé pour tester les fonctions du module
    ↪ 'random'."""
```

```
4
5       # Autres méthodes de test
6       def test_sample(self):
7           """Teste le fonctionnement de la fonction
            ↪ 'random.sample'."""
8           liste = list(range(10))
9           extrait = random.sample(liste, 5)
10          for element in extrait:
11              self.assertIn(element, liste)
```

Jusqu'ici, ce n'est pas bien différent de ce que nous avons fait précédemment.

Avez-vous essayé `random.sample` en précisant un nombre `K` plus élevé que la taille de la séquence ?

```
>>> liste = list(range(10))
>>> random.sample(liste, 20)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\python34\lib\random.py", line 313, in sample
    raise ValueError("Sample larger than population")
ValueError: Sample larger than population
>>>
```

Les erreurs ne doivent jamais être ignorées. Ce comportement est attendu et souhaitable. Autant le tester également :

```
class RandomTest(unittest.TestCase):

    """Test case utilisé pour tester les fonctions du module
    ↪ 'random'."""

    # Autres méthodes de test
    def test_sample(self):
        """Teste le fonctionnement de la fonction
        ↪ 'random.sample'."""
        liste = list(range(10))
        extrait = random.sample(liste, 5)
        for element in extrait:
            self.assertIn(element, liste)

        self.assertRaises(ValueError, random.sample, liste,
        ↪ 20)
```

La dernière ligne mérite quelques explications. On utilise encore une méthode d'assertion `assert*` (cette fois, `assertRaises`).

On peut l'appeler de deux façons :

— soit, comme on vient de le faire, en précisant d'abord le type de l'exception à lever, puis la fonction à appeler (la référence, sans parenthèses) et enfin les paramètres que cette dernière attend ;

— soit en utilisant un gestionnaire de contexte *(context manager)*, qui facilite la lecture du code.

Nous avons vu un gestionnaire de contexte au moment des fichiers. Rappelez-vous, c'est le bloc d'instructions qui commence par le mot-clé `with`.

Voyons comment écrire notre test avec un tel gestionnaire.

```
class RandomTest(unittest.TestCase):

    """Test case utilisé pour tester les fonctions du module
    ↪  'random'."""

    # Autres méthodes de test
    def test_sample(self):
        """Teste le fonctionnement de la fonction
        ↪  'random.sample'."""
        liste = list(range(10))
        extrait = random.sample(liste, 5)
        for element in extrait:
            self.assertIn(element, liste)

        with self.assertRaises(ValueError):
            random.sample(liste, 20)
```

Comme vous le voyez, cette seconde syntaxe est plus lisible :

1. On appelle un nouveau gestionnaire de contexte grâce au mot-clé `with` ouvert sur le retour de la méthode `assertRaises`. Cette fois, on ne passe en paramètre de cette méthode que le type de notre exception.
2. À l'intérieur de notre bloc, se trouve la ligne qui doit lever l'exception `ValueError`. Si le bloc dans le gestionnaire de contexte lève bien l'exception, alors le test passe. Sinon il ne passe pas.

Cette seconde syntaxe est plus lisible, à mon sens, mais je vous montre les deux car vous pourriez trouver la première au cours de vos lectures d'autres codes.

Initialisation des tests

Vous l'avez peut-être remarqué, toutes nos méthodes de test commencent par la ligne suivante :

```
liste = list(range(10))
```

Il existe un moyen pour éviter de la répéter à chaque fois. Nos méthodes de test partagent un point commun : elles sont définies dans la même classe. Autant en profiter.

`unittest.TestCase` nous propose une méthode nommée `setUp` qui est appelée avant chaque méthode de test. Il serait mieux d'y inclure la création de notre liste :

```
class RandomTest(unittest.TestCase):

```

```
    """Test case utilisé pour tester les fonctions du module
    ↪ 'random'."""

    def setUp(self):
        """Initialisation des tests."""
        self.liste = list(range(10))
```

Comme vous le voyez, on écrit directement notre liste en attribut d'instance de notre test. Cela veut dire qu'il va falloir modifier nos méthodes de test pour qu'elles l'utilisent :

```
class RandomTest(unittest.TestCase):

    """Test case utilisé pour tester les fonctions du module
    ↪ 'random'."""

    # Autres méthodes de test
    def test_sample(self):
        """Teste le fonctionnement de la fonction
        ↪ 'random.sample'."""
        extrait = random.sample(self.liste, 5)
        for element in extrait:
            self.assertIn(element, self.liste)

        with self.assertRaises(ValueError):
            random.sample(self.liste, 20)
```

Au lieu de créer la liste, on utilise l'attribut d'instance créé dans la méthode `setUp`. Il existe également une méthode `tearDown` qui est appelée après chaque test.

Récapitulatif complet du code de test

Voici le code complet de notre *test case* et de nos trois méthodes de test.

```
import random
import unittest

class RandomTest(unittest.TestCase):

    """Test case utilisé pour tester les fonctions du module
    ↪ 'random'."""

    def setUp(self):
        """Initialisation des tests."""
        self.liste = list(range(10))

    def test_choice(self):
        """Teste le fonctionnement de la fonction
        ↪ 'random.choice'."""
```

```
14          elt = random.choice(self.liste)
15          self.assertIn(elt, self.liste)
16
17      def test_shuffle(self):
18          """Teste le fonctionnement de la fonction
            ↪  'random.shuffle'."""
19          random.shuffle(self.liste)
20          self.liste.sort()
21          self.assertEqual(self.liste, list(range(10)))
22
23      def test_sample(self):
24          """Teste le fonctionnement de la fonction
            ↪  'random.sample'."""
25          extrait = random.sample(self.liste, 5)
26          for element in extrait:
27              self.assertIn(element, self.liste)
28
29          with self.assertRaises(ValueError):
30              random.sample(self.liste, 20)
```

Souvenez-vous : pour tester le code, vous pouvez ajouter l'instruction `unittest.main()` à la fin de votre module. Nous verrons un peu plus loin un autre moyen, plus simple, pour tester un ou plusieurs module(s).

Les principales méthodes d'assertion

Méthode	Explications
`assertEqual(a, b)`	`a == b`
`assertNotEqual(a, b)`	`a != b`
`assertTrue(x)`	`x is True`
`assertFalse(x)`	`x is False`
`assertIs(a, b)`	`a is b`
`assertIsNot(a, b)`	`a is not b`
`assertIsNone(x)`	`x is None`
`assertIsNotNone(x)`	`x is not None`
`assertIn(a, b)`	`a in b`
`assertNotIn(a, b)`	`a not in b`
`assertIsInstance(a, b)`	`isinstance(a, b)`
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`
`assertRaises(exception, fonction, *args, **kwargs)`	Vérifie que la fonction lève l'exception attendue.

Pour une liste complète, consultez la documentation officielle du module : `https://docs.python.org/fr/3/library/unittest.html`

Nous allons nous intéresser à présent à la découverte automatique des tests par Python.

La découverte automatique des tests

Lancer les tests avec `unittest.main()` peut s'avérer pratique, mais généralement on fera appel à la découverte automatique des tests. Cette fonctionnalité permet de rechercher tous les tests unitaires contenus dans un *package* et de les exécuter.

Lancement de tests unitaires depuis un répertoire

Pour commencer, nous allons essayer de lancer nos tests unitaires depuis un répertoire.

— Créez un répertoire où vous mettez généralement votre code Python. Pour moi, il s'appelle `python` et se trouve dans `Mes Documents`.
— Ouvrez la console. Sous Windows, cliquez sur `Exécuter...` dans le menu `Démarrer` (ou tapez `Windows` + `R`) et entrez `cmd`.
— Déplacez-vous dans le répertoire que vous avez créé :

```
cd python
```

Une fois dans le bon dossier, créez le fichier `test_random.py` et collez le code que nous avons vu précédemment. Sauvegardez ce fichier et revenez dans la console.

Vous devez maintenant exécuter Python avec l'option `-m unittest`. Sous Windows, vous aurez sûrement une commande comme la suivante :

```
python -m unittest
```

Sous Linux, elle s'écrira probablement comme suit :

```
python3.10 -m unittest
```

Si tout se passe bien, vous devriez voir les tests s'exécuter :

```
...
----------------------------------------------------------------
Ran 3 tests in 0.007s

OK
```

L'option `-m` exécute un module spécifique (ici `unittest`). Quand il est appelé directement depuis Python, `unittest` cherche les tests unitaires présents dans le dossier courant. Vous pouvez aussi lui donner un chemin de test à exécuter, par exemple `test_random.RandomTest.test_shuffle` :

1. `test_random` est le nom du module (le nom du fichier sans l'extension).
2. `RandomTest` est le nom de la classe dans notre module.
3. `test_shuffle` est le nom de notre méthode à exécuter.

```
python -m unittest test_random.RandomTest.test_shuffle
.
----------------------------------------------------------------
Ran 1 test in 0.002s
OK
```

Vos tests unitaires doivent être indépendants, c'est-à-dire qu'on peut les exécuter seuls (comme on vient de le faire) ou en groupe (comme on l'a fait plus tôt). En bref, ils ne doivent pas dépendre d'autres tests pour s'exécuter.

Structure d'un projet avec ses tests

Nous allons ici regarder un projet de taille respectable, CherryPy, qui propose un *framework* léger pour créer un serveur web. Je vous conseille d'ailleurs de jeter un oeil à ce projet : `https://cherrypy.org/`

Si vous téléchargez et décompressez les sources, vous verrez un dossier `cherrypy-version`. Entrez dedans et lancez les tests unitaires :

```
python -m unittest
```

Il peut être nécessaire d'installer le *package* au préalable (exécutez la commande `python setup.py install` pour ce faire).

Si Python trouve les tests unitaires du projet, c'est qu'il explore les répertoires de ce dernier. Il y a notamment le répertoire `cherrypy`, qui contient l'ensemble des sources. Il contient le sous-répertoire `test`, dans lequel se trouvent les tests de la bibliothèque.

Je ne rentrerai pas dans le détail ici, mais ce qu'il faut comprendre, c'est que la commande `python -m unittest` explore récursivement les *packages* et modules à la recherche de tests. Tous les *packages* sont explorés, mais les modules (comme les méthodes de test) doivent commencer par `test`.

Généralement, vous trouverez une certaine fonctionnalité (disons, par exemple, dans `cherrypy/fonctionnalite.py`) et le test correspondant dans un module spécifique (`cherrypy/test/test_fonctionnalite.py`). Le découpage du dossier `test` sera souvent le même que celui de vos sources (c'est plus une convention qu'une obligation).

Voilà pour ce tour d'horizon des tests unitaires. Là encore, si vous voulez en apprendre plus, rendez-vous sur la documentation officielle du module : `https://docs.python.org/fr/3/library/unittest.html`

En résumé

- On peut tester nos applications grâce à plusieurs modules sous Python, les tests unitaires étant pris en charge par le module `unittest`.
- Pour créer un test unitaire, il faut créer une classe qui hérite de `unittest.TestCase`. Les méthodes de test ont un nom commençant par `test`.
- La commande `python -m unittest` recherche automatiquement des tests dans le répertoire courant.

Chapitre 34

Déboguer son code avec pdb

Difficulté :

Vous voudriez écrire un programme sans erreur ? Rassurez-vous, vous n'êtes pas le seul. La triste vérité, c'est qu'il n'en existe pas. Ou du moins, je n'en ai jamais entendu parler. Un programme a toujours une erreur, si bien cachée soit-elle. Ce n'est pas toujours de votre faute, l'erreur pourrait se produire dans une situation très rare, mais c'est une erreur tout de même.

Combien de minutes ou d'heures avez-vous passées à fixer votre écran en vous demandant pourquoi une ligne (parfaitement correcte) produit une exception ? Est-ce que vous trouvez les messages des exceptions trop courts pour comprendre l'origine du problème ? Est-ce que vous avez envie de jeter votre ordinateur par la fenêtre (la fenêtre la plus proche, j'entends) après des heures de frustration ? Dites-vous qu'il ne faut pas aller loin pour trouver quelqu'un qui partage la même histoire. La communauté des développeurs est probablement constituée en majorité de personnes ayant expérimenté une bonne dose de frustration de temps à autre. Cela sans parler des autres, bien entendu.

Pour autant, y a-t-il une solution ? Oui et non. Il n'y a pas de solution absolue, mais il y a des réponses. Ce chapitre va expliquer en détail l'utilisation d'un outil couramment employé en programmation, permettant d'identifier la cause des erreurs.

Si donc vous n'avez aucune erreur, passez au chapitre suivant. Sinon...

Déboguer, ça sert à quoi ?

L'outil appelé le débogueur (*debugger* en anglais) est là pour vous aider à comprendre comment votre programme s'exécute. Il vous montre chaque ligne qui s'exécute et l'état des variables à un moment précis. Il vous aide à comprendre la raison pour laquelle une erreur se produit. Il vous permet également de « faire des essais » en entrant du code Python qui n'est pas dans votre programme, mais sera exécuté, pour voir si le code inséré réglerait le problème au final.

Parlons de bogues

Cela n'a rien à voir avec les châtaignes. Un bogue (par opposition à une bogue de châtaigne) est une traduction contestable et contestée du terme anglais **bug** signifiant à l'origine « insecte ». Représentez-vous votre magnifique code comme un circuit électrique (rappelez-vous que l'informatique a des racines profondes dans l'ingénierie) et imaginez de petits insectes qui se baladent sur vos jolies lignes d'instruction (transformés en composants électriques ou fils). Le résultat, dans cette image, est que votre programme risque de ne pas se comporter exactement de la façon que vous espériez, que des « coupures » sont à prévoir. Et remonter à la source de l'erreur n'est pas toujours évident. Cette métaphore indique aussi qu'un bogue peut être aléatoire : il ne se produit pas toujours et comprendre son origine devient extrêmement difficile, puisque votre programme fonctionne comme il le devrait 99,99 % du temps.

Une solution serait d'afficher chaque ligne d'instruction de notre programme et de savoir s'il travaille sur les bonnes données et renvoie bien les bonnes informations au fur et à mesure. Quand le programme est constitué de quelques lignes de code, vous pourriez dessiner un tableau sur une feuille de papier, avec les instructions en lignes et les noms de variables en colonnes. Cependant, le problème ne vient pas toujours des variables... et dans tous les cas, votre programme va grandir.

On a donc inventé plusieurs outils pour exécuter votre programme dans un environnement particulier, vous permettre de faire une pause à un moment précis et d'examiner ce qui se passe. Le jeu consiste d'abord à savoir où se trouve l'erreur (le système d'exceptions de Python localise très précisément les différents fichiers et lignes de code impliquées). Ensuite, on lance le programme en lui demandant de se mettre en pause au moment de l'erreur et on examine l'état des variables, des modules et autres à ce moment.

Nous allons découvrir `pdb` dans ce chapitre, le débogueur officiel utilisé en Python, inclus avec la bibliothèque standard (rien à installer). Il en existe d'autres mais, généralement, maîtriser `pdb` vous permettra d'apprendre d'autres outils assez simplement.

Le débogueur n'est pas un plus

Ce chapitre s'inscrit dans une section optionnelle, sans contenu obligatoire. Apprendre à localiser et résoudre les erreurs est si utile, cependant, que c'est l'un des chapitres que je vous conseille vivement de lire.

Si vous comptez faire votre métier de la programmation en Python, sachez que beaucoup d'employeurs se baseront sur votre capacité à utiliser un débogueur avant de savoir s'ils vont vous engager ou non.

Pourquoi est-ce si important ? Un employeur a plutôt intérêt à ce que l'on code en permanence, non ?

Parfaitement. Si vous écrivez du code sans aucune erreur, votre employeur n'accordera aucune espèce d'importance à votre connaissance de pdb. Toutefois, il a probablement plusieurs années d'expérience en développement et sait qu'il est pour ainsi dire impossible de coder sans commettre d'erreurs. Là intervient un dilemme : d'un côté, si vous vous arrêtez trop souvent pour comprendre l'origine d'une erreur de votre propre code, vous ne produirez pas beaucoup. D'un autre côté, si vous continuez à coder en ignorant vos erreurs (peu importe que le code ne marche pas chez le client, il marche chez moi, donc j'ai fait tout ce que je pouvais), l'employeur considérera votre productivité comme nulle, puisque non exploitable. La plupart du temps, vous devrez donc trouver un juste milieu entre la productivité et la stabilité de votre code. Cet équilibre dépend en grande partie du secteur dans lequel vous travaillez. Il a suffi d'un caractère manquant dans le code pour ruiner l'une des sondes destinées à Mars (Phobos 1, 1988).

Vous devez également trouver un équilibre dans l'utilisation d'un débogueur : si vous lui faites confiance sans apprendre de vos erreurs, vous commettrez les mêmes la prochaine fois. Un employeur va juste constater que votre productivité baisse au lieu d'augmenter. Le débogueur peut vous aider grandement... mais en dernier recours. Le mieux est de comprendre (et apprendre de) vos erreurs. Sachez que c'est une critique dirigée contre le débogueur en général : certains disent que cet outil atrophie notre capacité à trouver les erreurs et qu'au lieu de devenir meilleur, on perd l'agilité intellectuelle nécessaire pour écrire nos programmes avec le minimum d'erreurs possibles. Je ne trancherai pas ici : à mon sens, savoir utiliser un débogueur est une compétence utile. Savoir si vous l'utiliserez ensuite est de votre responsabilité.

pdb pour remonter à la source d'erreurs

Nous allons analyser, dans cette partie, plusieurs erreurs qui ne sont pas nécessairement faciles à comprendre. Notez cependant que, pour l'exemple, nous allons travailler sur de tout petits scripts : trouver l'origine de l'erreur ne devrait pas être trop dur, même sans outil spécialisé. Un débogueur devient particulièrement utile quand le programme sur lequel on travaille commence à prendre un peu de poids.

Lancer pdb

pdb est intégré à Python. Depuis Python 3.7, il existe même un raccourci très pratique pour le lancer. Voyons un petit script d'exemple pour commencer :

```
from random import randint

def rouler(nombre: int) -> int:
    """Fait rouler plusieurs dés à six faces, renvoie la
    ↪ somme.

    Args:
    nombre (int) : le nombre de dés à 6 faces à faire rouler.

    Renvoie :
    aléa (int) : le nombre aléatoire compris entre (nombre)
    et (nombre * 6).

    """
    aléa = 0
    for i in range(nombre):
        aléa += randint(1, 6)

    return aléa

# Test de notre fonction
nombre = rouler(3)
print(f"Nous avons fait rouler 3 dés et obtenu {nombre}.")
```

Ce code ne devrait pas être trop dur à comprendre : on crée une fonction (`rouler`) qui se charge de faire rouler plusieurs dés à 6 faces en fonction du nombre qu'on lui donne en paramètre. Si on fait rouler 5 dés par exemple, on obtient un total entre 5 et 30 (5 dés, 6 faces). On teste ensuite la fonction dans le même script. Si vous lancez ce script avec Python, vous devriez voir quelque chose comme ce qui suit :

```
Nous avons fait rouler 3 dés et obtenu 9.
```

Il ne semble pas y avoir d'erreur dans notre code. Voyons tout de même comment lancer le débogueur pdb avant de voir des cas concrets.

Dans la partie précédente, j'ai expliqué qu'un débogueur permettait de découper notre programme et d'observer un état à un moment précis. Comment « indiquer » cet instant précis ? Depuis Python 3.7, il suffit d'ajouter une fonction : le débogueur se lance et s'arrête à la ligne de notre fonction. Modifiez donc la fonction `rouler` pour inclure la ligne :

```
def rouler(nombre: int) -> int:
    """Fait rouler plusieurs dés à six faces, renvoie la
    ↪ somme.
```

```
3
4       Args:
5           nombre (int) : le nombre de dés à 6 faces à faire
6           rouler.
7
8       Renvoie :
9           aléa (int) : le nombre aléatoire compris entre
10          (nombre) et (nombre * 6).
11
12      """
13      breakpoint()
14      aléa = 0
15      for i in range(nombre):
16          aléa += randint(1, 6)
17
18      return aléa
```

La seule différence est l'appel à la fonction `breakpoint()` au début de la fonction `rouler()`. La fonction native `breakpoint()` lance le débogueur `pdb` et met en pause en attendant notre décision. Essayons de lancer notre programme de nouveau !

```
1 > python hasard.py
2 > d:\livre\python\part4\chap8\hasard.py(14)rouler()
3 -> aléa = 0
4 (Pdb)
```

La toute dernière ligne indique que nous pouvons maintenant entrer des commandes pour déboguer notre programme. Nous allons découvrir ces commandes dans la suite de ce chapitre. Avant tout, observez que `pdb` se lance et indique deux choses : le fichier (et la ligne) où il se trouve, puis le code Python qu'il va exécuter.

Faites bien attention à cela : la ligne que `pdb` vous montre est celle qu'il *va* exécuter, pas celle qu'il vient d'exécuter.

Quand notre code est assez long, connaître juste la ligne que `pdb` va exécuter n'est pas d'un grand secours. On peut demander au débogueur d'afficher un peu plus de détail. C'est la première commande que vous apprendrez ici : `list`, que vous pouvez abréger en `l`.

Entrez donc `l` dans votre console et validez :

```
1 (Pdb) l
2   9         Renvoie :
3  10             aléa (int) : le nombre aléatoire compris
   ↪  entre (nombre) et (nombre * 6).
4  11
5  12         """
6  13         breakpoint()
```

```
 14  ->      aléa = 0
 15          for i in range(nombre):
 16              aléa += randint(1, 6)
 17
 18          return aléa
 19
(Pdb)
```

`pdb` nous affiche à présent les lignes autour du point d'arrêt (`breakpoint`). Notez celle débutant par une flèche (`->`) ; il s'agit, là encore, de la prochaine ligne que va exécuter `pdb`. Les lignes autour aident parfois à comprendre le contexte d'exécution (surtout si nous travaillons dans un gros programme).

Examiner le contexte

Pendant que le programme est ouvert avec `pdb`, nous pouvons examiner le contexte, en particulier les variables qui existent. Entrez simplement le nom de la variable ou de la méthode, comme si vous étiez dans l'interpréteur Python :

```
(Pdb) randint
<bound method Random.randint of <random.Random object at
 ↪  0x00B06A20>>
(Pdb)
```

Nous voyons ici la fonction `random.randint` qui a été importée.

```
(Pdb) aléa
*** NameError: name 'aléa' is not defined
(Pdb)
```

Pourquoi cette erreur ? Souvenez-vous que nous sommes juste avant la ligne suivante :

```
aléa = 0
```

Notre variable n'existe donc pas encore.

Tout ça, c'est bien beau, mais sans avancer, on ne fait pas grand-chose. Pour passer à la ligne suivante, utilisez la commande `next` (abrégée en `n`).

```
(Pdb) n
> d:\livre\python\part4\chap8\hasard.py(15)rouler()
-> for i in range(nombre):
(Pdb)
```

Python a exécuté notre première ligne (`aléa = 0`) et il attend maintenant avant la seconde ligne. Avant de continuer, vérifions bien que notre variable `aléa` existe cette fois :

```
(Pdb) aléa
0
(Pdb)
```

Cette fois, la variable existe bien et elle vaut `0`.

Avançons un petit peu pour voir. Vous devriez maintenant pouvoir suivre les étapes sans trop d'explications :

```
(Pdb) n
> d:\livre\python\part4\chap8\hasard.py(16)rouler()
-> aléa += randint(1, 6)
(Pdb) i
0
(Pdb) n
> d:\livre\python\part4\chap8\hasard.py(15)rouler()
-> for i in range(nombre):
(Pdb) n
> d:\livre\python\part4\chap8\hasard.py(16)rouler()
-> aléa += randint(1, 6)
(Pdb) n
> d:\livre\python\part4\chap8\hasard.py(15)rouler()
-> for i in range(nombre):
(Pdb) aléa
10
(Pdb) i
1
(Pdb)
```

Non seulement `pdb` fait une pause à chaque ligne, mais en plus il exécute chaque itération de notre boucle (notre boucle, ici, est exécutée `nombre` fois, c'est-à-dire...

```
(Pdb) nombre
3
(Pdb)
```

... trois fois, merci).

Nous voyons notre variable `aléa` progresser (`10` au deuxième tour de boucle).

Un raccourci utile : si vous appuyez sur `Entrée`, `pdb` exécute la commande que vous aviez saisie précédemment. C'est utile, pour répéter la commande `next`.

```
(Pdb) n
> d:\livre\python\part4\chap8\hasard.py(16)rouler()
-> aléa += randint(1, 6)
(Pdb)
> d:\livre\python\part4\chap8\hasard.py(15)rouler()
-> for i in range(nombre):
(Pdb)
> d:\livre\python\part4\chap8\hasard.py(18)rouler()
-> return aléa
(Pdb) aléa
```

```
11
(Pdb) nombre
3
(Pdb) i
2
(Pdb)
```

Après le troisième tour, `pdb` remonte au début de la boucle pour indiquer que, cette fois, on a bien fini (`i` est passé de `0` à `1`, puis à `2`, faisant tourner la boucle 3 fois en tout). Il affiche donc qu'il est prêt et passe à la suite (l'instruction `return` dans notre cas). On vérifie les variables avant de terminer la fonction.

Dans cette section, nous avons donc découvert certaines commandes. J'en ajoute quelques autres que vous utiliserez fréquemment :

— `list` ou `l` : liste les lignes autour du point d'exécution de `pdb`.
— `next` ou `n` : exécute la ligne au point d'exécution, passe à la ligne suivante.
— `quit` ou `q` : quitte `pdb` et referme le programme.
— `continue` ou `c` : ferme `pdb` et continue l'exécution du programme jusqu'au prochain point d'arrêt.

Notez aussi un peu de vocabulaire introduit en passant :

— Point d'exécution : c'est la ligne que `pdb` s'apprête à exécuter. Vous pouvez voir ce point d'exécution comme un curseur clignotant (juste avant la ligne indiquée).
— Point d'arrêt : il s'agit d'un endroit où nous indiquons au débogueur que nous souhaiterions examiner l'exécution du code. Pour créer un point d'arrêt, nous avons utilisé la fonction `breakpoint()` de Python. Il existe d'autres moyens.

Un remplacement de nom

Voyons maintenant un programme un peu plus utile. Il contient une erreur que peut-être vous avez déjà commise et qui n'est pas nécessairement simple à comprendre :

```
from math import sqrt

def racine_carrée(nombre_1: int | float, nombre_2: int |
↪  float) -> int | float:
    """Renvoie la somme des racines carrées des deux nombres
    ↪  entrés."""
    if nombre_1 <= 0 or nombre_2 <= 0:
        raise ValueError("nombre négatif ou nul")

    sqrt = sqrt(nombre_1)
    sqrt += sqrt(nombre_2)
    return sqrt

résultat = racine_carrée(4, 8)
print(f"Le résultat est {résultat}.")
```

Est-ce que le code vous semble bon ? Peut-être que oui, peut-être que non. Essayons de l'exécuter :

```
> python remplacement.py
Traceback (most recent call last):
  File "remplacement.py", line 12, in <module>
    résultat = racine_carrée(4, 8)
  File "remplacement.py", line 8, in racine_carrée
    sqrt = sqrt(nombre_1)
UnboundLocalError: local variable 'sqrt' referenced before
 ↪  assignment

>
```

Cette erreur est intéressante. Elle n'est pas tout à fait logique à première vue. Cette fois-ci, plaçons notre point d'arrêt juste avant d'appeler notre fonction :

```
from math import sqrt

# ...

breakpoint()
résultat = racine_carrée(4, 8)
print(f"Le résultat est {résultat}.")
```

Lançons de nouveau le programme :

```
> python remplacement.py
> d:\livre\python\part4\chap8\remplacement.py(13)<module>()
-> résultat = racine_carrée(4, 8)
(Pdb)
```

Commençons par vérifier que la fonction `sqrt` est bien définie :

```
(Pdb) sqrt
<built-in function sqrt>
(Pdb)
```

Bien, très bien jusqu'ici. Maintenant commençons à exécuter notre fonction. Si vous tapez `n`, `pdb` exécute votre fonction et se heurte à la même erreur. Ce n'est pas bien utile. Nous voudrions exécuter en détail notre fonction.

Le plus simple n'est-il pas de mettre un appel à `breakpoint()` au début de la fonction, comme nous l'avons fait plus haut ?

C'est une solution, mais il y a plus simple : `pdb` est lancé, après tout, autant en profiter. Au lieu d'utiliser la commande `next` qui va exécuter directement notre fonction, nous allons appeler `step` qui permet d'examiner la fonction sous le point d'exécution (c'est-à-dire notre fonction `racine_carrée` ici).

```
(Pdb) s
--Call--
> d:\livre\python\part4\chap8\remplacement.py(3)racine_carrée()
-> def racine_carrée(nombre_1: int | float, nombre_2: int | float) -> int |
↪  float:
(Pdb) n
> d:\livre\python\part4\chap8\remplacement.py(5)racine_carrée()
-> if nombre_1 <= 0 or nombre_2 <= 0:
(Pdb)
> d:\livre\python\part4\chap8\remplacement.py(8)racine_carrée()
-> sqrt = sqrt(nombre_1)
(Pdb)
```

La commande `step` (ou `s`) nous a permis de rentrer dans le code d'exécution d'une fonction. On se retrouve sur la ligne de la définition de la fonction et donc entrons la commande `n` pour continuer. Notez que la condition apparaît bien mais elle est fausse, donc Python n'exécute pas le code qu'elle contient.

Nous voici sur la ligne fatale qui provoque l'erreur. Nous sommes juste avant l'erreur, tâchons de la comprendre :

```
(Pdb) sqrt
<built-in function sqrt>
(Pdb) n
UnboundLocalError: local variable 'sqrt' referenced before assignment
> d:\livre\python\part4\chap8\remplacement.py(8)racine_carrée()
-> sqrt = sqrt(nombre_1)
(Pdb)
```

`pdb` a rencontré l'erreur, mais il nous propose malgré tout d'exécuter à nouveau la même ligne ou de trouver une solution. Pourquoi cette erreur ? On a vu que `sqrt` pointait bien vers la fonction `math.sqrt`...

La réponse se trouve dans la ligne que `pdb` vous affiche, mais il faut la décomposer pour la comprendre : on appelle bien la fonction `sqrt`, mais on capture le retour dans une variable... `sqrt`! Du coup, quand on veut écrire `sqrt(nombre_1)`, Python s'imagine que l'on veut appeler notre variable, alors qu'elle n'est pas encore créée.

La solution a pu vous apparaître tout de suite : changeons le nom de notre variable. De toute façon, `sqrt`, ce n'est pas un beau nom en français. Nous pourrions fermer le programme et relancer `pdb`, mais avant tout, voyons si cette correction résout bien le problème.

Vous pouvez entrer du code Python dans `pdb`. En fait, c'est ce qu'on a fait quand on voulait afficher des variables. La méthode que je vous ai montrée a plusieurs inconvénients, parmi lesquels le fait que vous ne puissiez pas examiner une variable appelée `list` ou `step`, car ce sont des noms de commandes pour le débogueur. Si vous souhaitez entrer du code Python dans `pdb`, je vous conseille d'utiliser la notation suivante : préfixez la commande d'un point d'exclamation. Comme cela, le débogueur sait que ce qui suit est du code Python et pas une commande qui lui est adressée :

```
(Pdb) !racine = sqrt(nombre_1)
(Pdb) racine
```

```
2.0
(Pdb)
```

De toute évidence, cette solution règle le problème. Quittez `pdb` (avec `q`) et modifiez le code de la fonction `racine_carrée` :

```
def racine_carrée(nombre_1: int | float, nombre_2: int |
↪  float) -> int | float:
    """Renvoie la somme des racines carrées des deux nombres
    ↪  entrés."""
    if nombre_1 <= 0 or nombre_2 <= 0:
        raise ValueError("nombre négatif ou nul")

    racine = sqrt(nombre_1)
    racine += sqrt(nombre_2)
    return racine
```

Lançons à nouveau notre programme :

```
> python remplacement.py
> d:\livre\python\part4\chap8\remplacement.py(13)<module>()
-> résultat = racine_carrée(4, 8)
(Pdb) n
> d:\livre\python\part4\chap8\remplacement.py(14)<module>()
-> print(f"Le résultat est {résultat}.")
(Pdb) n
Le résultat est 4.82842712474619.
--Return--
> d:\livre\python\part4\chap8\remplacement.py(14)<module>()
->None -> print(f"Le résultat est {résultat}.")
(Pdb) q
```

Cette fois, le code fonctionne sans erreur.

Pour conclure, ou continuer

Il existe beaucoup d'autres commandes `pdb` ; certaines que vous n'avez pas besoin de connaître, d'autres qui vous seront utiles dans des cas un peu moins fréquents. Voici le récapitulatif de celles que nous avons découvertes, ainsi que quelques autres :

- `list` ou `l` : affiche les lignes autour du point d'exécution.
- `next` ou `n` : exécute la ligne au point d'exécution, passe à la ligne suivante.
- `quit` ou `q` : quitte `pdb` et referme le programme.
- `continue` ou `c` : ferme `pdb` et continue l'exécution du programme jusqu'au prochain point d'arrêt.
- `step` ou `s` : examine le corps d'exécution d'une fonction appelée au point d'exécution.
- `return` ou `r` : se déplace à la fin de la fonction.

— `break` ou `b` : permet de poser un autre point d'arrêt.
— ENTRÉE : répète la dernière commande envoyée à `pdb`.

Pour la liste exhaustive, reportez-vous à la documentation officielle (et en français) du module : `https://docs.python.org/fr/3/library/pdb.html`

— Un débogueur comme `pdb` permet d'examiner de façon détaillée un code source à la recherche d'erreurs ou incohérences.
— On peut placer un point d'arrêt et lancer `pdb` avec la fonction native `breakpoint()` (depuis Python 3.7).

Chapitre 35

La programmation parallèle avec `threading`

Difficulté : 

Jusqu'ici , nous avons utilisé Python de façon linéaire : les instructions s'exécutaient dans l'ordre et, pour que la suivante s'exécute, celle d'avant devait être terminée.

Cependant, Python nous propose dans sa bibiliothèque standard plusieurs modules pour faire de la « programmation parallèle », c'est-à-dire que plusieurs instructions de code s'exécuteront en même temps, ou presque en même temps.

Nous allons regarder de plus près le module `threading` qui propose une interface simple pour créer des ***threads***, c'est-à-dire des portions de notre code qui seront exécutées en même temps.

Pour suivre ce chapitre, vous aurez besoin de savoir comment créer des classes et connaître les bases de l'héritage.

Création de threads

Jusqu'ici, nous avons travaillé avec de la programmation « linéaire ». Considérez le code suivant :

```
import time
print("Avant la pause...")
time.sleep(5)
print("Après la pause.")
```

Si vous exécutez ce code, sans surprise, le premier message s'affiche, puis 5 secondes plus tard, c'est au tour du second message.

Les *threads* permettent d'exécuter plusieurs instructions en même temps. On parle de « programmation parallèle » car, au lieu de développer selon un seul flux d'instructions, on développe plusieurs flux en parallèle.

Premier exemple d'un thread

Voyons un code linéaire pour commencer. Je fais appel à plusieurs fonctions que vous n'avez peut-être jamais vues, mais je commente les lignes en question plus loin :

```
import random
import sys
import time

# Répète 20 fois
for i in range(20):
    print("1", end="")
    sys.stdout.flush() # on force le message à apparaître
    attente = 0.2
    attente += random.randint(1, 60) / 100
    # attente est à présent entre 0.2 et 0.8
    time.sleep(attente)

print()
```

1. D'abord, on importe les modules `random`, `sys` et `time`, que nous allons utiliser par la suite.
2. Ensuite, on crée une boucle qui va s'exécuter 20 fois.
3. On affiche simplement le chiffre `1`. On appelle `print` en lui spécifiant qu'elle ne doit pas ajouter de saut de ligne à la fin ; donc tous nos messages apparaissent sur la même ligne. On fait appel à `sys.stdout.flush()` pour demander à Python d'afficher le chiffre tout de suite. Si vous oubliez cette seconde ligne, les chiffres n'apparaîtront qu'à la fin de l'exécution du programme.
4. On crée une variable `attente` et on la fait varier, grâce à `random`, entre `0.2` et `0.8`.

5. Enfin, on appelle `time.sleep()` qui met en pause notre programme pendant le temps d'attente que nous avons configuré (c'est-à-dire entre 0,2 et 0,8 seconde).
6. À la fin de la boucle, on affiche un saut de ligne en appelant `print` sans aucun argument.

Si vous exécutez ce code, vous devez voir apparaître 20 fois le chiffre `1` sur la même ligne mais, entre chaque chiffre, le programme se met en pause (de durée variable).

Approche parallèle

Maintenant, nous allons créer deux *threads* qui vont s'exécuter ensemble : le premier affichera des `1` sur l'écran, tandis que le second affichera des `2`. Lancés en même temps, vous devriez voir plus clairement la façon dont ils s'exécutent.

Pour créer un *thread*, il faut créer une classe qui hérite de `threading.Thread`. On peut redéfinir son constructeur et la méthode `run`.

Cette seconde méthode est appelée au lancement du *thread* et contient le code qui doit s'exécuter en parallèle du reste du programme.

Voyons un exemple :

```
import random
import sys
from threading import Thread
import time

class Afficheur(Thread):

    """Thread chargé simplement d'afficher une lettre dans la
    ↪  console."""

    def __init__(self, lettre):
        Thread.__init__(self)
        self.lettre = lettre

    def run(self):
        """Code à exécuter pendant l'exécution du thread."""
        # Répète 20 fois
        for i in range(20):
            print(self.lettre, end="")
            sys.stdout.flush()
            attente = 0.2
            attente += random.randint(1, 60) / 100
            time.sleep(attente)

        print()
```

Nous avons défini un *thread* :

— Le constructeur ne devrait pas trop vous surprendre. Il prend en paramètre la lettre à afficher (nous verrons des exemples plus loin). Il appelle le constructeur parent (`Thread.__init__(self)`) et c'est une étape importante, ne l'oubliez pas quand vous redéfinissez le constructeur de votre *thread*.
— La méthode `run` est également redéfinie. Le code qu'elle contient vous semble sans doute familier : c'est celui que nous avons utilisé dans notre exemple précédent.

Une fois encore, si vous exécutez ce code, vous obtenez... rien du tout ! Vous avez défini le *thread*, mais il nous reste à le créer. Ou plutôt, à les créer, car nous allons essayer d'en exécuter deux en même temps :

```
# Création des threads
thread_1 = Afficheur("1")
thread_2 = Afficheur("2")

# Lancement des threads
thread_1.start()
thread_2.start()

# Attend que les threads se terminent
thread_1.join()
thread_2.join()
```

1. D'abord, on crée nos deux *threads*. Les objets `Thread` sont conservés dans nos variables `thread_1` et `thread_2`. Notez qu'on passe des chiffres différents en paramètres de nos deux *threads*, pour être en mesure de les différencier quand ils commenceront à afficher les informations dans la console.
2. Ensuite, on appelle `thread_1.start()`. Cette méthode va créer un *thread* (une partie du code qui va pouvoir s'exécuter en parallèle) et exécuter la méthode `run`. Nos chiffres `1` commencent ainsi à s'afficher dans notre console. Toutefois, la méthode `start` n'attend pas que tous les chiffres soient écrits avant de retourner et on passe tout de suite à la ligne suivante.
3. C'est au tour du second *thread* d'être lancé. Les deux processus s'exécutent en même temps.
4. Enfin, on appelle la méthode `join()` sur les deux *threads*. Cette méthode bloque et ne retourne que quand ces derniers sont terminés. Si le programme se terminait pendant qu'ils tournent, les *threads* risqueraient d'être fermés brusquement.

Pour récapituler, voici le code complet :

```
import random
import sys
from threading import Thread
import time

class Afficheur(Thread):

```

```
    """Thread chargé simplement d'afficher une lettre dans la
    ↪ console."""

    def __init__(self, lettre):
        Thread.__init__(self)
        self.lettre = lettre

    def run(self):
        """Code à exécuter pendant l'exécution du thread."""
        # Répète 20 fois
        for i in range(20):
            print(self.lettre, end="")
            sys.stdout.flush()
            attente = 0.2
            attente += random.randint(1, 60) / 100
            time.sleep(attente)

        print()

# Création des threads
thread_1 = Afficheur("1")
thread_2 = Afficheur("2")

# Lancement des threads
thread_1.start()
thread_2.start()

# Attend que les threads se terminent
thread_1.join()
thread_2.join()
```

Quand vous exécutez ce programme, vous obtenez une ligne similaire à la suivante :

```
12211212121221212112211222121212211211211
```

Comme vous le constatez, les deux *threads* s'exécutent en même temps. Puisque le temps de pause est variable, parfois on a un seul chiffre `1` qui s'affiche avant un chiffre `2`, parfois on en a plusieurs. Au final, il y en a bien 20 de chaque.

Pour cette fois d'ailleurs, remarquez que `thread_1` est le plus long à s'exécuter (le dernier chiffre de la ligne est un `1` et le dernier `2` est un peu avant). Vous pouvez essayer la même chose en créant plusieurs autres *threads*, 3 ou 4 ou 5 ou plus, si vous voulez.

La programmation parallèle peut être très pratique, mais elle a aussi ses pièges. Nous allons en voir certains à présent et les méthodes qui existent pour les éviter.

La synchronisation des threads

Programmer plusieurs flux d'instructions apporte son lot de difficultés. Au premier abord, cela semble très pratique d'avoir plusieurs parties de notre code qui s'exécutent en même temps. Pendant une tâche qui peut durer longtemps (peut-être le téléchargement d'une information depuis un site Internet), on peut faire autre chose, pas seulement attendre que la ressource soit téléchargée.

Cependant, le développement risque d'être compliqué en proportion. Il vous faut garder en tête que les différents flux d'instructions sont avancés à différents points à un moment précis.

Opérations concurrentes

Considérez ce tout petit exemple :

```
1 nombre = 1
2 nombre += 1
```

C'est la deuxième ligne qui nous intéresse ici. Si vous y faites appel dans un de vos *threads* et que `nombre` est partagé par plusieurs de vos *threads*, vous pourriez avoir des résultats étranges, mais pas tout le temps. C'est tout le problème : la plupart du temps vous n'aurez aucun souci, parfois vous aurez des résultats étranges.

Disons que ce `nombre` sert à compter une information (le nombre de fois où une certaine opération s'exécute, peut-être). Si vous n'avez pas de chance, deux *threads* accéderont à ce code, mais `nombre` ne sera augmenté que de 1.

Cela est dû au fait que `nombre+=1` réalise trois choses :

1. Elle récupère la valeur de la variable `nombre`.
2. Elle y ajoute 1.
3. Elle écrit le résultat dans la variable `nombre`.

Représentez-vous ces étapes sur une feuille. Maintenant, représentez-vous les mêmes étapes pour un second *thread*.

Admettons que `thread_1` et `thread_2` s'exécutent presque en même temps :

— `thread_1` commence à exécuter les étapes 1 et 2, mais pas encore la 3.
— `thread_2`, plus rapide, exécute les trois étapes et modifie donc la valeur de `nombre`.
— Et notre `thread_1` exécute l'étape 3 et écrit le résultat dans la variable. Malheureusement, ce dernier se base sur l'ancienne valeur de `nombre` (avant que `thread_2` ne soit appelé). Au final, après l'exécution de nos deux *threads*, `nombre` n'a été incrémenté que de 1.

Comme vous le voyez ici, une ligne d'instruction très simple risque de produire des résultats inattendus si elle est appelée au même moment par différents *threads*.

Accès simultané à des ressources

Le problème est encore plus flagrant quand vous voulez accéder à des ressources (par exemple, écrire dans un même fichier) depuis différents *threads*.

Voici notre code un peu modifié pour qu'il affiche des mots complets dans la console au lieu de simples lettres. Regardez surtout la méthode `run` :

```
1  import random
2  import sys
3  from threading import Thread
4  import time
5
6  class Afficheur(Thread):
7
8      """Thread chargé simplement d'afficher un mot dans la
       ↪  console."""
9
10     def __init__(self, mot):
11         super().__init__()
12         self.mot = mot
13
14     def run(self):
15         """Code à exécuter pendant l'exécution du thread."""
16         # Répète 5 fois
17         for i in range(5):
18             for lettre in self.mot:
19                 print(lettre, end="")
20                 sys.stdout.flush()
21                 attente = 0.2
22                 attente += random.randint(1, 60) / 100
23                 time.sleep(attente)
24
25         print()
26
27 # Création des threads
28 thread_1 = Afficheur("canard")
29 thread_2 = Afficheur("TORTUE")
30
31 # Lancement des threads
32 thread_1.start()
33 thread_2.start()
34
35 # Attend que les threads se terminent
36 thread_1.join()
37 thread_2.join()
```

— On veut afficher des mots au lieu de lettres ; le constructeur est donc modifié en conséquence.

— On ne boucle que 5 fois, ce qui sera suffisant pour comprendre l'exemple.
— Pour chaque occurrence du mot, on boucle sur chaque lettre, l'affiche et fait une pause.

Et quand vous exécutez ce code, vous devez voir quelque chose comme ce qui suit :

```
cTORanaTUrEdcTaOnRarTdUcEanTaOrRdTcUaEnTaORrdTcanUaErdTORTUE
```

Comme vous le voyez, nos mots sont complètement mélangés, ce qui n'est pas bien surprenant. Vous pouvez toujours suivre les parties en majuscules ou minuscules et vérifier que les mots s'affichent bien mais, puisque nous écrivons sur la même ressource partagée (la console, ici), le résultat s'affiche mélangé.

Les verrous à la rescousse

Il existe plusieurs moyens de « synchroniser » nos *threads*, c'est-à-dire de faire en sorte qu'une partie du code ne s'exécute que si personne n'utilise la ressource partagée. Le mécanisme de synchronisation le plus simple est le verrou (*lock* en anglais).

C'est un objet proposé par `threading` qui est extrêmement simple à utiliser : au début des instructions qui utilisent notre ressource partagée, on pose un verrou qui bloque l'accès à cette dernière. Si un autre *thread* veut faire appel à cette ressource, il doit patienter jusqu'à ce qu'elle soit libérée.

Plutôt qu'un long discours, je vous propose notre code légèrement modifié pour utiliser les verrous.

```
import random
import sys
from threading import Thread, RLock
import time

verrou = RLock()

class Afficheur(Thread):

    """Thread chargé simplement d'afficher un mot dans la
    ↪  console."""

    def __init__(self, mot):
        super().__init__()
        self.mot = mot

    def run(self):
        """Code à exécuter pendant l'exécution du thread."""
        # Répète 5 fois
        for i in range(5):
            with verrou:
                for lettre in self.mot:
```

```
                    print(lettre, end="")
                    sys.stdout.flush()
                attente = 0.1
                attente += random.randint(1, 2) / 2
                time.sleep(attente)

# Création des threads
thread_1 = Afficheur("canard")
thread_2 = Afficheur("TORTUE")

# Lancement des threads
thread_1.start()
thread_2.start()

# Attend que les threads se terminent
thread_1.join()
thread_2.join()
```

Quelques petites choses ont été modifiées dans le code précédent, mais c'est surtout pour voir un résultat concret.

1. On importe `RLock` du module `threading` ;
2. On crée un verrou.
3. Dans notre méthode `run`, on verrouille une partie de notre *thread*.

```
with verrou:
    for lettre in self.mot:
        print(lettre, end="")
        sys.stdout.flush()
        attente = 0.1
        attente += random.randint(1, 2) / 2
        time.sleep(attente)
```

On utilise là encore un gestionnaire de contexte pour indiquer quand poser le verrou. Ce dernier se débloque à la fin du bloc `with`.

La partie verrouillée de notre code ne s'exécute que dans un *thread* à la fois.

1. D'abord, `thread_1` est lancé. Il pose le verrou et commence à afficher les lettres de son mot (« `canard` »).
2. `thread_2` est lancé entre-temps, mais il se bloque au moment d'afficher son propre mot, car le verrou est détenu par `thread_1`. Ce n'est que quand ce dernier le relâche (à la fin du bloc `with`) que `thread_2` peut continuer à s'exécuter.
3. ... Et ainsi de suite jusqu'à la fin des deux *threads*.

Si vous exécutez ce code, vous obtenez un résultat similaire au suivant :

```
canardcanardTORTUETORTUEcanardcanardcanardTORTUETORTUETORTUE
```

Cette fois, les mots ne sont plus mélangés, mais le reste du code s'exécute bien en parallèle (notez que les mots apparaissent dans un ordre aléatoire, même si il y en a bien 5 de chaque).

Si vous testez ce code, vous pouvez très bien voir un mot apparaître 5 fois, puis l'autre mot apparaître à son tour 5 fois. Le système d'exploitation décide de l'ordre de l'exécution... et vous pourriez avoir l'impression de perdre l'avantage de la programmation parallèle. Si c'est le cas, essayez de lancer ce code plusieurs fois, ou changez les pauses et le nombre de fois que le mot est affiché.

Il existe d'autres méthodes de synchronisation et la programmation parallèle en tant que telle mérite plus un livre entier qu'un chapitre d'introduction. Vous avez pu cependant voir ici les bases de ce type de programmation. Si vous voulez plus d'informations sur les mécanismes de synchronisation (ainsi que d'autres informations générales sur les *threads*), reportez-vous à la documentation officielle du module `threading` : `https://docs.python.org/fr/3/library/threading.html`

En résumé

— Il existe plusieurs mécanismes de programmation parallèle, dont les *threads* proposés dans le module `threading` de la bibliothèque standard.
— Créer un *thread* se fait en définissant une classe héritée de `threading.Thread` et en appelant sa méthode `start`.
— On peut utiliser les verrous *(locks)* pour synchroniser nos *threads* et faire en sorte que certaines parties de notre code s'exécutent bien à la suite des autres.

Chapitre 36

La programmation asynchrone avec `asyncio`

Difficulté : 

Nous avons vu au chapitre précédent une solution pour faire tourner plusieurs morceaux de code de façon simultanée, ou du moins, presque simultanée. Je vous propose dans ce chapitre la découverte d'une autre façade de la programmation parallèle en Python : avec le module `asyncio` et les mots-clés `async` et `await`.

Autant vous prévenir tout de suite : ce chapitre sera un peu plus théorique et risque de sembler complexe. Il est facultatif, comme tous les chapitres dans cette section, mais la façade asynchrone en Python est, à mon sens, suffisamment utile pour justifier un peu de temps et d'efforts.

La programmation asynchrone

Qu'est-ce que la programmation asynchrone ?

Python nous propose de nombreuses approches possibles pour faire de la programmation parallèle, notamment la répartition en plusieurs processus (`multiprocess`), la répartition sur plusieurs *threads* (`threading`) et la séparation en tâches (`asyncio`). C'est ce dernier que nous allons découvrir dans ce chapitre ; une petite partie, pour tout dire, ce sera déjà beaucoup à retenir.

Adoptons la définition simpliste suivante : **`asyncio` permet de découper son programme en tâches, qui s'exécutent ensuite en parallèle.**

Cela vous aide-t-il ? Non, sans doute pas. Une analogie pourrait vous éclairer : on vous donne une certaine quantité de devoirs à faire pour le lendemain. Vous avez un exercice à rendre en maths, un questionnaire en français, une carte à dessiner pour le cours de géographie... et peut-être d'autres choses. Confrontés à cette montagne de devoirs, vous adoptez probablement une approche synchrone : vous faites chaque devoir, un par un, surtout pas plusieurs devoirs en même temps pour rester concentré sur une matière en particulier. Si l'exercice de maths demande 30 minutes, celui de français 15 minutes, celui de géographie 45 minutes, vous en avez donc pour une heure et demie. C'était simple ce soir.

Maintenant, dans l'esprit asynchrone, ces différents exercices seraient des tâches. Au lieu de les faire les unes à la suite des autres, votre ordinateur les commence toutes d'un coup. Il va répondre à une question sur l'exercice de maths, une autre sur l'exercice de français, il va tracer une ligne sur la carte à rendre en géographie... et revenir aux maths. Au final, en dépit du fait que chaque tâche prend exactement autant de temps, procéder de cette façon va réduire le temps complet de vos devoirs... pour un ordinateur. J'insiste sur le fait que pour un humain, ce n'est pas nécessairement la meilleure approche, mais cette comparaison illustre le concept des tâches et celui de travail en parallèle, un élément essentiel pour la suite de ce chapitre.

Vous n'avez pas fini de voir des métaphores dans ce chapitre ; si vous n'aimez pas celle-là, vous pouvez inventer la vôtre. Le principe reste assez similaire et assez facile à transposer aux tâches ménagères, à l'assistant qui doit faire plusieurs choses à la fois, à la joueuse d'échecs professionnelle...

Retour au code

Nous avons pris un exemple de tâche, mais qu'est-ce qu'une tâche en termes de code ? C'est une opération à effectuer qui peut prendre plus ou moins longtemps. Cela ne vous dit-il rien ?

Une tâche n'est ni plus ni moins qu'une fonction. La syntaxe de sa définition est presque identique (nous allons voir ça assez vite) et son contenu est assez semblable également.

Cependant, ces fonctions ont une particularité importante, qui fait toute la différence : elles savent se mettre en pause. Quand cela se produit, Python sait qu'il peut commencer à exécuter la tâche suivante. Si la seconde tâche se met en pause, Python cherche la tâche suivante à exécuter... ainsi de suite. Après un moment, la première tâche va indiquer qu'elle est prête à continuer et Python va lui redonner le droit de s'exécuter jusqu'à sa prochaine pause.

Pause ? La tâche s'arrête et ne fait rien ; quel en est l'intérêt ?

Un programme peut attendre pour de nombreuses raisons. Le plus souvent, une tâche se met en pause pour une raison précise et attend qu'une ressource soit disponible. Par exemple : une tâche (fonction) pourrait demander de télécharger un gros fichier depuis Internet. Elle se met en pause pendant le téléchargement (permettant à d'autres fonctions de s'exécuter sans les bloquer), mais quand le fichier est disponible, elle indique qu'elle voudrait continuer à s'exécuter. Nous allons voir cet exemple traduit en code dans une autre section ; il fait appel à plusieurs mécanismes à assimiler.

Pour l'heure, nous allons prendre un exemple assez simple : un programme qui crée trois tâches. Nous allons lancer ces trois tâches... qui vont afficher deux messages à l'écran. Et faire une pause d'une seconde entre chaque affichage. Cela vous permettra de comprendre la différence entre le synchrone et l'asynchrone.

Version synchrone

Commençons par voir une version synchrone, normale et facile à lire. Vous devriez pouvoir deviner ce qu'elle fait avant même d'exécuter le code Python :

```
import time

def tâche(numéro: int):
    print(f"Tâche {numéro} : avant de dormir...")
    time.sleep(1) # Met en pause pendant 1 seconde
    print(f"Tâche {numéro} : je suis réveillée et j'ai fini.")

def main():
    tâche(1)
    tâche(2)
    tâche(3)

t1 = time.time()
main()
t2 = time.time()
durée = t2 - t1
print(f"Exécution en {durée:.2f} secondes.")
```

— Nous commençons par définir une fonction, `tâche`, qui sera appelée pour nos trois tâches.
— Nous définissons une fonction, `main`, qui se charge de créer nos trois tâches. On appelle la fonction `tâche` avec des paramètres différents, juste pour les différencier et comprendre la logique d'exécution.
— On mesure le temps d'exécution, juste pour voir ce qu'on gagne au final.

Seriez-vous surpris par le résultat ?

```
Tâche 1 : avant de dormir...
Tâche 1 : je suis réveillée et j'ai fini.
Tâche 2 : avant de dormir...
Tâche 2 : je suis réveillée et j'ai fini.
Tâche 3 : avant de dormir...
Tâche 3 : je suis réveillée et j'ai fini.
Exécution en 3.02 secondes.
```

Nos trois tâches se sont exécutées l'une après l'autre. Le temps total d'exécution est de 3 secondes (un peu plus), car chaque tâche a attendu une seconde. Notez donc que la tâche 2 ne se lance qu'après une seconde. La tâche 3 ne se lance qu'après 2 secondes.

Version asynchrone

Maintenant, analysons le même exemple, modifié pour que les tâches s'exécutent de façon asynchrone. Nous allons passer plus de temps sur le code, qui comporte beaucoup de choses nouvelles.

```
import asyncio
import time

async def tâche(numéro: int):
    print(f"Tâche {numéro} : avant de dormir...")
    await asyncio.sleep(1) # La tâche se met en pause, indique
    ↪  qu'on peut en exécuter une autre
    print(f"Tâche {numéro} : je suis réveillée et j'ai fini.")

async def main():
    await asyncio.gather(tâche(1), tâche(2), tâche(3))

t1 = time.time()
asyncio.run(main())
t2 = time.time()
durée = t2 - t1
print(f"Exécution en {durée:.2f} secondes.")
```

Si vous lancez ce code, vous devez obtenir un résultat similaire au suivant :

```
Tâche 1 : avant de dormir...
Tâche 2 : avant de dormir...
```

```
Tâche 3 : avant de dormir...
Tâche 1 : je suis réveillée et j'ai fini.
Tâche 2 : je suis réveillée et j'ai fini.
Tâche 3 : je suis réveillée et j'ai fini.
Exécution en 1.02 secondes.
```

Avant de plonger dans le code, il est important de comprendre ce qui s'est passé.

- Cette fois, Python a lancé les trois tâches, l'une après l'autre. En vérité, il a lancé la tâche 1 tout de suite... quand la tâche 1 a indiqué qu'elle ne faisait plus rien, la tâche 2 s'est lancée. Quand la tâche 2 a indiqué qu'elle ne faisait plus rien, la tâche 3 s'est lancée.
- Quand la tâche 3 a indiqué qu'elle ne faisait plus rien, Python a attendu. La tâche 1 a fini par indiquer qu'elle pouvait continuer. Puis la tâche 2, puis la tâche 3.
- Résultat : en exécutant ces tâches de façon asynchrone, notre programme dure un peu plus d'une seconde, au lieu de trois.

Si ce n'est pas bien clair, reprenez-le code, ajoutez des informations à afficher (vous pouvez le faire même sans comprendre tout le code, à ce stade).

Il n'y a pas moins de quatre nouvelles choses dans ce code. Prenons-les dans l'ordre :

- Notez d'abord que la définition des fonctions `tâche` et `main` est un peu différente. On ajoute le mot-clé `async` devant. Il indique à Python que la fonction contenue est censée être une tâche. C'est valable aussi pour `main`, qui doit lancer nos tâches et est donc aussi considérée comme telle.
- Dans la fonction `tâche`, on trouve une ligne étrange : `await asyncio.sleep(1)`. Cette ligne remplace notre `time.sleep(1)`. Non seulement elle indique que la tâche doit se mettre en pause une seconde, mais elle signale à Python qu'il peut exécuter une autre tâche en attendant. C'est pourquoi il lance toutes les tâches presque instantanément. Le mot-clé `await` indique : *« mettre la fonction en pause ici et reprendre (quand je te le dirai) »*.
- Dans notre fonction `main`, nous avons une ligne encore plus bizarre : `await asyncio.gather(tâche(1), tâche(2), tâche(3))`. Elle va créer nos trois tâches asynchrones (la fonction `tâche` avec trois paramètres différents), indiquer qu'elles doivent être exécutées et attendre qu'elles se lancent.
- Enfin, un peu plus bas, nous lançons `main` comme une tâche avec `asyncio.run(main())`. Elle lance à son tour trois instances de la fonction `tâche` et attend qu'elles se soient exécutées.
- `asyncio.run` bloque donc pendant 1 seconde, pas 3. C'est le plus important à observer et il est également important de comprendre pourquoi. Si le code vous semble obscur, concentrez-vous sur le résultat, vous aurez de quoi pratiquer le code dans ce chapitre.

Pour résumer, nous avons donc vu deux fonctions du module `asyncio` : `gather`, qui lance des tâches, et `run`, qui lance la tâche initiale. Nous avons aussi vu les mots-clés `async` (avant la définition d'une fonction considérée comme une tâche) et `await` (indiquant que la tâche doit se mettre en pause).

Pourquoi passer par la fonction `main` ? N'aurait-on pas mieux fait de lancer les trois tâches directement ?

Il s'agit d'une contrainte un peu technique : `asyncio.run` lance une tâche (c'est-à-dire une fonction). On ne peut pas en lancer plusieurs. On passe donc par une tâche (fonction asynchrone) intermédiaire dont le rôle est simplement de lancer nos trois tâches.

Pourquoi utiliser `async` ? Python peut deviner que cette fonction est une tâche, après tout, vu comment elle est exécutée.

C'est en vérité une bonne question. Sachez qu'il y a des développeurs qui insistent sur le fait que cette syntaxe alourdit trop le code. Elle a pourtant un avantage : elle est explicite. Dès qu'on voit un `async`, on sait que ce qui suit est une tâche, pas une fonction standard de Python.

Pour essayer de comprendre un peu mieux la différence, ouvrons l'interpréteur :

```
>>> import asyncio
>>> async def tâche(numéro: int):
...     print(f"Tâche {numéro} : avant de dormir...")
...     await asyncio.sleep(1) # La tâche se met en pause, indique qu'on peut en exécuter une autre
...     print(f"Tâche {numéro} : je suis réveillée et j'ai fini.")
...
>>> tâche
<function tâche at 0x012685D0>
>>> t1 = tâche(1)
>>> t1
<coroutine object tâche at 0x03B41DF8>
>>> asyncio.run(t1)
Tâche 1 : avant de dormir...
Tâche 1 : je suis réveillée et j'ai fini.
>>>
```

On recrée la fonction `tâche` (fonction asynchrone, on place le mot-clé `async` devant `def`).

On regarde ensuite la référence de la fonction. Elle semble normale à première vue.

On l'appelle en capturant le résultat dans la variable `t1`. On examine `t1` et... surprise ! Si `tâche` avait été une fonction standard de Python, `t1` contiendrait son résultat. Or, `t1` contient un objet bizarre : `<coroutine object tâche at ...>`.

Coroutine ?

On aborde le vocabulaire un peu particulier que Python (et d'autres langages) utilisent. J'ai parlé de tâches jusqu'ici, car c'était plus simple à comprendre. En vérité, une tâche asynchrone est appelée en Python une **coroutine**. Ce nom pourrait vous sembler familier... du chapitre sur les générateurs ? Ce n'est pas une coïncidence ; l'origine des coroutines en Python vient des générateurs, mais nous n'allons pas rentrer dans le détail historique.

`t1` contient donc une représentation de la tâche 1. La fonction `tâche` s'est-elle exécutée ? Si vous choisissez « non », vous avez gagné. La fonction ne s'est pas exécutée à ce stade. `t1` contient la référence vers la tâche asynchrone mais c'est tout pour l'heure.

On donne directement cette référence à `asyncio.run`. Cette fonction va lancer notre tâche et va attendre qu'elle soit finie. Le résultat est peu impressionnant avec une seule tâche : la fonction met une seconde pour s'exécuter.

Puisque nous n'avons lancé qu'une tâche, on ne voit pas trop l'intérêt de la programmation asynchrone dans cet exemple, mais il devrait vous permettre de comprendre un peu mieux ce que sont les tâches asynchrones (coroutines) et ce que fait `asyncio.run`.

Pourquoi une tâche se mettrait-elle en attente ?

Est-ce que notre programme Python n'est pas censé tourner à 100 % de ses capacités tout le temps jusqu'à la fin du monde, ou du moins, la fin de notre ordinateur ?

Réfléchissons à l'utilité. Pourquoi mettre en pause volontairement notre programme ?

La réponse est qu'on met assez rarement notre programme en pause pour rien (juste se mettre en pause X secondes), comme nous l'avons fait. Considérez par exemple notre tâche chargée de télécharger un fichier d'Internet. Le fait qu'elle paralyse le reste du programme pendant le téléchargement du fichier (ce qui pourrait prendre plusieurs minutes) est assez problématique. Avec la programmation asynchrone, on peut indiquer à Python que la tâche se met en pause jusqu'au moment où le fichier est téléchargé. Le programme ne bloque plus. On peut faire plusieurs choses presque simultanément et gagner en temps.

N'est-ce pas ce qu'on fait avec `threading` ?

Une précision s'impose : `threading` n'est pas conçu dans le même esprit. Plusieurs parties de votre code pourraient s'exécuter en même temps. Vous n'avez pas forcément un gros contrôle sur l'ordre d'exécution et vous devez faire attention à des choses complexes, comme l'utilisation parallèle des ressources. Utiliser `asyncio` et englober des tâches dans des fonctions assure que, à un moment donné, une seule tâche tourne : il n'y a jamais plusieurs fonctions qui s'exécutent en même temps, ce qui résout bon nombre des problèmes soulevés par `threading`. Cela ne veut pas dire que `asyncio` soit

meilleur que `threading` : ces deux modules abordent le problème de façon différente et trouvent des solutions différentes. L'une ou l'autre solution pourrait être plus appropriée à votre problème et plus rapide.

Il est temps de voir quelques cas concrets du système asynchrone en action !

La puissance des tâches asynchrones

Quels sont les cas où notre programme doit attendre longtemps et ne peut rien faire ? Ces cas existent-ils ? Nous avons vu un exemple dans la partie précédente : le programme se met en pause, sans rien faire.

On voudrait, au contraire, qu'il fasse quelque chose d'utile. Y a-t-il des cas plus fréquents ? Votre programme s'exécute-t-il plus vite que d'autres opérations sur votre ordinateur ?

Eh bien... pour commencer, votre programme pourrait « perdre du temps » à attendre que vous lui donniez des informations, comme des entrées utilisateur, et faire autre chose d'utile en attendant. Sachez qu'il perd un peu de temps aussi en accédant à votre disque dur et qu'il pourrait travailler sur autre chose durant ce créneau. Ou bien, admettons que votre programme tente d'accéder à Internet. La vitesse de téléchargement de certains est incroyablement rapide, mais la vitesse d'écriture en mémoire est souvent encore plus rapide. Donc votre programme est plus rapide que beaucoup de choses (vous y compris, sans vouloir offenser personne). C'est là que l'asynchrone entre en jeu : nous n'avons pas discuté le terme jusqu'ici, mais le module pourrait se décomposer en `async` (asynchrone) et `IO` (entrées et sorties). Autrement dit, au lieu de bloquer en attendant des entrées ou sorties, on pourrait faire quelque chose d'utile en attendant que ces informations soient disponibles.

Nous allons ici regarder un cas on ne peut plus pratique : le téléchargement de gros fichiers depuis Internet (je prends l'exemple de musiques libres). La notion de « gros » fichiers dépend évidemment de votre connexion.

Nous allons voir deux approches, code à l'appui. La version synchrone, que vous devriez comprendre et peut-être coder, télécharge les fichiers les uns à la suite des autres et les écrit sur votre disque en bloquant complètement l'exécution de votre programme. Dans la version asynchrone, dès qu'un fichier est téléchargé, on l'écrit sur le disque ; on peut donc télécharger plusieurs fichiers en même temps (un morceau par-ci, un morceau par-là).

Dépendances à installer

Nous devons installer plusieurs dépendances pour essayer tout cela. Je vous conseille vivement d'installer un environnement indépendant (`virtualenv`, voir page 477). Si vous avez besoin d'aide sur l'ouverture de la console pour l'installation des dépendances avec `pip`, vous trouverez plus d'informations à la page 473.

Une fois la console ouverte (avec votre environnement indépendant actif, si vous en avez créé un), entrez la commande suivante :

```
python -m pip install requests aiohttp cchardet aiodns
↪  aiofiles
```

Cette commande installe de nombreuses bibliothèques ; raison de plus pour les isoler dans un environnement indépendant.

La bibliothèque `requests` que nous allons utiliser à présent permet de télécharger une ressource depuis Internet, de façon synchrone. On pourrait le faire autrement mais autant le faire avec `requests`, qui a de nombreux avantages. Les autres bibliothèques nous seront utiles pour la version asynchrone.

Enfin, je compte m'appuyer sur le même exemple pour les deux scripts : nous voudrions télécharger une liste de fichiers d'un site de musique libre, appelé `TeknoAxe`. Ce choix est un peu arbitraire, je voulais des « gros » fichiers à télécharger ; adaptez en fonction de ce que vous voulez.

Version synchrone

Bien, commençons par analyser la version synchrone. Elle est assez facile à coder (`requests` n'est pas bien dur à utiliser). Voici le code que je vous propose :

```
import time

import requests

def télécharger(session, fichier):
    """Télécharge le fichier indiqué sur TeknoAxe, retourne
    ↪  les octets (bytes).

    Arguments :
        session (Session) : la session (requests) à utiliser.
        fichier (str) : le fichier à télécharger.

    Renvoie :
        octets (bytes) : les octets reçus, ou b"" si erreur.

    """
    url =
    ↪  "http://www.teknoaxe.com/Music/mobile_direct_download.php"
    print(f"{fichier} : on se prépare à télécharger...")
    réponse = session.get(url, params={"file": fichier})
    if réponse.status_code != 200: # C'est une erreur
        print(f"{fichier} : erreur de téléchargement.")
        return b""
```

```
22
23      octets = réponse.content
24      print(f"{fichier} : téléchargé {len(octets)} octets")
25      return octets
26
27  def enregistrer(session, fichier):
28      """Télécharge et enregistre le résultat dans un fichier.
29
30      Arguments :
31          session (Session) : la session (requests) à utiliser.
32          fichier (str) : le nom du fichier.
33
34      Exemple :
35          enregistrer(session, "Sheltered In Subdued Rain.mp3")
36          # Cela enverra la requête
    ↪   http://www.teknoaxe.com/Music/mobile_direct_download.php
    ↪   ?file=Sheltered_In_Subdued_Rain.mp3
37
38      """
39      fichier_sans_espaces = fichier.replace(" ", "_")
40      print(f"{fichier} : demande le téléchargement...")
41      octets = télécharger(session, fichier_sans_espaces)
42      if octets: # Inutile d'enregistrer un fichier vide
43          with open(fichier, "wb") as fichier_à_enregistrer:
44              fichier_à_enregistrer.write(octets)
45              print(f"{fichier} : écrit {len(octets)} octets
                ↪  dans le fichier {fichier}.")
46
47  def tout_télécharger():
48      """Télécharge et enregistre tous les fichiers indiqués."""
49      fichiers = [
50          "Sheltered In Subdued Rain.mp3",
51          "Metal and Medeival.mp3",
52          "Isle of Doom.mp3",
53          "Wayward Ghouls.mp3",
54          "Figuring Out the Technicalities.mp3",
55          "Until Your Engine Stops II.mp3",
56          "Disco Attempt_1.mp3"
57      ]
58
59      with requests.Session() as session:
60          for fichier in fichiers:
61              enregistrer(session, fichier)
62
63  t1 = time.time()
64  tout_télécharger()
```

```
t2 = time.time()
print(f"Exécution en {t2 - t1:.2f} secondes.")
```

Le code n'est pas trop difficile. Partez du bas pour comprendre la logique du flux d'instructions : on lui donne 7 fichiers à télécharger. La fonction `tout_télécharger` appelle 7 fois la fonction `enregistrer`. Cette dernière appelle `télécharger` pour chaque musique et écrit le retour du téléchargement dans un fichier sur votre disque. On utilise le mode d'écriture en octets (`wb`) car on veut écrire le fichier tel qu'on le reçoit, pas une version lisible pour nous autres êtres humains.

Prenez le temps de relire ce code et de bien le comprendre. Nous allons bâtir dessus dans la prochaine section pour notre version asynchrone. En attendant, voici le retour que j'obtiens, chez moi, quand j'exécute le programme :

```
Sheltered In Subdued Rain.mp3 : demande le téléchargement...
Sheltered_In_Subdued_Rain.mp3 : on se prépare à télécharger...
Sheltered_In_Subdued_Rain.mp3 : téléchargé 8602236 octets
Sheltered In Subdued Rain.mp3 : écrit 8602236 octets dans le fichier Sheltered In Subdued Rain.mp3.
Metal and Medeival.mp3 : demande le téléchargement...
Metal_and_Medeival.mp3 : on se prépare à télécharger...
Metal_and_Medeival.mp3 : téléchargé 9722360 octets
Metal and Medeival.mp3 : écrit 9722360 octets dans le fichier Metal and Medeival.mp3.
Isle of Doom.mp3 : demande le téléchargement...
Isle_of_Doom.mp3 : on se prépare à télécharger...
Isle_of_Doom.mp3 : téléchargé 4121701 octets
Isle of Doom.mp3 : écrit 4121701 octets dans le fichier Isle of Doom.mp3.
Wayward Ghouls.mp3 : demande le téléchargement...
Wayward_Ghouls.mp3 : on se prépare à télécharger...
Wayward_Ghouls.mp3 : téléchargé 8983613 octets
Wayward Ghouls.mp3 : écrit 8983613 octets dans le fichier Wayward Ghouls.mp3.
Figuring Out the Technicalities.mp3 : demande le téléchargement...
Figuring_Out_the_Technicalities.mp3 : on se prépare à télécharger...
Figuring_Out_the_Technicalities.mp3 : téléchargé 9641916 octets
Figuring Out the Technicalities.mp3 : écrit 9641916 octets dans le fichier Figuring Out the Technicalities.mp3.
Until Your Engine Stops II.mp3 : demande le téléchargement...
Until_Your_Engine_Stops_II.mp3 : on se prépare à télécharger...
Until_Your_Engine_Stops_II.mp3 : téléchargé 10282433 octets
```

```
Until Your Engine Stops II.mp3 : écrit 10282433 octets dans
 ↪  le fichier Until Your Engine Stops II.mp3.
Disco Attempt_1.mp3 : demande le téléchargement...
Disco_Attempt_1.mp3 : on se prépare à télécharger...
Disco_Attempt_1.mp3 : téléchargé 8136202 octets
Disco Attempt_1.mp3 : écrit 8136202 octets dans le fichier
 ↪  Disco Attempt_1.mp3.
Exécution en 195.50 secondes.
```

Ma connexion n'est pas des meilleures : 195 secondes (plus de 3 minutes) pour télécharger 7 fichiers de 8 à 10 Mo en moyenne.

Ce qu'il faut noter ici, c'est que tout s'exécute dans l'ordre : nos fichiers sont téléchargés et écrits les uns à la suite des autres. Notre temps d'exécution total est donc la somme du temps d'exécution de chaque fichier téléchargé. Voyez-vous le problème ?

Version asynchrone

Étudions à présent notre version asynchrone. Le code change un peu et nous allons voir plus en détail ce qu'il fait. La première chose, c'est que nous n'utilisons plus `requests` : `requests` a une interface synchrone qui bloquerait bien trop dans notre cas. On utilise donc deux bibliothèques : `aiohttp`, pour obtenir des données depuis Internet de façon asynchrone, et `aiofiles`, pour lire et écrire sur le disque dur de façon asynchrone.

Notre but est simple : au lieu de télécharger les fichiers les uns à la suite des autres, on voudrait lancer tous les téléchargements d'un coup et réduire ainsi notre temps d'exécution total. Vous n'allez sans doute pas pouvoir deviner ce code (sauf si vous allez faire un tour du côté de la documentation des bibliothèques tierces utilisées). Je commente ma solution en détail juste après :

```
import time

import aiofiles
import aiohttp
import asyncio

async def télécharger(session, fichier):
    """Télécharge le fichier indiqué sur TeknoAxe, retourne
     ↪  les octets (bytes).

    Arguments :
        session (Session) : la session aiohttp à utiliser.
        fichier (str) : le fichier à télécharger.

    Renvoie :
```

```
        octets (bytes) : les octets reçus, ou b"" si erreur.

    """
    url =
    ↪ "http://www.teknoaxe.com/Music/mobile_direct_download.php"
    print(f"{fichier} : on se prépare à télécharger...")
    async with session.get(url, params={"file": fichier}) as
    ↪ réponse:
        if réponse.status != 200: # C'est une erreur
            print(f"{fichier} : erreur de téléchargement.")
            return b""

        octets = await réponse.read()
        print(f"{fichier} : téléchargé {len(octets)} octets")
        return octets

async def enregistrer(session, fichier):
    """Télécharge et enregistre le résultat dans un fichier.

    Arguments :
        session (Session) : la session aiohttp à utiliser.
        fichier (str) : le nom du fichier.

    Exemple :
        enregistrer(session, "Sheltered In Subdued Rain.mp3")
        # Cela enverra la requête
        http://www.teknoaxe.com/Music/
        mobile_direct_download.php
        ?file=Sheltered_In_Subdued_Rain.mp3

    """
    fichier_sans_espaces = fichier.replace(" ", "_")
    print(f"{fichier} : demande le téléchargement...")
    octets = await télécharger(session, fichier_sans_espaces)
    if octets: # Inutile d'enregistrer un fichier vide
        async with aiofiles.open(fichier, "wb") as
        ↪ fichier_à_enregistrer:
            await fichier_à_enregistrer.write(octets)
            print(f"{fichier} : écrit {len(octets)} octets
            ↪ dans le fichier {fichier}.")

async def tout_télécharger():
    """Télécharge et enregistre tous les fichiers indiqués."""
    fichiers = [
        "Sheltered In Subdued Rain.mp3",
        "Metal and Medeival.mp3",
```

```
        "Isle of Doom.mp3",
        "Wayward Ghouls.mp3",
        "Figuring Out the Technicalities.mp3",
        "Until Your Engine Stops II.mp3",
        "Disco Attempt_1.mp3"
    ]

    tâches = []
    async with aiohttp.ClientSession() as session:
        for fichier in fichiers:
            tâches.append(enregistrer(session, fichier))
        await asyncio.gather(*tâches)

t1 = time.time()
asyncio.run(tout_télécharger())
t2 = time.time()
print(f"Exécution en {t2 - t1:.2f} secondes.")
```

Avant d'étudier le code, voyons le résultat (là encore, chez moi) :

```
Sheltered In Subdued Rain.mp3 : demande le téléchargement...
Sheltered_In_Subdued_Rain.mp3 : on se prépare à
 ↪  télécharger...
Metal and Medeival.mp3 : demande le téléchargement...
Metal_and_Medeival.mp3 : on se prépare à télécharger...
Isle of Doom.mp3 : demande le téléchargement...
Isle_of_Doom.mp3 : on se prépare à télécharger...
Wayward Ghouls.mp3 : demande le téléchargement...
Wayward_Ghouls.mp3 : on se prépare à télécharger...
Figuring Out the Technicalities.mp3 : demande le
 ↪  téléchargement...
Figuring_Out_the_Technicalities.mp3 : on se prépare à
 ↪  télécharger...
Until Your Engine Stops II.mp3 : demande le téléchargement...
Until_Your_Engine_Stops_II.mp3 : on se prépare à
 ↪  télécharger...
Disco Attempt_1.mp3 : demande le téléchargement...
Disco_Attempt_1.mp3 : on se prépare à télécharger...
Isle_of_Doom.mp3 : téléchargé 4121701 octets
Isle of Doom.mp3 : écrit 4121701 octets dans le fichier Isle
 ↪  of Doom.mp3.
Sheltered_In_Subdued_Rain.mp3 : téléchargé 8602236 octets
Sheltered In Subdued Rain.mp3 : écrit 8602236 octets dans le
 ↪  fichier Sheltered In Subdued Rain.mp3.
Metal_and_Medeival.mp3 : téléchargé 9722360 octets
Metal and Medeival.mp3 : écrit 9722360 octets dans le fichier
 ↪  Metal and Medeival.mp3.
```

```
Figuring_Out_the_Technicalities.mp3 : téléchargé 9641916 octets
Figuring Out the Technicalities.mp3 : écrit 9641916 octets dans le fichier Figuring Out the Technicalities.mp3.
Disco_Attempt_1.mp3 : téléchargé 8136202 octets
Disco Attempt_1.mp3 : écrit 8136202 octets dans le fichier Disco Attempt_1.mp3.
Wayward_Ghouls.mp3 : téléchargé 8983613 octets
Wayward Ghouls.mp3 : écrit 8983613 octets dans le fichier Wayward Ghouls.mp3.
Until_Your_Engine_Stops_II.mp3 : téléchargé 10282433 octets
Until Your Engine Stops II.mp3 : écrit 10282433 octets dans le fichier Until Your Engine Stops II.mp3.
Exécution en 70.40 secondes.
```

Trois choses sont à retenir ici :

- Python lance tous les téléchargements les uns à la suite des autres, sans attendre qu'ils soient terminés.
- Dès qu'un téléchargement est fini, il est écrit sur le disque dur. Notez ici que le premier téléchargement à s'être terminé n'est pas le premier lancé.
- Enfin, remarquez le temps d'exécution : 70 secondes au lieu de 195. C'est bien plus rapide en asynchrone !

Examinons le code plus en détail à présent :

- La structure générale (les fonctions `télécharger`, `enregistrer` et `tout_télécharger`) reste identique. Cela vous permet de comparer.
- On importe plus de bibliothèques cette fois : `asyncio`, `aiohttp` pour la lecture de nos ressources sur Internet et `aiofiles` pour l'écriture sur le disque des fichiers téléchargés.
- On indique à Python que nos fonctions `télécharger`, `enregistrer` et `tout_télécharger` sont des coroutines (des tâches asynchrones) en plaçant le mot-clé `async` devant leur définition.
- Dans `télécharger`, vous remarquez une première bizarrerie : on place également `async` devant `with` ! C'est en fait pour indiquer que le contexte est asynchrone (on va lire dedans, en l'occurrence, mais on doit le fermer une fois qu'il est inutilisé, même si cette utilisation se fait en asynchrone).
- On lit le contenu de la réponse de façon asynchrone (avec `await`) et on l'écrit dans la variable `octets`. Réflexe à prendre : quand vous voyez `await`, souvenez-vous que l'instruction qui le suit (`réponse.read()` en l'occurrence) pourra prendre quelques secondes à s'exécuter et que Pyton met en pause la fonction pour l'heure. Il la reprendra quand le fichier sera téléchargé.
- Dans la fonction `enregistrer`, on appelle (en attendant son retour) la fonction `télécharger`. Si la fonction retourne bien quelque chose (en cas d'erreur, elle renvoie une chaîne vide), on crée un fichier sur le disque pour y enregistrer le retour de `télécharger`. On utilise `aiofiles.open` à la place de `open` ; là encore, l'opération sera asynchrone et pourra prendre un peu de temps pendant

lequel on laisse le programme travailler sans bloquer. Notez l'utilisation du `async with` de nouveau, ainsi que le `await` à l'intérieur.
— La fonction `tout_télécharger` est celle qui a le plus changé : au lieu d'appeler les coroutines séquentiellement (ce qui détruirait tout l'intérêt), on crée une liste dans laquelle on les enferme. Souvenez-vous qu'appeler `enregistrer(session, fichier)` n'exécute pas la fonction ici, mais retourne une coroutine de cette fonction. On conserve donc une liste des coroutines dans `tâches`. Et on transfère cette tâche à `asyncio.gather`, dans une liste de paramètres avec l'étoile devant le nom de la liste.
— Enfin, on appelle `tout_télécharger` comme coroutine, avec `asyncio.run`.

Cette version est nettement plus difficile à comprendre. Cela étant, remarquez que le code en lui-même n'a pas tellement changé. Il y a juste beaucoup de choses à comprendre, des choses qui semblent déroutantes. L'utilisation des mots-clés `async` et `await`, pour ne pas aller chercher plus loin, est une nouveauté de ce chapitre. L'utilisation des `async with` est une nouveauté de cet exemle. Il n'y a hélas pas le temps de rentrer dans le détail de toutes ces fonctionnalités mais cela vous donne un aperçu de la force de l'asynchrone en Python.

Côté positif donc : passé ces questions de syntaxe un peu délicates, on a un code assez facile à lire car il reste séquentiel (on peut facilement voir l'ordre, au moins au coeur d'une coroutine). Le gain en temps d'exécution est édifiant. Il est même sensiblement plus rapide que le même exemple écrit avec `threading` sans présenter autant de difficulté à la lecture.

Côté négatif : comme nous l'avons vu, l'asynchrone nécessite ses propres bibliothèques qui ne sont pas présentes dans la bibliothèque standard, pour la plupart. De nombreuses fonctionnalités que vous trouverez dans la bibliothèque standard ne sont pas faites pour l'asynchrone et ne seront sans doute jamais portées ; il s'agit d'une approche somme toute optionnelle. Une fois qu'on sait quoi utiliser, cela dit, cet obstacle est assez mineur. Plus important : il faut assimiler un peu de nouvelle syntaxe et surmonter des concepts qui semblent déroutants et contre-intuitifs, mais c'est un chemin qui en vaut la peine.

Quelques autres exemples

Le plus important dans ce chapitre, ce n'est pas le code, mais la théorie qui se cache derrière. Si vous comprenez, à peu près, l'intérêt d'`asyncio`, c'est le plus important. Le code pourra vous servir de modèle pour vos tests par la suite, si vous en avez l'utilité.

Nous avons vu un cas concret d'utilisation d'`asyncio`. Il y en a, bien entendu, de nombreux autres. En voici quelques-uns :

— Prenez votre jeu vidéo favori et examinez-le attentivement. Est-ce qu'il attend que vous appuyiez sur une touche de votre clavier ou cliquiez avec votre souris pour continuer ? Est-ce toujours le cas ? Les jeux en temps réel, notamment, ont généralement au moins deux tâches qui s'exécutent à peu près en même temps : le jeu s'exécute d'un côté (joue de la musique, des sons, fait avancer ou reculer des personnages dans l'univers) et surveille les entrées des utilisateurs (clavier, souris, manette de jeu et autres). On peut même décomposer en bien d'autres

tâches en fonction des besoins. Dans ce contexte, `asyncio` pourrait aider (avec d'autres approches possibles).
— Un client de messagerie. Est-ce que votre client attend que vous envoyiez un message pour recevoir ceux des autres ? Là encore, généralement, un client de messagerie exécute deux tâches parallèles : surveiller les messages reçus, surveiller les entrées de l'utilisateur. Sans cela, on ne pourrait rien recevoir à moins d'envoyer un message, ce qui serait assez embêtant pour la communication via ce type de client.
— Le navigateur Internet. Nous avons vu qu'`asyncio` permet de télécharger plusieurs ressources en même temps, depuis Internet. Cependant, vous pouvez également ouvrir plusieurs fenêtres et plusieurs onglets dans la plupart des navigateurs. Certains sites Internet ont du contenu dynamique. Ce contenu ne va pas bloquer tous les autres sites Internet chargés (même si le dit contenu peut fortement les ralentir parfois).
— Le réseau pour un serveur. Nous l'avons vu lors du chapitre sur `socket`, un serveur peut communiquer avec de nombreux clients. Au lieu d'utiliser `selector`, il est possible d'utiliser `asyncio` et les `streams` en appelant `await` pour lire et écrire dans ces connexions : `https://docs.python.org/fr/3/library/asyncio-stream.html` .
— ... et bien d'autres.

Si vous voyez pourquoi l'asynchrone pourrait être utile dans chacun de ces exemples, c'est tant mieux. Si vous trouvez d'autres exemples par vous-mêmes, c'est encore mieux ! Si l'intérêt vous semble nébuleux ou limité, pas de panique. `asyncio` est une façade de programmation parallèle, il en existe d'autres... et souvent, coder en parallèle n'est pas indispensable !

En résumé

— L'asynchrone est une des facettes de la bibliothèque standard permettant d'exécuter des tâches en parallèle.
— Une tâche, définie comme une fonction avec le mot-clé `async def` est appelée une coroutine en Python.
— L'agencement des tâches asynchrones permet de gagner du temps d'exécution tout en conservant une lisibilité assez élevée.
— Des bibliothèques tierces doivent souvent être utilisées pour tirer parti de l'asynchrone.

Chapitre 37

Des interfaces graphiques avec `Tkinter`

Difficulté :

Nous allons maintenant découvrir comment créer des interfaces graphiques à l'aide d'un module présent par défaut dans Python : `Tkinter`.

Il permet de créer des interfaces graphiques en offrant une passerelle entre Python et la bibliothèque Tk.

Vous allez apprendre dans ce chapitre à créer des fenêtres, ajouter des boutons et faire réagir vos objets graphiques à certains évènements.

Présentation de Tkinter

`Tkinter` *(*`Tk` *interface)* est un module intégré à la bibliothèque standard de Python, bien qu'il ne soit pas maintenu directement par les développeurs du langage. Il offre un moyen de créer des interfaces graphiques via Python.

`Tkinter` est disponible sur Windows et la plupart des systèmes Unix. Les interfaces que vous pourrez développer auront donc toutes les chances d'être portables d'un système à l'autre.

Notez qu'il existe d'autres bibliothèques pour créer des interfaces graphiques. `Tkinter` a l'avantage d'être disponible par défaut, sans nécessiter une installation supplémentaire.

Pour savoir si vous pouvez utiliser le module `Tkinter` via la version de Python installée sur votre système, tapez la commande suivante dans l'interpréteur :

```
from tkinter import *
```

Si une erreur se produit, renseignez-vous sur la page du Wiki Python consacrée à `Tkinter` : `https://wiki.python.org/moin/TkInter`

Votre première interface graphique

Commençons par voir le code minimal pour créer une fenêtre avec `Tkinter`. Ensuite, petit à petit, nous apprendrons à ajouter des éléments.

Étant en Python, ce code minimal est plutôt court :

```
"""Premier exemple avec Tkinter.

On crée une fenêtre simple qui souhaite la bienvenue à
 ↪  l'utilisateur.

"""

# On importe Tkinter
from tkinter import *

# On crée une fenêtre, racine de notre interface
fenêtre = Tk()

# On crée un label (ligne de texte) souhaitant la bienvenue
# Note : le premier paramètre passé au constructeur de Label
# est notre interface racine
champ_label = Label(fenêtre, text="Salut les Zér0s !")

# On affiche le label dans la fenêtre
champ_label.pack()

```

```
# On démarre la boucle Tkinter qui s'interrompt quand on ferme
↪  la fenêtre
fenêtre.mainloop()
```

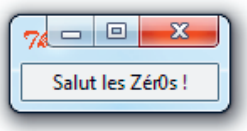


FIGURE 37.1 – Notre première fenêtre avec `Tkinter`

Vous pouvez recopier ce code dans un fichier `.py` (n'oubliez pas d'ajouter la ligne spécifiant l'encodage) et ensuite exécuter votre programme, ce qui affiche une fenêtre (simple, certes, mais une fenêtre tout de même).

Comme vous le constatez, la fenêtre est tout juste assez grande pour que le message s'affiche.

Regardons le code d'un peu plus près.

1. On commence par importer `Tkinter`, sans grande surprise.
2. On crée ensuite un objet de la classe `Tk`. La plupart du temps, cet objet sera la fenêtre principale de notre interface.
3. On crée un `Label`, c'est-à-dire un objet graphique affichant du texte.
4. On appelle la méthode `pack` de notre `Label`. Elle sert à positionner l'objet dans notre fenêtre (et, par conséquent, à l'afficher).
5. Enfin, on appelle la méthode `mainloop` de notre fenêtre racine. Elle ne retourne que lorsqu'on ferme la fenêtre.

Quelques petites précisions :

— Nos objets graphiques (boutons, champs de texte, cases à cocher, barres de progression…) sont appelés des *widgets*.
— On peut préciser plusieurs options lors de la construction de nos *widgets*. Ici, on définit l'option `text` de notre `Label` à `"Salut les Zér0s !"`.

Il existe plusieurs options communes à la plupart des *widgets* (leur couleur `fg`, celle du fond `bg`, etc.) et d'autres plus spécifiques à un certain type.

Les options sont modifiables lors de la création du *widget*, mais également après :

```
>>> champ_label["text"]
'Salut les Zér0s !'
>>> champ_label["text"] = "Maintenant, au revoir !"
>>> champ_label["text"]
'Maintenant, au revoir !'
>>>
```

Il suffit de passer le nom de l'option entre crochets (comme pour accéder à une valeur d'un dictionnaire). C'est le même principe pour accéder à la valeur actuelle de l'option ou pour la modifier.

De nombreux widgets

`Tkinter` définit un grand nombre de *widgets* utilisables dans notre fenêtre. Nous allons en présenter ici quelques-uns.

Les widgets les plus communs

Les labels

C'est le premier *widget* que nous ayons vu, hormis notre fenêtre principale qui en est un également. On s'en sert pour afficher du texte dans notre fenêtre, du texte qui ne sera pas modifié par l'utilisateur.

```
1 champ_label = Label(fenêtre, text="contenu de notre champ
  ↪  label")
2 champ_label.pack()
```

N'oubliez pas que, pour qu'un *widget* apparaisse, il faut :

— qu'il prenne la fenêtre principale en premier paramètre du constructeur ;
— qu'il fasse appel à la méthode `pack`.

La méthode `pack` définit la position d'un objet dans une fenêtre ou dans un cadre ; nous verrons plus loin quelques-uns de ses paramètres optionnels.

Les boutons

Les boutons sont des *widgets* sur lesquels on clique pour déclencher des actions ou **commandes** comme nous le verrons ultérieurement plus en détail.

```
1 bouton_quitter = Button(fenêtre, text="Quitter",
  ↪  command=fenêtre.quit)
2 bouton_quitter.pack()
```

J'imagine que vous vous posez des questions sur le dernier paramètre passé à notre constructeur de `Button`. Il s'agit de l'action liée à un clic sur le bouton. Ici, c'est la méthode `quit` de notre fenêtre racine qui est appelée.

Ainsi, quand vous cliquez sur le bouton `Quitter`, la fenêtre se ferme. Nous verrons plus tard comment créer nos propres commandes.

Si vous faites des tests depuis l'interpréteur Python en ligne de commande, la fenêtre Tk reste ouverte tant que la console l'est. Le bouton `Quitter` interrompra la boucle `mainloop` mais ne fermera pas l'interface.

Une ligne de saisie

Le *widget* que nous allons voir à présent est une zone de texte dans lequel l'utilisateur peut écrire. En fait de zone, il s'agit d'une ligne simple.

On préférera créer une variable `Tkinter` associée au champ de texte. Regardez le code qui suit :

```
1 var_texte = StringVar()
2 ligne_texte = Entry(fenêtre, textvariable=var_texte, width=30)
3 ligne_texte.pack()
```

À la ligne 1, nous créons une variable `Tkinter`. En résumé, c'est elle qui va ici contenir le texte de notre `Entry`. Il est possible de la lier à une méthode qui sera appelée appelée quand la variable sera modifiée (l'utilisateur écrit dans le champ `Entry`).

Pour en savoir plus, je vous renvoie à la méthode `trace` de la variable.

Comme vous l'avez peut-être remarqué, le *widget* `Entry` n'est qu'une zone de saisie. Pour que l'utilisateur sache ce qu'il doit y écrire, il est utile d'ajouter une indication auprès du champ. Le *widget* `Label` est le plus approprié dans ce cas.

Notez qu'il existe également le *widget* `Text`, qui représente un champ de texte à plusieurs lignes.

Les cases à cocher

Les cases à cocher sont définies dans la classe `Checkbutton`. Là encore, on utilise une variable pour surveiller la sélection de la case.

Pour surveiller l'état d'une case à cocher (qui peut être soit active soit inactive), on préférera créer une variable de type `IntVar` plutôt que `StringVar`, bien que ce ne soit pas une obligation.

```
1 var_case = IntVar()
2 case_à_cocher = Checkbutton(fenêtre, text="Ne plus poser cette
  ↪  question", variable=var_case)
3 case_à_cocher.pack()
```

Vous pouvez ensuite contrôler l'état de la case à cocher en interrogeant la variable :

```
1 var_case.get()
```

Si la case est cochée, la valeur renvoyée par la variable sera `1`. Si elle n'est pas cochée, ce sera `0`.

Notez qu'à l'instar d'un bouton, vous pouvez lier la case à cocher à une commande qui sera appelée quand son état change.

Les boutons radio

Les boutons radio (*radio buttons* en anglais) sont des boutons généralement présentés en groupes. C'est, à proprement parler, un ensemble de cases à cocher mutuellement

exclusives : quand vous cliquez sur l'un des boutons, celui-ci se sélectionne et tous les autres boutons du même groupe se désélectionnent.

Ce type de *widget* est donc surtout utile dans le cadre d'un regroupement.

Pour créer un groupe de boutons, il faut simplement qu'ils soient tous associés à la même variable `Tkinter`, du type que vous voulez.

Quand l'utilisateur change le bouton sélectionné, la valeur de la variable change également en fonction de l'option `value` associée au bouton. Voyons un exemple :

```
var_choix = StringVar()

choix_rouge = Radiobutton(fenêtre, text="Rouge",
 ↪  variable=var_choix, value="rouge")
choix_vert = Radiobutton(fenêtre, text="Vert",
 ↪  variable=var_choix, value="vert")
choix_bleu = Radiobutton(fenêtre, text="Bleu",
 ↪  variable=var_choix, value="bleu")

choix_rouge.pack()
choix_vert.pack()
choix_bleu.pack()
```

Pour récupérer la valeur associée au bouton actuellement sélectionné, « interrogez » la variable :

```
var_choix.get()
```

La liste déroulante

Ce *widget* permet de construire une liste dans laquelle on sélectionne un ou plusieurs élément(s). Le fonctionnement n'est pas tout à fait identique à celui des boutons radio. Ici, la liste comprend plusieurs lignes et non un groupe de boutons.

Créer une liste se fait assez simplement :

```
liste = Listbox(fenêtre)
liste.pack()
```

On insère ensuite des éléments. La méthode `insert` prend deux paramètres :

1. la position à laquelle insérer l'élément ;
2. l'élément même, sous la forme d'une chaîne de caractères.

Si vous voulez insérer des éléments à la fin de la liste, utilisez la constante `END` définie par `Tkinter` :

```
liste.insert(END, "Pierre")
liste.insert(END, "Feuille")
liste.insert(END, "Ciseau")
```

Pour accéder à la sélection, utilisez la méthode `curselection` de la liste. Elle renvoie un *tuple* de chaînes de caractères, chacune étant la position de l'élément sélectionné.

Par exemple, si `liste.curselection()` renvoie `('2',)`, c'est le troisième élément de la liste qui est sélectionné (`Ciseau` en l'occurrence).

Organiser ses widgets dans la fenêtre

Il existe plusieurs *widgets* qui servent à en contenir d'autres. L'un d'entre eux se nomme `Frame`. C'est un cadre rectangulaire dans lequel vous pouvez placer vos *widgets*... ainsi que d'autres objets `Frame` si besoin est.

Si vous voulez qu'un *widget* apparaisse dans un cadre, spécifiez le `Frame` comme son parent lors de sa création :

```
cadre = Frame(fenêtre, width=768, height=576, borderwidth=1)
cadre.pack(fill=BOTH)

message = Label(cadre, text="Notre fenêtre")
message.pack(side="top", fill=X)
```

Comme vous le voyez, nous avons passé plusieurs arguments nommés à notre méthode `pack`.

En précisant `side="top"`, on demande à ce que l'élément soit placé en haut de son parent (ici, notre cadre).

Il existe aussi l'argument nommé `fill` qui permet au *widget* de remplir son parent, soit en largeur si la valeur est `X`, soit en hauteur si la valeur est `Y`, soit en largeur et hauteur si la valeur est `BOTH`.

D'autres arguments nommés existent, bien entendu. Si vous voulez une liste exhaustive, rendez-vous sur le chapitre consacré à `Tkinter` dans la documentation officielle de Python : `https://docs.python.org/fr/3/library/tk.html`

Une partie est consacrée au *packer* et à sa méthode `pack`.

Notez qu'il existe aussi le *widget* `Labelframe`, un cadre avec un titre, ce qui nous évite d'avoir à placer un `label` en haut du cadre. Il se construit comme un `Frame` mais prend en argument, à la construction, le texte représentant le titre : `cadre = Labelframe(..., text="Titre du cadre")`.

Bien d'autres widgets

Vous devez vous en douter, ceci n'est qu'une approche très sommaire de quelques éléments de `Tkinter`. Il en existe de nombreux autres et ceux que nous avons présentés ont bien d'autres options.

Il est notamment possible de créer une barre de menus avec ses menus imbriqués, d'afficher des images, des `canvas` dans lesquels vous pouvez dessiner pour personnaliser votre fenêtre... bref, il vous reste bien des choses à apprendre.

Détaillons les commandes que nous avons effleurées jusqu'ici sans trop nous pencher dessus.

Les commandes

Nous avons vu très brièvement comment faire en sorte qu'un bouton ferme une fenêtre quand on clique dessus :

```
bouton_quitter = Button(fenêtre, text="Quitter",
    command=fenêtre.quit)
```

C'est le dernier argument qui est important ici. Il a pour nom `command` et a pour valeur la méthode `quit` de notre fenêtre.

Sur ce modèle, nous pouvons créer assez simplement des commandes personnalisées, en écrivant des méthodes.

Cependant, il y a ici une petite subtilité : la méthode que nous devons créer ne prend aucun paramètre. Si nous voulons qu'un clic sur le bouton modifie le bouton lui-même ou un autre objet, nous devons placer nos *widgets* dans un corps de classe.

D'ailleurs, à partir du moment où on sort du cadre d'un test, il est préférable d'écrire le code dans une classe.

On peut la faire hériter de `Frame`, ce qui signifie que notre classe sera un *widget* elle aussi. Voyons un code complet :

```
from tkinter import *

class Interface(Frame):

    """Notre fenêtre principale.

    Tous les widgets sont stockés comme attributs
    de cette fenêtre.

    """

    def __init__(self, fenêtre, **kwargs):
        Frame.__init__(self, fenêtre, width=768, height=576,
            **kwargs)
        self.pack(fill=BOTH)
        self.nb_clic = 0

        # Création de nos widgets
        self.message = Label(self, text="Vous n'avez pas
            cliqué sur le bouton.")
        self.message.pack()

        self.bouton_quitter = Button(self, text="Quitter",
            command=self.quit)
        self.bouton_quitter.pack(side="left")

```

```
24          self.bouton_cliquer = Button(self, text="Cliquez ici",
             ↪  fg="red",
25                  command=self.cliquer)
26          self.bouton_cliquer.pack(side="right")
27
28      def cliquer(self):
29          """Il y a eu un clic sur le bouton.
30
31          On change la valeur du label message.
32
33          """
34          self.nb_clic += 1
35          self.message["text"] = f"Vous avez cliqué
             ↪  {self.nb_clic} fois."
```

Et créons notre interface :

```
1 fenêtre = Tk()
2 interface = Interface(fenêtre)
3
4 interface.mainloop()
5 interface.destroy()
```

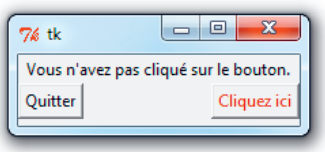


FIGURE 37.2 – La fenêtre est créée avec ses deux boutons

Dans l'ordre :

1. On crée une classe qui contiendra toute la fenêtre. Elle hérite de `Frame`, c'est-à-dire d'un cadre `Tkinter`.
2. Dans le constructeur de la fenêtre, on appelle celui du cadre et on positionne et affiche (`pack`) le cadre.
3. Toujours dans le constructeur, on crée les différents *widgets* de la fenêtre. On les positionne et les affiche également.
4. On crée une méthode `cliquer`, qui est appelée quand on appuie sur `bouton_cliquer`. Elle ne prend aucun paramètre. Elle va mettre à jour le texte contenu dans le `label self.message` pour afficher le nombre de clics enregistrés sur le bouton.
5. On crée la fenêtre `Tk` qui est l'objet parent de l'interface que l'on instancie ensuite.
6. On rentre dans la boucle `mainloop`. Elle s'interrompra quand on fermera la fenêtre.
7. Ensuite, on détruit la fenêtre grâce à la méthode `destroy`.

Pour conclure

Ceci n'est qu'un survol, j'insiste sur ce point. `Tkinter` est une bibliothèque trop riche pour être présentée en un chapitre. Vous trouverez de nombreux exemples d'interfaces de par le Web, si vous cherchez quelque chose de plus précis.

En résumé

- `Tkinter` est un module intégré à la bibliothèque standard et qui sert à créer des interfaces graphiques.
- Les objets graphiques (boutons, zones de texte, cases à cocher…) sont appelés des *widgets*.
- Dans `Tkinter`, les *widgets* prennent, lors de leur construction, leur objet parent en premier paramètre.
- Chaque *widget* possède des options à peut préciser comme arguments nommés lors de sa construction.
- On peut également accéder aux options d'un *widget* comme suit : `widget["nom_option"]`.

Cinquième partie

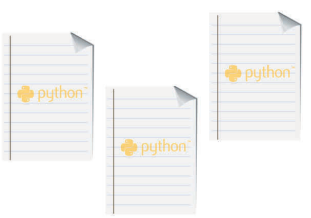
Annexes

Annexe 38

Écrire nos programmes Python dans des fichiers

Difficulté :

Ce petit chapitre vous explique comment garder votre code Python dans un fichier pour l'exécuter. Vous pouvez le lire très rapidement ; tant que vous savez à quoi sert la fonction `print`, c'est tout ce dont vous avez besoin.



Garder le code dans un fichier

Pour placer du code dans un fichier que nous pourrons ensuite exécuter, la démarche est très simple :

1. Ouvrez un éditeur standard sans mise en forme (Notepad++, VIM ou Emacs...). Dans l'absolu, le Bloc-notes Windows est aussi candidat, mais il reste moins agréable pour programmer (pas de coloration syntaxique du code, notamment). Et il va poser problème avec vos accents (voir la section suivante).
2. Dans ce fichier, recopiez simplement `print("Bonjour le monde !")`, comme à la figure 38.1.
3. Enregistrez ce code dans un fichier à l'extension `.py`, comme à la figure 38.2. Cela est surtout utile sur Windows.

Je recommande fortement Notepad++ aux utilisateurs de Windows : il est léger, gratuit et très complet : `https://notepad-plus-plus.org/fr/`

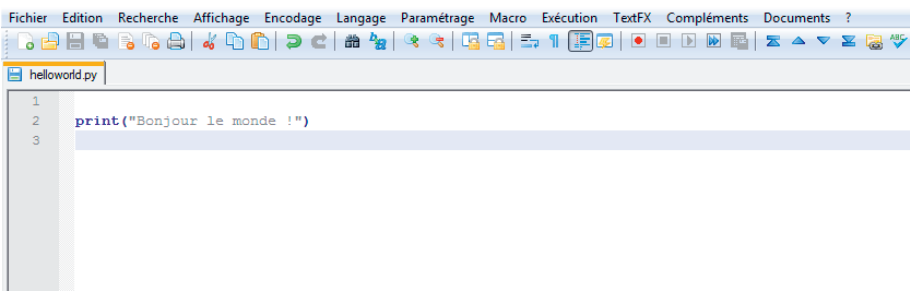


FIGURE 38.1 – Une ligne de code dans Notepad++

Exécuter notre code sur Windows

Dans l'absolu, vous pouvez double-cliquer sur le fichier à l'extension `.py` dans l'explorateur de fichiers. Cependant, la fenêtre s'ouvre et se referme très rapidement. Pour éviter cela, vous avez trois possibilités :

— mettre le programme en pause (voir la dernière section de ce chapitre) ;
— lancer le programme depuis la console Windows (je ne m'attarderai pas ici sur cette solution) ;
— exécuter le programme avec `IDLE`.

C'est cette dernière opération que je vais détailler brièvement. Cliquez-droit sur le fichier `.py`. Dans le menu contextuel, vous devriez voir apparaître un intitulé du type `edit with IDLE`. Cliquez dessus.

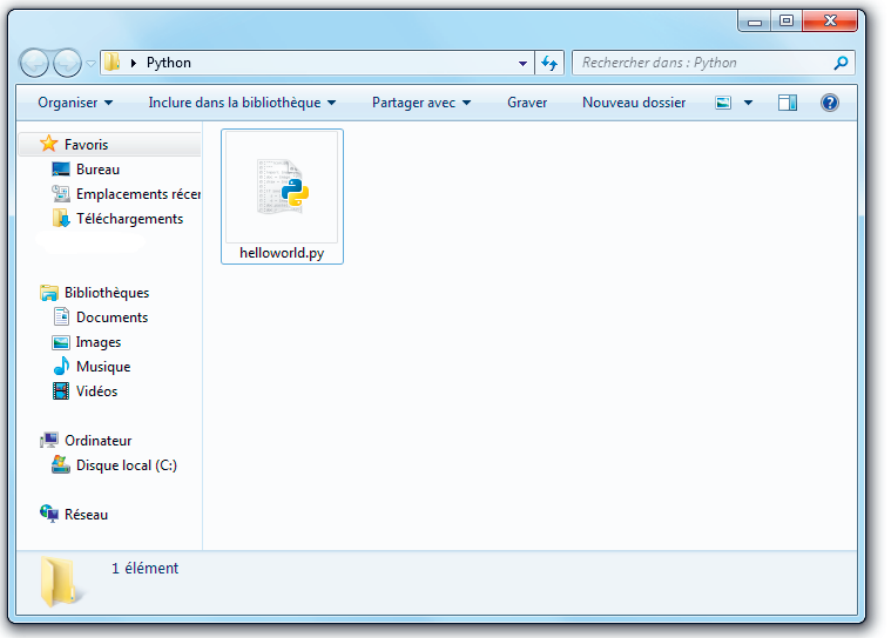


FIGURE 38.2 – Un fichier `.py` sous Windows

La fenêtre d'IDLE s'affiche. Vous voyez votre code, ainsi que plusieurs boutons. Cliquez sur `run` puis sur `run module` (ou appuyez sur F5 directement).

Le code du programme se lance. Cette fois, la fenêtre de console reste ouverte pour que vous puissiez voir le résultat ou les erreurs éventuelles.

Sur les systèmes Unix

Il est nécessaire d'ajouter, tout en haut de votre programme, une ligne qui indique le chemin menant vers l'interpréteur. Elle se présente sous la forme `#!/chemin`.

Les habitués du Bash devraient reconnaître cette ligne assez rapidement. Pour les autres, sachez qu'il suffit de mettre à la place de « `chemin` » le chemin absolu de l'interpréteur (celui qui, en partant de la racine du système, mène à l'interpréteur Python). Voici un exemple :

```
1 #!/usr/local/bin/python3.10
```

En changeant les droits d'accès en exécution sur le fichier, vous devriez pouvoir le lancer directement.

Préciser l'encodage de travail

Si vous êtes sous Windows (en particulier si vous utilisez le Bloc-notes pour coder) et si vous utilisez les accents dans votre programme, vous devez préciser l'encodage choisi pour l'écrire.

Très brièvement, l'encodage est une table contenant une série de codes symbolisant différents accents. Deux encodages sont très utilisés en France : **Latin-1** sur Windows et **UTF-8** que l'on retrouve surtout sur les machines Unix.

Si vous ne précisez pas l'encodage, Python part du principe que vous utilisez l'UTF-8. C'est un choix standard et assez logique... sauf pour les utilisateurs de Windows (en particulier avec le Bloc-notes). Une ligne de commentaire doit être ajoutée tout en haut de votre code (si vous êtes sur un système Unix, sous la ligne qui fournit le chemin menant vers l'interpréteur) :

```
1 # -*-coding:encodage -*
```

Remplacez `encodage` par celui que vous utilisez en fonction de votre système.

Sur Windows, on trouvera donc vraisemblablement `# -*-coding:Latin-1 -*`

Sur Linux ou Mac, ce sera plus problablement `# -*-coding:Utf-8 -*`

Si votre programme est en UTF-8, ce que je vous conseille, que ce soit sous Windows ou ailleurs, vous n'avez pas besoin d'inclure cette ligne déclarative car c'est le choix par défaut.

Notez que, sous Windows, si vous utilisez Notepad++ ou un autre éditeur du même type, vous pouvez généralement le configurer pour utiliser l'UTF-8, ce qui vous dispense d'avoir à ajouter cette ligne systématiquement.

Mettre notre programme en pause

Sur Windows se pose un problème : si vous lancez votre programme en double-cliquant directement dessus dans l'explorateur, il aura tendance à s'ouvrir et se fermer très rapidement. Python exécute bel et bien le code et affiche le résultat, mais tout cela très rapidement. Et une fois que la dernière ligne de code a été exécutée, Windows ferme la console.

Pour pallier ce problème, on peut demander à Python de se mettre en pause à la fin de l'exécution du code.

Il va falloir ajouter deux lignes, l'une au début de notre programme et l'autre tout à la fin. La première importe le module `os` et la seconde utilise une fonction de ce dernier pour mettre le programme en pause.

```
1 # -*-coding:Latin-1 -*
2 # (La ligne précédente est utile si vous êtes sous Windows
  ↪  avec le Bloc-notes)
```

```
3
4 import os # On importe le module os
5 print("Bonjour le monde !")
6 os.system("pause")
```

Ce sont les lignes 4 et 6 que vous devez retenir. Quand vous exécutez ce code, vous obtenez :

```
1 Bonjour le monde !
2 Appuyez sur une touche pour continuer...
```

Vous pouvez maintenant lancer ce programme en double-cliquant directement dessus dans l'explorateur de fichiers.

Notez bien que ce code ne fonctionne que sur Windows !

Si vous voulez mettre votre programme en pause sur Linux ou Mac, vous devrez utiliser un autre moyen. Même si la fonction `input` n'est pas faite pour, il est possible de vous en servir et de conclure votre programme par la ligne suivante :

```
1 input("Appuyez sur ENTREE pour fermer ce programme...")
```

En résumé

- Pour créer un programme Python, il suffit d'écrire du code dans un fichier (de préférence avec l'extension `.py`).
- Si on utilise des accents dans le code, sauf si nos fichiers utilisent l'UTF-8, il est nécessaire de préciser, en tête de fichier, l'encodage utilisé.
- Sur Windows, pour lancer notre programme Python depuis l'explorateur, il faut le mettre en pause grâce au module `os` et à sa fonction `system`.

Annexe 39

Distribuer facilement nos programmes Python avec PyInstaller

Difficulté :

Comme nous l'avons vu, Python nous permet de générer des exécutables d'une façon assez simple. Cependant, si vous voulez distribuer votre programme, vous risquez de vous heurter au problème suivant : pour lancer votre code, votre destinataire doit installer Python, qui plus est dans la bonne version. De plus, si vous faites appel à des bibliothèques tierces, il doit aussi les installer !

Heureusement, il existe plusieurs moyens pour produire des fichiers exécutables distribuables et qui incluent tout le nécessaire. Sur Windows, il faut enfermer vos fichiers à l'extension `.py` dans un `.exe` accompagné de fichiers `.dll`. PyInstaller est un des outils pour atteindre cet objectif.

En théorie

L'objectif de ce chapitre est de vous montrer comment produire des programmes dits *standalone*[1]. Comme vous le savez, pour que vos fichiers `.py` s'exécutent, il faut que Python soit installé sur votre machine. Toutefois, vous pourriez vouloir transmettre votre programme sans obliger vos utilisateurs à installer le langage sur leur ordinateur.

Une version *standalone* de votre programme contient, en plus de votre code, l'exécutable Python et les dépendances dont il a besoin.

Sur Windows, vous vous retrouverez avec un fichier `.exe` et plusieurs fichiers compagnons, bien plus faciles à distribuer et, pour vos utilisateurs, à exécuter.

Le programme résultant ne sera pas sensiblement plus rapide ou plus lent. Il ne s'agit pas de compilation, Python reste un langage interprété et l'interpréteur sera appelé pour lire votre code, même si celui-ci se trouvera dans une forme un peu plus compressée.

Avantages de PyInstaller

— Portabilité : PyInstaller est fait pour fonctionner aussi bien sur Windows que sur Linux ou macOS.
— Simplicité : créer son programme *standalone* avec PyInstaller est incroyablement simple.
— Souplesse : il est aisé de personnaliser votre programme *standalone* avant de le construire.

Il existe d'autres outils similaires, dont les plus célèbres sont sans doute `cx_Freeze` et `py2exe`. `cx_Freeze` est robuste et assez semblable à PyInstaller (l'un ou l'autre outil est généralement conseillé et vous trouverez de fervents partisans de chaque). `py2exe` a l'inconvénient de ne produire des programmes exécutables que sous Windows.

Nous allons voir à présent comment installer PyInstaller et comment construire nos programmes *standalone*.

En pratique

Il existe plusieurs façons d'utiliser PyInstaller. Il nous faut dans tous les cas commencer par l'installer.

Installation

Le plus simple est d'utiliser l'utilitaire `pip` en ligne de commande. Je vous renvoie à l'annexe en page 473 pour plus d'informations sur l'installation de bibliothèques tierces.

Une fois la console ouverte, entrez la commande suivante :

```
1 python -m pip install pyinstaller
```

1. Que l'on peut traduire très littéralement par « qui se tient seul ».

Elle va installer la dernière version de PyInstaller compatible avec votre version de Python. Je vous recommande d'installer PyInstaller dans un environnement indépendant (voir page 477).

Utiliser le script `pyinstaller`

Après l'installation, vous créez facilement un fichier exécutable grâce à la commande `pyinstaller`. Commencez par créer un fichier pour l'exemple (disons `bonjour.py`). Placez-y du code simple :

```
1 import os # Sous Windows en particulier
2 print("Bonjour tout le monde !")
3 os.system("pause") # Sous Windows uniquement
```

Enregistrez ce fichier. Dans le même dossier, ouvrez une ligne de commande et entrez l'instruction suivante :

```
1 pyinstaller bonjour.py
```

La commande devrait prendre un moment. Après une ou plusieurs minute(s), l'utilitaire devrait vous indiquer que tout s'est bien passé :

```
1 10124 INFO: Building COLLECT COLLECT-00.toc completed
  ↪  successfully.
```

Le message est assez surprenant et je n'expliquerai pas ce qu'il entend par `COLLECT`. L'important, c'est que, dans le dossier où vous avez lancé la commande, vous avez à présent deux sous-dossiers : `build` et `dist`. Ne nous occupons pas de `build` pour l'heure. Ouvrez le dossier `dist`. Dedans devrait se trouver un sous-dossier du nom de votre script (`bonjour`). Ouvrez-le et dedans...

... il y a beaucoup de choses ! Le plus important, c'est le fichier `bonjour.exe` (sous Windows) ou `bonjour` (sous d'autres systèmes). Double-cliquez dessus et votre script se lance.

Cependant, contrairement à votre fichier `bonjour.py`, si vous donnez le fichier `bonjour` ou `bonjour.exe` à quelqu'un qui n'a pas installé Python (sans oublier les autres fichiers présents dans le dossier), il devrait pouvoir l'exécuter. Ce sera bien plus facile à distribuer.

Et PyInstaller nous propose encore plus simple. Retournez dans le dossier où se trouve votre script Python (`bonjour.py`). Supprimez les dossiers `build` et `dist`. Retournez dans votre console pour une nouvelle commande (la même, mais avec une option en plus) :

```
1 pyinstaller bonjour.py --onefile
```

Retournez dans votre dossier `dist` une fois que PyInstaller a fini son travail. Cette fois, vous ne devriez y voir qu'un seul fichier : notre exécutable (`bonjour.exe` sous Windows, `bonjour` ailleurs). Si vous cliquez dessus, votre script se lancera.

La différence ici, c'est que tout est regroupé dans un seul fichier, bien plus facile à partager avec vos amis ou utilisateurs. Cet unique fichier exécutable contient votre script, l'exécutable Python et les dépendances nécessaires pour le lancer. Et il ne pèse pas bien lourd (moins de 5 Mo pour moi). Il est un petit peu plus long à se lancer, mais il tourne aussi rapidement que la version créée à l'étape précédente.

Enfin, une dernière note sur PyInstaller. Si vous examinez votre dossier d'origine, vous constatez qu'il vous a créé un fichier, `bonjour.spec`. Il contient du code Python ; vous pouvez l'ouvrir comme d'habitude et le modifier. Il liste les spécifications de construction de votre exécutable et permet de retenir la stratégie à utiliser pour créer le fichier à partager. Sachez que `pyinstaller` (le script) possède un grand nombre d'options. La personnalisation au travers du `fichier.spec` est également très élaborée. Et PyInstaller propose encore plus poussé si vous en avez besoin. C'est un outil puissant et simple à utiliser.

Malgré tout, il arrive, quand on travaille sur des scripts avec des dépendances un peu inhabituelles, que l'outil ait du mal à créer un exécutable fonctionnel. Vous pouvez l'aider à comprendre votre code et trouver les dépendances. Vous pouvez aussi changer d'outil si vous trouvez que PyInstaller n'est pas le meilleur choix. Privilégiez ce qui fonctionne pour vous et n'hésitez pas à explorer ; tout est là.

Annexe 40

De bonnes pratiques

Difficulté :

Nous allons à présent nous intéresser à quelques **bonnes pratiques** de codage en Python.

Les conventions dont nous allons discuter sont, naturellement, des propositions. Vous pouvez coder en Python sans les suivre.

Toutefois, prenez le temps de considérer les quelques affirmations ci-après. Si vous vous sentez concerné, ne serait-ce que par l'une d'entre elles, je vous invite à lire ce chapitre :

- Un code dont on est l'auteur peut être difficile à relire si on l'abandonne quelque temps.
- Lire le code d'un autre développeur est toujours plus délicat.
- Si votre code doit être utilisé par d'autres, il doit être facile à reprendre (à lire et à comprendre).

Pourquoi suivre les conventions des PEP ?

Vous avez absolument le droit de répondre en disant que personne ne lira votre code de toute façon et que vous-même n'aurez aucun mal à le comprendre. Seulement, si votre code prend des proportions importantes ou si l'application que vous développez est de plus en plus utilisée, il est préférable pour vous d'adopter quelques conventions clairement définies dès le début. Et, étant donné qu'il n'est jamais certain qu'un projet, même démarré comme un amusement passager, ne devienne pas un jour énorme, ayez les bons réflexes dès le début !

En outre, vous ne pouvez jamais être sûrs à cent pour cent qu'aucun développeur ne vous rejoindra, à terme, sur le projet. Si votre application est utilisée par d'autres, là encore, ce jour arrivera peut-être lorsque vous n'aurez pas assez de temps pour poursuivre seul son développement.

Quoi qu'il en soit, je vais vous présenter plusieurs conventions qui nous sont proposées au travers des PEP [1]. Encore une fois, il s'agit de propositions et vous êtes libres d'en choisir d'autres si celles-ci ne vous plaisent pas.

La PEP 20 : toute une philosophie

La PEP 20, intitulée *The zen of Python*, nous donne des conseils très généraux sur le développement. Vous la trouverez à l'adresse suivante : `https://www.python.org/dev/peps/pep-0020/`

Bien entendu, ce sont davantage des conseils axés sur « comment programmer en Python », mais ils sont pour la plupart applicables à la programmation en général.

Je vous propose une traduction de cette PEP :

- *Beautiful is better than ugly* : le beau est préférable au laid.
- *Explicit is better than implicit* : l'explicite est préférable à l'implicite.
- *Simple is better than complex* : le simple est préférable au complexe.
- *Complex is better than complicated* : le complexe est préférable au compliqué.
- *Flat is better than nested* : le plat est préférable à l'imbriqué [2].
- *Sparse is better than dense* : l'aéré est préférable au compact.
- *Readability counts* : la lisibilité compte.
- *Special cases aren't special enough to break the rules* : les cas particuliers ne sont pas suffisamment particuliers pour casser la règle.
- *Although practicality beats purity* : même si l'aspect pratique doit prendre le pas sur la pureté [3].
- *Errors should never pass silently* : les erreurs ne devraient jamais passer silencieusement.

1. *Python Enhancement Proposal* : propositions d'amélioration de Python.
2. Moins littéralement, du code trop imbriqué (par exemple une boucle imbriquée dans une boucle imbriquée dans une boucle...) est plus difficile à lire.
3. Moins littéralement, il est difficile de faire un code à la fois fonctionnel et « pur ».

— *Unless explicitly silenced* : à moins qu'elles n'aient été explicitement réduites au silence.
— *In the face of ambiguity, refuse the temptation to guess* : en cas d'ambiguïté, résistez à la tentation de deviner.
— *There should be one – and preferably only one – obvious way to do it* : il devrait exister une (et de préférence une seule) manière évidente de procéder.
— *Although that way may not be obvious at first unless you're Dutch* : même si cette manière n'est pas forcément évidente au premier abord, à moins que vous ne soyez Néerlandais [4].
— *Now is better than never* : maintenant est préférable à jamais.
— *Although never is often better than *right* now* : même si jamais est parfois préférable à *immédiatement*.
— *If the implementation is hard to explain, it's a bad idea* : si la mise en œuvre est difficile à expliquer, c'est une mauvaise idée.
— *If the implementation is easy to explain, it may be a good idea* : si la mise en œuvre est facile à expliquer, ce peut être une bonne idée.
— *Namespaces are one honking great idea – let's do more of those* : les espaces de noms sont une très bonne idée (faisons-en plus !).

Comme vous le voyez, c'est une liste d'aphorismes très simples. Ils donnent des idées sur le développement Python mais, en les lisant pour la première fois, vous n'y voyez sans doute que peu de conseils pratiques.

Cependant, cette liste est vraiment importante et peut se révéler très utile. Certaines des idées qui s'y trouvent couvrent des pans entiers de la philosophie de Python.

Si vous travaillez sur un projet en équipe, un autre développeur pourra contester la mise en œuvre d'un extrait de code quelconque en se basant sur l'un de ces aphorismes.

Quand bien même vous travailleriez seul, il est toujours préférable de comprendre et d'appliquer la philosophie d'un langage quand on l'utilise pour du développement.

Je vous conseille donc de garder sous les yeux, autant que possible, cette synthèse de la philosophie de Python et de vous y référer à la moindre occasion. Commencez par lire chaque proposition. Les lignes sont courtes, prenez le temps de bien comprendre ce qu'elles signifient.

Sans trop détailler, je signale à votre attention que plusieurs de ces aphorismes parlent surtout de l'allure du code. L'idée qui semble se dissimuler derrière, c'est qu'un code fonctionnel n'est pas suffisant : il faut, autant que possible, faire du « beau code ».

Maintenant, nous allons nous intéresser à deux autres PEP qui vous donnent des conseils très pratiques sur le développement :

— La première nous donne des conseils très précis sur la présentation du code.
— La seconde concerne la documentation au cœur de notre code.

4. Petit clin d'œil au créateur du langage, Guido van Rossum, qui est... Néerlandais !

La PEP 8 : des conventions précises

Maintenant que nous avons vu des directives très générales, nous allons nous intéresser à une autre proposition d'amélioration, la PEP 8. Elle nous donne des conseils très précis sur la forme du code. Là encore, c'est à vous de voir : vous pouvez appliquer la totalité des conseils donnés ici ou une partie seulement. Voici l'adresse web vous permettant d'accéder à cette PEP : `https://www.python.org/dev/peps/pep-0008/`

Je ne vais pas reprendre tout ce qui figure dans cette PEP, mais expliquer la plupart des conseils en les simplifiant. Par conséquent, si l'une des propositions présentées dans cette section n'est pas claire à vos yeux, je vous conseille de consulter la PEP originale. Ce qui suit n'est pas une traduction complète, j'insiste sur ce point.

Introduction

L'une des convictions de Guido van Rossum [5] est que le code est lu beaucoup plus souvent qu'il n'est écrit. Les conseils donnés ici sont censés améliorer la lisibilité du code. Comme le dit la PEP 20, *la lisibilité compte* !

Un guide comme celui-ci parle de cohérence. La cohérence au cœur d'un projet est importante. La cohérence au sein d'une fonction ou d'un module est encore plus importante.

Cependant, il est encore plus essentiel de savoir « quand » être incohérent (parfois, les conseils de style donnés ici ne s'appliquent pas). En cas de doute, remettez-vous en à votre bon sens. Regardez plusieurs exemples et choisissez celui qui semble le meilleur.

Il y a deux bonnes raisons de ne pas respecter une règle donnée :

1. quand appliquer la règle rend le code moins lisible ;
2. dans un souci de cohérence avec du code existant qui ne respecte pas cette règle non plus. Ce cas peut se produire si vous utilisez un module ou une bibliothèque qui ne respecte pas les mêmes conventions que celles définies ici.

Forme du code

— Indentation : utilisez quatre espaces par niveau d'indentation.
— Tabulations ou espaces : ne mélangez *jamais*, dans le même projet, des indentations à base d'espaces et d'autres à base de tabulations. À choisir, on préfère généralement les espaces.
— Longueur maximum d'une ligne : limitez vos lignes à un maximum de 79 caractères, longueur plébiscitée par de nombreux éditeurs. Pour les blocs de texte relativement longs (*docstrings*, par exemple), limitez-vous de préférence à 72 caractères par ligne.
Quand cela est possible, découpez vos lignes en utilisant des parenthèses, crochets ou accolades plutôt que la barre oblique inverse.

5. Guido Van Rossum, créateur et ex-BDFL (*Benevolent Dictator For Life* : « dictateur bienveillant à vie ») de Python.

Exemple :

```
appel_d_une_fonction(paramètre_1, paramètre_2,
        paramètre_3, paramètre_4)
```

Si vous devez découper une ligne trop longue, faites la césure *après* l'opérateur, pas avant.

```
# Oui
    un_long_calcul = variable + \
            taux * 100

# Non
    un_long_calcul = variable \
            + taux * 100

# Encore mieux (on utilise des parenthèses pour éviter la
 ↪  barre oblique inverse)
    un_long_calcul = (variable +
            taux * 100)
```

— Sauts de ligne : séparez par deux sauts de ligne la définition d'une fonction et celle d'une classe.
Les définitions de méthodes au cœur d'une classe sont séparées par une ligne vide.
Des sauts de ligne peuvent également être utilisés, parcimonieusement, pour délimiter des portions de code
— Encodage : à partir de Python 3.0, il est conseillé d'utiliser, dans du code comportant des accents, l'encodage UTF-8.

Directives d'importation

— Les directives d'importation doivent préférentiellement se trouver sur plusieurs lignes. Par exemple, on écrira :

```
import os
import sys
```

plutôt que :

```
import os, sys
```

Cette syntaxe est cependant acceptée quand on importe certaines données d'un module :

```
from subprocess import Popen, PIPE
```

— Les directives d'importation doivent toujours se trouver en tête du fichier, sous la documentation éventuelle du module mais avant la définition de variables globales ou de constantes du module.
— Les directives d'importation doivent être divisées en trois groupes, dans l'ordre :

1. celles faisant référence à la bibliothèque standard ;

2. celles faisant référence à des bibliothèques tierces ;
3. celles faisant référence à des modules de votre projet.

Il devrait y avoir un saut de ligne entre chaque groupe.

— Dans vos directives d'importation, utilisez des chemins absolus plutôt que relatifs :

```
from paquet.souspaquet import module

# Est préférable à
from . import module
```

Le signe espace dans les expressions et instructions

— Évitez le signe espace dans les situations suivantes :

— Au cœur des parenthèses, crochets et accolades :

```
# Oui
    trucs(chose[1], {bidule: 2})

# Non
    trucs( chose[ 1 ], { bidule: 2 } )
```

— Juste avant une virgule, un point-virgule ou un signe deux-points :

```
# Oui
    if x == 4: print x, y; x, y = y, x

# Non
    if x == 4 : print x , y ; x , y = y , x
```

— Juste avant la parenthèse ouvrante qui introduit la liste des paramètres d'une fonction :

```
# Oui
    trucs(1)

# Non
    trucs (1)
```

— Juste avant le crochet ouvrant indiquant une indexation ou sélection :

```
# Oui
    dict['clé'] = liste[index]

# Non
    dict ['clé'] = liste [index]
```

— Plus d'un espace autour de l'opérateur d'affectation = (ou autre) pour l'aligner avec une autre instruction :

```
# Oui
    x = 1
    y = 2
    variable_longue = 3

# Non
    x                = 1
    y                = 2
    variable_longue = 3
```

— Toujours entourer les opérateurs suivants d'un espace (un avant le symbole, un après) :
  — affectation : `=`, `+=`, `-=`, etc. ;
  — comparaison : `<`, `>`, `<=`, ..., `in`, `not in`, `is`, `is not` ;
  — booléens : `and`, `or`, `not` ;
  — arithmétiques : `+`, `-`, `*`, etc.

```
# Oui
    i = i + 1
    submitted += 1
    x = x * 2 - 1
    hypot2 = x * x + y * y
    c = (a + b) * (a - b)

# Non
    i=i+1
    submitted +=1
    x = x*2 - 1
    hypot2 = x*x + y*y
    c = (a+b) * (a-b)
```

Attention : n'utilisez pas d'espaces autour du signe = si c'est dans le contexte d'un paramètre ayant une valeur par défaut (définition d'une fonction) ou d'un appel de paramètre (appel de fonction).

```
# Oui
    def fonction(paramètre=5):
        ...
    fonction(paramètre=32)

# Non
    def fonction(paramètre = 5):
        ...
    fonction(paramètre = 32)
```

— Il est déconseillé de mettre plusieurs instructions sur une même ligne :

```
# Oui
    if truc == 'chose':
```

```
        faire_truc_chose()
    faire_1()
    faire_2()
    faire_3()

# Plutôt que
    if truc == 'chose': faire_truc_chose()
    faire_1(); faire_2(); faire_3()
```

Commentaires

Les commentaires qui contredisent le code sont pires qu'une absence de commentaire. Lorsque le code doit changer, faites passer parmi vos priorités absolues la mise à jour des commentaires !

— Les commentaires doivent être des phrases complètes, commençant par une majuscule. Le point terminant la phrase peut être absent si le commentaire est court.
— Si vous écrivez en anglais, les règles de langue définies par Strunk and White dans « The elements of style » s'appliquent.
— À l'attention des codeurs non anglophones : s'il vous plaît, écrivez vos commentaires en anglais, sauf si vous êtes sûrs à 120 % que votre code ne sera jamais lu par quelqu'un qui ne comprend pas votre langue (ou que vous ne parlez vraiment pas un mot d'anglais !).

Conventions de nommage

Noms à éviter

N'utilisez jamais les caractères suivants de manière isolée comme noms de variables : `l` (L minuscule), `O` (o majuscule) et `I` (i majuscule). L'affichage de ces caractères dans certaines polices conduit à les confondre avec les chiffres `0` et `1`.

Noms des modules et packages

Les modules et *packages* doivent avoir des noms courts, constitués de lettres minuscules. Les noms de modules peuvent contenir le signe souligné (`_`). Bien que les noms de *packages* soient également autorisés à en contenir, la PEP 8 nous le déconseille.

Noms de classes

Sans presque aucune exception, les noms de classes utilisent la convention suivante : ils sont en minuscules, exceptée la première lettre de chaque mot qui les constitue (par exemple : `MaClasse`).

Noms d'exceptions

Les exceptions étant des classes, elles suivent la même convention. En anglais, si l'exception est une erreur, on fait suivre le nom du suffixe `Error`. Vous retrouvez cette convention dans `SyntaxError`, `IndexError`...

Noms de variables, fonctions et méthodes

Les noms de variables (instances d'objets), de fonctions ou de méthodes suivent une même convention : le nom est entièrement écrit en minuscules et les mots sont séparés par des signes soulignés (`_`). Exemple : `nom_de_fonction`.

Constantes

Les constantes doivent être écrites entièrement en majuscules, les mots étant séparés par un signe souligné (`_`). Exemple : `NOM_DE_MA_CONSTANTE`.

Conventions de programmation

Comparaisons

Les comparaisons avec des singletons (comme `None`) doivent toujours se faire avec les opérateurs `is` et `is not`, jamais avec les opérateurs `==` ou `!=`.

```
# Oui
    if objet is None:
        ...

# Non
    if objet == None:
        ...
```

Quand cela est possible, utilisez l'instruction `if objet:` si vous voulez dire `if objet is not None:`.

La vérification du type d'un objet doit se faire avec la fonction `isinstance` :

```
# Oui
    if isinstance(variable, str):
        ...

# Non
    if type(variable) == str:
        ...
```

Quand vous comparez des séquences, servez-vous du fait qu'une séquence vide est `False`.

```
if liste: # La liste n'est pas vide
```

Enfin, ne comparez pas des booléens à `True` ou `False` :

```
# Oui
    if booléen: # Si booléen est vrai
        ...
    if not booléen: # Si booléen n'est pas vrai
        ...

# Non
    if booléen == True:
        ...

# Encore pire
    if booléen is True:
        ...
```

Conclusion

Voilà pour la PEP 8 ! Elle contient beaucoup de conventions et toutes ne figurent pas ici. Celles que j'ai présentées, dans tous les cas, sont moins détaillées. Je vous invite donc à vous référer au texte original si vous désirez en savoir plus.

La PEP 257 : de belles documentations

Nous allons nous intéresser à présent à la PEP 257, qui définit d'autres conventions concernant la documentation via les *docstrings* : `https://www.python.org/dev/peps/pep-0257/`

```
def fonction(paramètre1, paramètre2):
    """Documentation de la fonction."""
```

La ligne 2 de ce code, que vous avez sans doute reconnue, est une *docstring*. Nous allons exposer quelques conventions autour de leur écriture (notamment comment les rédiger et qu'y faire figurer).

Une fois de plus, je vais prendre quelques libertés avec le texte original de la PEP. Je ne vous en proposerai pas une traduction complète, mais j'insisterai sur les points que je considère importants.

La PEP 257 a été un peu moins largement acceptée par la communauté des développeurs. Vous trouverez de nombreuses autres conventions si vous lisez du code d'autres programmeurs. Là encore, l'important est d'être cohérent au sein du même projet... et de savoir quand être incohérent.

Qu'est-ce qu'une docstring ?

La *docstring* (chaîne de documentation) est une chaîne de caractères placée juste après la définition d'un module, d'une classe, fonction ou méthode. Elle devient l'attribut spécial `__doc__` de l'objet.

```
>>> fonction.__doc__
'Documentation de la fonction.'
>>>
```

Tous les modules doivent être documentés ainsi. Les fonctions et classes exportées par un module doivent également l'être. Cela vaut aussi pour les méthodes publiques d'une classe (y compris le constructeur `__init__`). Un *package* peut être documenté via une *docstring* placée dans le fichier `__init__.py`.

Pour des raisons de cohérence, utilisez toujours des guillemets triples """ autour de vos *docstrings*, surtout si votre chaîne comporte des barres obliques inverses.

Ces chaînes peuvent être écrites sous deux formes :

— sur une seule ligne ;
— sur plusieurs lignes.

Les docstrings sur une seule ligne

```
def kos_root():
    """Return the pathname of the KOS root directory."""
    global _kos_root
    if _kos_root: return _kos_root
    ...
```

Notes

— Les guillemets triples sont utilisés même si la chaîne tient sur une seule ligne. Ainsi, il est plus simple de l'étendre par la suite.
— Dans une chaîne d'une seule ligne, les trois guillemets """ de fin sont sur la même ligne que les trois du début.
— Il n'y a aucun saut de ligne avant ou après la docstring.
— La chaîne de documentation est une phrase ; elle se termine par un point.
— La *docstring* sur une seule ligne *ne doit pas* décrire la signature des paramètres à passer à la fonction/méthode, ou son type de retour. N'écrivez pas :

```
def fonction(a, b):
    """fonction(a, b) -> list"""
```

Cette syntaxe est uniquement valable pour les fonctions C (comme les *built-in*). Pour les fonctions Python, on se sert de l'introspection pour déterminer les paramètres attendus. L'introspection ne peut cependant pas être utilisée pour déterminer le type de retour de la fonction/méthode. Si vous voulez le préciser, incluez-le dans la documentation sous une forme explicite :

```
    """Fonction faisant cela et renvoyant une liste."""
```

Bien entendu, « faisant cela » doit être remplacé par une description utile de ce que fait la fonction !

Les docstrings sur plusieurs lignes

Les chaînes de documentation sur plusieurs lignes sont constituées d'une première ligne résumant brièvement l'objet (fonction, méthode, classe, module), suivie d'un saut de ligne, puis d'une description plus longue. Respectez autant que faire se peut cette convention : une ligne de description brève, un saut de ligne puis une description plus longue.

La première ligne peut se trouver juste après les guillemets ouvrant la chaîne ou juste en-dessous. Dans tous les cas, le reste de la *docstring* doit être indenté au même niveau que la première ligne :

```
class MaClasse:
    def __init__(self, ...):
        """
        Constructeur de la classe MaClasse.

        Une description plus longue...
        sur plusieurs lignes...

        """
```

Insérez un saut de ligne avant et après chaque *docstring* documentant une classe.

La chaîne de documentation d'un module doit généralement dresser la liste des classes, exceptions et fonctions, ainsi que des autres objets exportés par ce module (une ligne de description par objet). Cette ligne de description donne généralement moins d'informations sur l'objet que sa propre documentation. La *docstring* d'un *package* (dans le fichier `__init__.py`) doit également dresser la liste des modules et sous-*packages* qu'il exporte.

La documentation d'une fonction ou méthode doit décrire son comportement, ses arguments, sa valeur de retour, ses effets de bord, les exceptions qu'elle peut lever et les restrictions concernant son appel (quand ou dans quelles conditions l'appeler). Les paramètres optionnels doivent également être documentés.

```
def complexe(reel=0.0, image=0.0):
    """Forme un nombre complexe.

    Paramètres nommés :
    reel -- la partie réelle (0.0 par défaut)
    image -- la partie imaginaire (0.0 par défaut)

    """
    if image == 0.0 and reel == 0.0: return complexe_zero
    ...
```

La documentation d'une classe doit, de même, décrire son comportement, documenter ses méthodes publiques et ses attributs.

L'auteur de la PEP nous conseille de sauter une ligne avant de fermer nos *docstrings* quand elles sont sur plusieurs lignes. Les trois guillemets de fin sont ainsi sur une ligne vide par ailleurs.

```
def fonction():
    """Documentation brève sur une ligne.

    Documentation plus longue...

    """
```

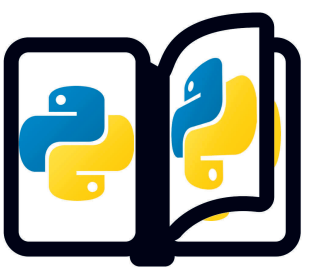
Annexe 41

Installer et gérer nos dépendances en Python

Difficulté :

Un programmeur débutant code ; un programmeur expérimenté utilise le code des autres.

Cette annexe se concentre sur les dépendances en Python : savoir quand utiliser le code d'autres personnes ou équipes en profitant du partage libre (philosophie `open-source`), comment les installer et les utiliser efficacement. Bien qu'optionnelle, cette annexe est peut-être la plus importante de ce livre.



Pourquoi apprendre à utiliser des dépendances en Python ?

Le principe exposé dans l'introduction peut sembler assez arbitraire. Toutefois, bien que simplifié, il est assez vrai dans n'importe quel langage : apprendre à programmer consiste à apprendre à coder par soi-même mais, plus le temps passe, plus les projets s'étendent et se complexifient, plus on gagne du temps si on sait comment utiliser le code des autres. Au lieu de construire toute la machine avec ses engrenages complexes, on récupère différents morceaux à différents endroits et on se contente souvent de quelques lignes de code pour permettre à ces morceaux de fonctionner ensemble.

Il y a plusieurs avantages à savoir exploiter le code qui existe :

— Gain de temps : avec de l'expérience et surtout de l'habitude, il devient très facile de gagner en temps dans nos tâches de programmeur. On code bien moins et on arrive bien plus vite à un résultat appréciable.
— Gain en performances : le code écrit par d'autres pour un usage précis est souvent bien plus performant, sécurisé, optimisé, que ce que l'on pourra programmer. Certains de ces projets, qui ne constituent qu'une toute petite partie de notre code final, ont des années d'expérience et de contribution, sans parler de dizaines ou de centaines de collaborateurs.

Et, puisqu'une grande partie du code rédigé par les autres est disponible gratuitement, pourquoi ne pas en profiter ?

Par exemple, considérez la bibliothèque standard de Python, dont ce cours présente plusieurs modules. Ces derniers ne sont pas toujours utiles pour effectuer une tâche mais, quand un problème précis se présente, on est content de ne pas avoir à recoder la génération de nombres pseudo-aléatoires (`random`), la gestion des répertoires et fichiers sur le disque dur (`pathlib`) ou l'enregistrement de données sur le disque (`pickle`). De la même façon, il existe des quantités de modules que vous pouvez installer, du code écrit par d'autres programmeurs pour effectuer une tâche plus ou moins précise. N'importe qui peut proposer son code. Une fois en ligne sur l'un des services de partage de dépendances, n'importe qui peut installer ce code.

Cette annexe est un peu longue et ne présente qu'une introduction à certaines des fonctions principales concernant l'installation et la gestion des dépendances en Python. Il s'agit d'un sujet complexe mais qu'il est utile de connaître. Pour preuve, si vous comptez faire votre métier de la programmation Python, sachez que de nombreux recruteurs se baseront sur votre capacité à utiliser du code existant, plutôt que de tout recoder. « Ne réinventons pas la roue » est une phrase que vous risquez d'entendre très souvent, peut-être trop souvent à votre goût. Dans cette annexe, vous apprendrez à installer des dépendances avec `pip` et à les isoler avec `virtualenv`.

Installer des dépendances avec `pip`

La bibliothèque standard de Python est assez impressionnante en l'état. Je vous conseille de regarder sa page d'accueil si ce n'est pas déjà fait ; il y a beaucoup de modules : `https://docs.python.org/fr/3/library/index.html` Pourtant, il arrive qu'on en veuille plus, que la bibliothèque standard ne propose pas tout à fait ce que l'on veut. Et parfois, utiliser des dépendances externes est bien plus simple.

Pour prendre un exemple, nous allons installer et utiliser une petite bibliothèque permettant de créer un serveur web minimaliste en Python, `Flask` : `http://flask.pocoo.org/docs/1.0/installation/`

Une fois lancé avec `Flask`, vous pourrez rediriger votre navigateur (Mozilla Firefox, Google Chrome, Edge, Safari...) vers votre serveur web et construire un site qui sera accessible depuis Internet (mais c'est une étape un peu plus complexe que nous ne détaillerons pas ici).

J'ai choisi `Flask` pour l'exemple à cause de sa simplicité : il n'y a pas besoin de beaucoup de code pour l'utiliser. En revanche, pour créer un serveur web en Python, il y a de très nombreuses alternatives qui sont plus puissantes, mais plus difficiles à utiliser. Si ce sujet vous intéresse, je vous recommande `Django` qui possède à présent sa documentation en français, dont un tutoriel complet : `https://www.djangoproject.com/` `https://docs.djangoproject.com/fr/3.2/intro/tutorial01/`

Avant de commencer

Avant de jouer avec `pip`, il est bon de comprendre ce que l'on va faire. `Flask` n'est pas installé sur notre machine actuellement :

```
1 Python 3.10.0 (tags/v3.10.0:b494f59, Oct  4 2021, 18:46:30)
   ↪  [MSC v.1929 32 bit (Intel)] on win32
2 Type "help", "copyright", "credits" or "license" for more
   ↪  information.
3 >>> import flask
4 Traceback (most recent call last):
5   File "<stdin>", line 1, in <module>
6 ModuleNotFoundError: No module named 'flask'
7 >>>
```

Il faut donc le récupérer depuis Internet et l'installer dans Python. Ces étapes sont automatiquement gérées par `pip`. Ce dernier est un outil installé par défaut dans Python. C'est le choix officiel quand il s'agit d'installer des dépendances. La gestion des dépendances n'ayant pas été une des priorités immédiates du langage, il existe plusieurs outils pour installer des paquets sous Python. Ils s'utilisent différemment, ont des points forts et des points faibles. `pip` est sans doute le choix le plus fréquent, mais ne soyez pas

surpris si vous rencontrez une bibliothèque qui doit être installée d'une autre manière ; il est très facile de trouver des instructions d'installation dans tous les cas.

Nous allons utiliser `pip` pour installer `flask`. Il va interroger `Pypi` (prononcer *paille-paille*) : `https://pypi.org` `Pypi` est l'un des principaux serveurs hébergeant les dépendances en Python. S'il trouve `flask`, il va en installer la version correspondant à notre version de Python. On peut lui demander d'installer une version précise de la bibliothèque, mais nous ne verrons pas cela pour l'instant. Après l'installation, nous ouvrirons de nouveau Python et importerons `flask` pour l'utiliser.

La commande `pip`

Le plus simple est d'appeler Python en lui demandant d'exécuter `pip`. On pourrait appeler la commande `pip` directement, mais cela n'est pas nécessairement portable (en particulier sous Windows) et peut nécessiter un peu de configuration supplémentaire. Ouvrez donc une console pour commencer :

— Sous Windows, le plus simple est sans doute de cliquer-droit sur le menu `Démarrer` et de choisir l'option pour ouvrir une console (invite de commande). Sous Windows 10, c'est `PowerShell` qui se lance ; sur d'autres versions du système, ce sera probablement `cmd.exe`, mais le résultat est à peu près le même pour nous.
— Sous Linux, vous ne devriez pas avoir besoin d'instructions pour ouvrir la console, si, en vérité, ce n'est pas déjà le cas.

Une fois dans la console, vérifiez que tout marche bien en saisissant la commande suivante :

```
1 python --version
```

Si vous voyez une version débutant par 3.10 (ou ultérieure), tout va bien. Sous Linux, il est préférable de préciser le numéro de Python directement dans la commande : pour tous les exemples suivants, remplacer les commandes `python...` par `python3...` (adaptez en fonction de votre version).

Vérifiez que `pip` est installé et fonctionnel :

```
1 python -m pip --version
```

Sur mon PC, la console me répond :

```
1 pip 21.3.1 from C:\...\lib\site-packages\pip (python 3.10)
```

On a demandé à Python d'exécuter `pip` et d'afficher sa version ; ici, c'est la 21.3. La bonne nouvelle, c'est qu'il est très facile de mettre `pip` à jour... avec lui-même ! Si vous cherchez à installer une dépendance, `pip` vous proposera peut-être de se mettre à jour en vous donnant la commande précise à entrer.

Installons `flask`

Maintenant que l'on sait comment utiliser `pip`, une commande suffit pour installer `flask` :

```
python -m pip install flask
```

- `python` (ou `python3.10` sous Linux) appelle l'interpréteur Python.
- `-m` est une option pour signaler qu'au lieu d'un script, on veut appeler un module interne.
- `pip` indique que l'on veut appeler le module `pip`.
- `install` est la commande transmise à `pip`. Elle indique que nous voulons installer une dépendance.
- `flask` est le nom de la dépendance à installer.

Entrez cette commande dans votre console et validez. Vous devriez voir plusieurs lignes indiquant que `pip` interroge `Pypi`, télécharge la version de `flask` compatible avec Python 3.10 et l'installe :

```
Collecting flask
Downloading https://files.pythonhosted.org/packages/7f/e7/
   08578774ed4536d3242b14dacb4696386634607af824ea997202cd0edb4b
   /Flask-1.0.2-py2.py3-none-any.whl (91kB)
Collecting itsdangerous>=0.24 (from flask)
Downloading https://files.pythonhosted.org/packages/76/ae/
   44b03b253d6fade317f32c24d100b3b35c2239807046a4c953c7b89fa49e
   /itsdangerous-1.1.0-py2.py3-none-any.whl
Collecting Jinja2>=2.10 (from flask)
Downloading https://files.pythonhosted.org/packages/1d/e7/
   fd8b501e7a6dfe492a433deb7b9d833d39ca74916fa8bc63dd1a4947a671
   /Jinja2-2.10.1-py2.py3-none-any.whl (124kB)
Collecting Werkzeug>=0.14 (from flask)
Downloading https://files.pythonhosted.org/packages/18/79/
   84f02539cc181cdbf5ff5a41b9f52cae870b6f632767e43ba6ac70132e92
   /Werkzeug-0.15.2-py2.py3-none-any.whl (328kB)
Collecting click>=5.1 (from flask)
Downloading https://files.pythonhosted.org/packages/fa/37/
   45185cb5abbc30d7257104c434fe0b07e5a195a6847506c074527aa599ec
   /Click-7.0-py2.py3-none-any.whl (81kB)
Collecting MarkupSafe>=0.23 (from Jinja2>=2.10->flask)
Downloading https://files.pythonhosted.org/packages/5b/d4/
   1deb3c5dc3714fb160c7e2116fc6dff36a063d9156a9328cce54ef35cc52
   /MarkupSafe-1.1.1-cp37-cp37m-win32.whl
Installing collected packages: itsdangerous, MarkupSafe, Jinja2, Werkzeug,
↪  click, flask
Successfully installed Jinja2-2.10.1 MarkupSafe-1.1.1 Werkzeug-0.15.2
↪  click-7.0 flask-1.0.2 itsdangerous-1.1.0
```

Si tout va bien, après un moment dépendant de votre connexion Internet, vous devriez voir la ligne confirmant l'installation.

Je voulais installer `flask` et il a installé plein d'autres choses.

Une bibliothèque a souvent des dépendances vers d'autres, qui peuvent avoir, à leur tour, des dépendances vers d'autres... C'est une raison pour laquelle un utilitaire comme `pip` simplifie les choses : il s'occupe de télécharger ce dont il a besoin et de l'installer. Il y a, hélas, quelques problèmes avec cette stratégie, que nous discuterons dans la section suivante. En attendant, il suffit de savoir que `flask` dépend d'autres bibliothèques et que `pip` les installe toutes. Rassurez-vous, elles ne sont pas extrêmement lourdes.

Tester `flask`

successfully est toujours un bon signe, mais autant en avoir la preuve, non ? Ouvrez votre interpréteur Python (vous pouvez vous contenter d'entrer la commande `python`, ou `python3.10` sous Linux) :

```
Python 3.10.0 (tags/v3.10.0:b494f59, Oct  4 2021, 18:46:30)
 ↪  [MSC v.1929 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more
 ↪  information.
>>> import flask
>>>
```

Pas d'erreur cette fois : Python trouve bien la bibliothèque `flask` et l'importe sans erreur. Maintenant qu'elle est installée, autant l'utiliser.

Laissez la console ouverte, vous en aurez besoin de nouveau. Dans le dossier où elle se trouve, créez un fichier appelé `app.py`. Ce n'est pas un nom arbitraire ; il simplifiera la recherche de Flask. Si vous n'avez pas le droit de créer un fichier à cet endroit, n'hésitez pas à changer de dossier dans la console, en utilisant `cd`.

Sous Windows, il peut être plus facile de fermer la console, ouvrir votre explorateur de fichiers, aller dans un dossier où vous avez le droit d'écrire, créer un fichier `app.py` puis, dans l'explorateur, sélectionner la barre d'adresse et entrer `cmd` avant de valider. Cela ouvrira la console dans le même dossier, ce qui est souvent bien pratique.

Dans le fichier `app.py`, placez le code suivant :

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Bonjour, de la part de <strong>Flask</strong> !"
```

Sauvegardez et fermez le fichier. Dans la console, lancez le serveur web :

```
python -m flask run
```

(Là encore, sous Linux, précisez plutôt `python3 -m flask run`).

```
 * Environment: production
   WARNING: Do not use the development server in a production
↪   environment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Laissez la console ouverte. Ouvrez votre navigateur web favori. Dans la barre d'adresse, entrez `http://127.0.0.1:5000/`.

Vous devriez voir une page blanche avec notre message « Bonjour, de la part de **Flask** ! » Ce dernier mot devrait être écrit en gras, sur la plupart des navigateurs, car nous avons écrit une balise HTML autour du nom dans notre code Python.

Retournez dans la console et appuyez sur CTRL + C pour fermer le serveur web. Expliquer le code Python, le code HTML généré ou Flask plus avant n'est pas le sujet ici. Remarquez simplement que nous avons installé notre première dépendance Python et que nous avons pu l'utiliser sans difficulté. Mission accomplie !

Si vous n'avez plus besoin de Flask, désinstallez-le, toujours avec `pip` :

```
python -m pip uninstall flask
```

On vous demande confirmation, entrez `Y` (pour *yes*) et validez.

Si vous lisez attentivement les messages de retour, vous verrez que `pip` désinstalle Flask, mais qu'il laisse les dépendances de ce dernier. C'est un autre problème de ce type d'installation/désinstallation. Si l'on veut tester de nombreuses bibliothèques, notre installation de Python risque de s'alourdir de bien des choses inutiles. Nous allons maintenant voir un début de solution à ce problème.

Les environnements indépendants

Nous arrivons aux environnements indépendants *(virtualenv)* et je vous préviens tout de suite : c'est assez souvent la raison principale derrière une embauche ou un refus courtois dans les postes indépendants en Python. Savoir ce qu'est un environnement indépendant, comment le configurer et l'utiliser sont des compétences importantes et trop souvent négligées quand on arrive à un niveau intermédiaire de Python.

Qu'est-ce que c'est exactement ?

Un environnement indépendant (appelé *virtualenv* en anglais, ou *venv*) est un dossier contenant ni plus ni moins qu'une copie de votre version de Python installée.

Et... c'est tout ?

Oui. Enfin non ! L'avantage semble mince au premier abord. Pourquoi avoir plusieurs Python installés (surtout s'ils sont de la même version) ?

Une première réponse a été fournie précédemment : quand on commence à travailler sur plusieurs projets qui ont tous plusieurs dépendances, il est préférable de ne pas installer toutes les dépendances de tous les projets au même endroit. Considérez l'exemple suivant :

— Vous travaillez sur un projet, appelé *projet 1*. Il a besoin de Flask dans une version précise (disons 0.12).
— Vous travaillez aussi sur un second projet, *projet 2*. Celui-ci a besoin de Flask... dans une autre version (disons 1.0).

Que faire ?

On pourrait mettre à jour *projet 1* pour utiliser Flask 1.0... non ?

C'est une excellente idée ! Sauf que, parfois, vous ne pouvez pas. Si vous commencez votre travail de développeur en indiquant qu'il est nécessaire de mettre à jour les projets de l'entreprise pour utiliser la version que vous avez installée et éviter des conflits avec vos autres projets, je vous conseille vivement d'avoir de très grands talents de diplomate.

Il y a souvent de très bonnes raisons pour utiliser une version plutôt qu'une autre dans les dépendances. Et les choses se compliquent, car les dépendances ont besoin d'autres dépendances, mais peut-être pas dans la même version...

Bref, vous passez votre temps à réinstaller et désinstaller des dépendances et il est hélas probable qu'on vous signale, avec plus ou moins de tact, que vous n'êtes pas payé pour cela.

Retournons à nos environnements indépendants *(virtualenv)*. En quoi cela aiderait-il d'avoir plusieurs copies de Python sur votre système ?

Eh bien, vous auriez une copie avec les dépendances nécessaires à *projet 1* et une autre avec les dépendances nécessaires à *projet 2*. Les deux fonctionneront sans problème et sans conflit, puisque leurs dépendances (et les dépendances de leurs dépendances) seront installées dans des copies différentes de Python.

J'insiste sur ce point qui est important :

Dès lors que vous travaillez avec des dépendances (si vous utilisez `pip`), apprenez à utiliser des environnements indépendants *(virtualenv)*, même si vous êtes seul sur le projet, même si vous n'avez qu'un seul projet, même si celui-ci n'a qu'une seule dépendance. Et si vraiment vous ne les utilisez pas, au moins comprenez pourquoi.

Voilà ! C'est dit. Maintenant voyons comment cela marche, concrètement.

Créer un environnement indépendant (virtualenv)

La première chose à faire est de créer un environnement indépendant. Cet outil est fort heureusement intégré à Python. Ouvrez simplement une ligne de commande et entrez-y la commande suivante :

```
python -m virtualenv environnement
```

Elle ressemble à celle entrée pour `pip`, n'est-ce pas ? Nous remplaçons simplement `pip` par `virtualenv`. On donne en paramètre le nom du répertoire à créer (`environnement`, dans mon cas).

Cette commande risque de prendre un moment pour s'exécuter. Dans votre console, vous devriez voir quelque chose comme ce qui suit :

```
Using base prefix 'C:\\...\Python310-32'
New python executable in D:\environnement\Scripts\python.exe
Installing setuptools, pip, wheel...
done.
```

Dans le dossier où vous avez entré la commande se trouve à présent un dossier appelé `environnement`. Qu'y a-t-il dedans ?

... beaucoup de choses ! Si vous avez eu la curiosité d'ouvrir le dossier installé de Python, vous trouverez quelque chose d'assez semblable. Inutile de trop s'attarder dans ce dossier ; il est là, utilisons-le, supprimons-le mais, tant qu'il fonctionne, nous n'avons pas besoin de trop savoir comment (à notre niveau tout du moins).

Activer notre environnement indépendant

Il faut maintenant lancer notre environnement indépendant. Dans notre console, au même endroit (ne pas se déplacer dans le dossier `environnement`), entrez l'une des commandes suivantes :

- sous Linux : `source environnement/bin/activate`
- sous Windows : `environnement\scripts\activate`

Vous n'avez aucun message dans la console, mais vous devriez voir votre invite, indiquant que vous pouvez entrer une nouvelle commande. Elle a été modifiée pour préciser entre crochets, au début de la ligne, l'environnement indépendant utilisé :

```
(environnement) D:\>
```

Nous avons activé notre environnement indépendant. Que peut-on faire maintenant ?

Commençons par vérifier que nous sommes au bon endroit. Entrez les commandes suivantes :

```
(environnement) D:\>python -V
Python 3.10.0

(environnement) D:\>pip --version
pip 19.1.1 from d:\environnement\lib\site-packages\pip
 ↪  (python 3.10)
```

Notez le retour de la seconde commande. C'est intéressant. `pip` nous explique qu'il est installé dans notre environnement indépendant.

Vous l'avez sans doute remarqué, nous avons cette fois utilisé `pip` sans la longue commande `python -m pip`. On est certain que les commandes `pip` et `python` font bien référence aux versions installées dans notre environnement indépendant, tant qu'il est actif.

Installer une dépendance dans notre environnement indépendant

Essayons d'installer une dépendance dans notre environnement indépendant actif :

```
pip install django
```

Cela risque de prendre un peu de temps. `Django` n'est pas vraiment une petite bibliothèque ; soyez patients. Après l'opération, vérifions si `Django` s'est bien installé. Dès que l'invite revient, lancez Python :

```
python
```

Attention : l'interpréteur de commande s'ouvre dans votre environnement indépendant. Si vous ouvrez Python d'une autre manière, cela ne sera pas nécessairement vrai. Sous Windows, vous pouvez aussi vous rendre dans le dossier `environnement\scripts` et double-cliquer sur `python.exe`.

Une fois Python ouvert, essayons d'importer `Django` :

```
>>> import django
>>>
```

Impressionnant ! Rien ne se passe ! Enfin... `Django` est importé sans erreur. Maintenant, quittez l'interpréteur de Python pour revenir à votre ligne de commande (CTRL + D sous Linux, `exit()` sous Windows). Une fois de retour dans la ligne de commande, entrez ce qui suit :

```
deactivate
```

Cette commande désactive votre environnement indépendant. Vous ne le voyez plus dans l'invite ; il n'est plus actif. Si vous ouvrez Python de nouveau dans la console, en entrant `python`, et si vous essayez d'importer `Django`, vous obtenez une erreur :

```
>>> import django
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'django'
>>>
```

Donc, nous avons installé `Django` dans notre environnement indépendant mais, comme celui-ci n'est plus actif, on n'a plus accès à cet outil. Pour le retrouver, activez de nouveau l'environnement indépendant avec `source environnement/bin/active` sous Linux, ou `environnement\scripts\activate` sous Windows.

Vous aurez besoin d'activer l'environnement indépendant à chaque utilisation du projet. Retenez donc la commande d'activation.

Quel est l'intérêt du coup ? On entre des commandes en plus pour pas grand-chose.

`Django`, dans notre exemple, est installé directement dans notre copie de Python et pas dans notre système. Cela permet de mieux séparer les dépendances que vous installez et évite de tout stocker dans votre installation de base. Sachez qu'installer de nouvelles dépendances a tendance à légèrement ralentir Python ; il n'est pas judicieux d'installer des milliers de dépendances inutiles sur votre système, d'où les environnements indépendants.

Puis-je supprimer le dossier `environnement` maintenant ?

Si vous n'en avez plus besoin et s'il n'est plus actif, vous pouvez le supprimer sans problème. Pour certains projets auxquels je contribue, j'aime bien supprimer l'environnement indépendant et suivre les installations d'instruction de bout en bout pour m'assurer que les étapes sont correctes.

Les environnements indépendants forment un très vaste sujet. Il est possible de programmer en Python sans savoir comment utiliser un environnement indépendant, sans même savoir ce que c'est. Néanmoins, c'est un outil très pratique qui peut faire la différence dans de nombreux cas, que ce soit pour le travail individuel, la contribution aux projets libres ou le travail professionnel.

Une dernière note : vous avez pu constater que l'environnement indépendant copiait pratiquement tout ce que contient votre Python dans sa version système... dont l'exécutable python lui-même ! Cela signifie que vous pouvez très bien avoir plusieurs versions de Python installées, avec plusieurs environnements indépendants utilisant plusieurs versions différentes.

Annexe 42

Pour finir et bien continuer

Difficulté :

La fin de ce cours sur Python approche. Si ce langage vous a plu, vous aimeriez probablement concrétiser vos futurs projets avec lui. Je vous donne ici quelques indications qui devraient vous y aider.

Ce sera cependant en grande partie à vous d'explorer les pistes que je vous propose. Vous avez à présent un bagage suffisant pour vous lancer à corps perdu dans un projet d'une certaine importance, tant que vous vous en sentez la motivation.

Nous allons commencer par lister quelques-unes des ressources disponibles sur Python, pour compléter vos connaissances sur ce langage.

Nous citerons ensuite plusieurs bibliothèques tierces spécialisées dans certains domaines, qui permettent par exemple de réaliser des interfaces graphiques.

Quelques références

Dans cette section, je vais surtout parler des ressources officielles que l'on peut trouver sur le site de Python : `www.python.org`

Depuis la version 3.7, une partie croissante de la documentation officielle est disponible dans plusieurs langues, incluant le français. Il y a encore à ce jour des parties de la documentation non traduites, mais elles diminuent petit à petit.

Les ressources les plus actualisées se trouvent sur le site de Python lui-même.

En outre, les ressources mises à disposition sont clairement expliquées et détaillées avec assez d'exemples pour comprendre leur utilité.

La documentation officielle

`https://docs.python.org/fr/3/` Cette documentation est constituée de plusieurs sections, dont certaines que vous utiliserez plus souvent que d'autres.

Le tutoriel officiel

Oui, il existe un tutoriel officiel, à présent en français. Il a certains avantages sur le cours que vous venez de lire :

— Il est toujours maintenu et est adapté pour chaque version de Python. Quand une nouvelle version du langage sera disponible, certaines de ses fonctionnalités ne seront probablement pas présentées dans ce livre. Le tutoriel officiel reste à jour.
— Il est plus détaillé : son approche est plus lente et un peu plus théorique à mon sens, mais il donne également bien plus d'information sur chaque sujet. Eh oui, il y a des choses qui ne sont pas dans ce livre (Python est vaste).

Malgré tout, il est probable que vous n'ayez pas besoin de reprendre toutes les bases du langage non plus et vous ne serez pas bloqué pour continuer.

La référence de la bibliothèque standard

Vous vous êtes peut-être déjà rendus sur cette page. Je l'espère, en vérité. Elle comporte la documentation des types prédéfinis par Python, des fonctions natives *(built-in)* et exceptions, mais aussi des modules que l'on peut trouver dans la bibliothèque standard de Python. Ces derniers sont classés par catégories et il est assez facile (et parfois très utile) de survoler la table des matières pour savoir ce que Python nous propose sans installer de bibliothèque tierce.

C'est déjà beaucoup, comme vous pouvez le voir !

Je me flatte par moment de bien connaître Python, mais en survolant la liste des modules de la bibliothèque standard, je découvre presque toujours des choses que j'ignorais.

Cela dit, il existe certains cas où des bibliothèques tierces sont nécessaires. Nous exposerons quelques-uns de ces cas dans la partie suivante, ainsi que quelques bibliothèques utiles dans ces circonstances. Toutefois, il reste d'autres ressources utiles avant cela.

Le wiki Python

Il existe un *wiki* en ligne qui centralise de nombreuses informations autour de Python : `wiki.python.org`

Il n'est pas traduit ; vous devez donc accéder à la page d'accueil en anglais. En outre, gardez à l'esprit que ses informations sont en grande partie fournies ou éditées par les utilisateurs et qu'elles ne sont pas nécessairement à jour si vous utilisez une version récente de Python.

Une fois là, vous avez accès à une vaste quantité d'informations classées par catégories. Je vous laisse explorer si vous êtes intéressés.

L'index des PEP (Python Enhancement Proposal)

Vous pouvez également consulter l'index des PEP, les propositions d'amélioration dont nous avons parlé précédemment. Sachez que, pour que Python soit modifié dans sa version suivante, la première chose à faire est de proposer ces changements et qu'ils soient validés dans ces documents. Les PEP contiennent donc des informations utiles sur l'évolution de Python ou l'utilisation de certains outils (propositions à l'origine, réalités dans votre version de Python peut-être). `www.python.org/dev/peps/`

Un ensemble de tableaux reprend les PEP classées par catégories. Comme vous le constaterez, il y en a beaucoup et, dans ce livre, je n'ai pu vous en présenter que quelques-unes. Parcourez cet index et penchez-vous sur certaines des PEP en fonction des sujets qui vous intéressent plus particulièrement.

La documentation par version

Un dernier mot sur le sujet de la documentation officielle : les liens que j'ai donnés dans ce livre dirigent vers la dernière version de Python. Si vous commencez à utiliser ce langage pour une tâche précise, il est possible que vous ne vouliez tout simplement pas le mettre à jour. Dans ce cas, prenez bien soin de changer la version sur la page de la documentation officielle, pour avoir toujours la documentation correspondante.

Des bibliothèques tierces

La bibliothèque standard de Python comporte déjà beaucoup de modules et de fonctionnalités. Il arrive pourtant, pour certains projets, qu'elle ne suffise pas.

Si vous avez besoin de créer une application avec une interface graphique, la bibliothèque standard vous propose un module appelé `tkinter`. Il existe toutefois d'autres moyens de créer des interfaces graphiques, en faisant appel à des bibliothèques tierces.

Ces dernières se présentent comme des *packages* ou modules que vous installez pour les rendre accessibles depuis votre interpréteur Python.

Ceci étant posé, examinons-en quelques-unes. Il en existe un nombre incalculable et, naturellement, je n'en présente ici qu'une petite partie.

Pour créer une interface graphique

Nous avons parlé de **`tkinter`**. Il s'agit d'un module disponible par défaut dans la bibliothèque standard de Python. Il se base sur la bibliothèque `Tk` et permet de développer des interfaces graphiques.

Il est cependant possible que ce module ne corresponde pas à vos besoins. Il existe plusieurs bibliothèques tierces pour développer des interfaces graphiques, parfois avec quelques bonus. En voici trois parmi d'autres :

— **`PyQT`** est une bibliothèque pour développer des interfaces graphiques, actuellement en version 5. En outre, elle propose plusieurs *packages* gérant le réseau, le SQL (bases de données), un kit de développement web… et bien d'autres choses. Soyez vigilants cependant : elle est distribuée sous plusieurs licences, commerciales ou non. Vous devrez tenir compte de ce fait si vous commencez à l'utiliser. `https://riverbankcomputing.com/software/pyqt/intro`

— **`PyGObject`** (anciennement `PyGTK`) : comme son ancien nom l'indique, elle fait le lien entre Python et la bibliothèque `GTK`/`GTK+`. Elle est distribuée sous licence LGPL. `https://pygobject.readthedocs.io/en/latest/`

— **`wxPython`** fait le lien entre Python et la bibliothèque `WxWidget`. `www.wxpython.org`

Ces informations ne vous aident pas à faire un choix immédiat entre telle ou telle bibliothèque, j'en ai conscience. Aussi, je vous invite à consulter les sites de ces différents projets.

Ces trois bibliothèques ont l'avantage d'être disponibles sur de nombreuses plates-formes et, généralement, assez simples à apprendre. En fonction de vos besoins, vous vous tournerez plutôt vers l'une ou l'autre.

Dans le monde du Web

Il existe là encore de nombreuses bibliothèques, bien que je n'en présente ici que deux. Elles permettent de créer des sites web et concurrencent des langages comme PHP.

`Django`

La première, que nous avons brièvement citée au chapitre précédent, est `Django`. Sa documentation est en français. `https://docs.djangoproject.com/fr/`

`Django` est une bibliothèque, ou plutôt un *framework*, permettant de développer votre site dynamique en Python. Il propose de nombreuses fonctionnalités que vous trouverez, je pense, aussi puissantes que flexibles si vous prenez le temps de vous pencher sur le site du projet. `Django` propose une interface native à vos bases de données. Il permet de créer tous les aspects d'un site web (que ce soit un site vitrine ou très avancé). Et il dispose d'un excellent tutoriel (en français). Notez que, dans l'adresse suivante, la version est 3.2. Il est probable qu'une version plus récente existe quand vous lirez ces pages : `https://docs.djangoproject.com/fr/3.2/intro/tutorial01/`

Flask

`Flask` est une autre bibliothèque pour créer votre site web avec Python. Il est bien plus léger que `Django` et facile à construire pour des petits sites. Certains le préfèrent même pour des sites avancés. C'est une question de goût. Sans contredit, il est bien plus facile de démarrer avec `Flask` : `flask.pocoo.org`

Un peu de réseau

Pour finir, nous allons parler d'une bibliothèque assez connue, appelée `Twisted` : `https://twistedmatrix.com`

Elle prend en charge de nombreux protocoles de communication réseau (TCP et UDP, bien entendu, mais aussi HTTP, SSH et de nombreux autres). Si vous voulez créer une application utilisant l'un de ces protocoles, `Twisted` pourrait être une bonne solution.

Pour conclure

Ce ne sont là que quelques bibliothèques tierces ; il en existe de nombreuses autres, certaines dédiées à des projets très précis. Je vous invite à lancer des recherches plus avancées si vous avez des besoins plus spécifiques. Vous pouvez commencer avec les bibliothèques que j'ai citées, avec les réserves suivantes :

— Je ne donne que peu d'informations sur chaque bibliothèque et elles ne s'accordent peut-être plus avec celles disponibles sur le site du projet. En outre, la documentation de chaque bibliothèque reste et restera, dans tous les cas, une source plus sûre et actuelle.
— Ces projets évoluent rapidement. Il est fort possible que les informations que je fournis ne soient plus vraies à l'heure où vous lirez ces lignes. Pour les mettre à jour, il n'y a qu'une seule solution imparable : allez sur le site du projet !

Une dernière petite parenthèse avant de vous quitter : je me suis efforcé de présenter, tout au long de ce livre, des données utiles et à jour sur le langage de programmation

Python, dans sa branche 3.X. Il vous reste encore de nombreuses choses à découvrir sur le langage et ses bibliothèques, mais vous êtes désormais capables de voler de vos propres ailes. Bonne route ! ;-)

Index