Programmation avec Python : des jeux au Web

Robert Godin, Daniel Lemire

Troisième édition, juillet 2024

Avant-propos

Ce livre introduit les concepts fondamentaux de la programmation et du langage Python. Il vise un public large et ne nécessite pas de connaissances préalables en programmation. Le manuel peut être utilisé dans un cours d'introduction à la programmation et au langage Python. Il peut aussi servir à débutant curieux d'apprendre les mécanismes de base de l'animation par ordinateur employé dans les jeux vidéo. Il peut aussi servir comme préalable à un cours visant l'analyse de données et l'apprentissage machine, qui est l'objet d'un second livre développé dans cette optique.

Le manuel est aussi conçu de manière à être bénéfique aux lecteurs qui ont déjà une expérience de programmation dans un autre langage que Python. Plusieurs sujets relativement avancés viennent compléter le matériel, comme la programmation asynchrone.

Le langage Python est devenu populaire pour le développement d'applications Web, l'analyse des données, l'apprentissage machine jusqu'aux récents développement en apprentissage profond qui sont à l'origine d'applications innovantes en intelligence artificielle.

Ce livre a beaucoup de similarités avec le livre d'introduction à la programmation avec le langage Java : Java pas à pas, (Godin et Lemire, 2023, https://github.com/RobertGodin/JavaPasAPas)[1] Un grand nombre des exemples en sont inspirés. L'approche proposée consiste à lire des exemples de code et à pratiquer au moyen d'exercices qui permettent d'apprivoiser graduellement les différentes notions de manière concrète. Pour ceux qui ont déjà étudié le manuel Java pas à pas, le présent manuel sera un mélange de problèmes familiers et de nouvelles expériences notamment en ce qui a trait au Web.

La version électronique du livre contient un grand nombre de liens Web (adresse de page URL) qui peuvent être suivis pour obtenir plus d'information. Le lien vers le répertoire GitHub suivant contient le code des exemples et des exercices :

https://github.com/RobertGodin/CodePython

[1] En Word, une portion de texte souligné et de couleur bleue représente un lien Web qui peut être accédé en cliquant dessus avec la touche <CTRL> enfoncée.

Transparents

Des transparents destinés à l'enseignement sont aussi disponibles sur le répertoire GitHub pour chacun des chapitres. Ces transparents contiennent la majorité des figures qui apparaissent dans le livre.

Remerciements

Nous tenons à remercier tous ceux qui ont participé de près ou de loin à la réalisation de ce livre. La pandémie n'a pas eu que de mauvais côtés. Un gros merci à notre collègue Louis Martin qui, avec sa vaste expertise en programmation et en génie logiciel, a proposé plusieurs corrections et améliorations.

1 Concepts de base

" L'art de douter est le meilleur secret pour apprendre ", Marcel Prévost

Ce livre introduit la programmation et le langage Python. Ce chapitre présente les concepts et la terminologie de base nécessaires à la compréhension de la programmation.

Un ordinateur doit être programmé pour accomplir une tâche.

Programme, programmation, langage de programmation, logiciel (*software*), matériel (*hardware*)

Un *programme* est un ensemble d'instructions que l'ordinateur exécute afin d'accomplir une tâche. La *programmation* est le processus de production d'un programme. Un programme est exprimé à l'aide d'un *langage de programmation*. Il existe une grande variété de langages de programmation qui ont été développés tels que Java, Python, C, C++, C#, JavaScript, Swift, PHP, Scala, Go, R, etc.

Le terme logiciel (*software*) désigne l'ensemble des programmes, par opposition au matériel (*hardware*) de l'ordinateur qui correspond à l'ensemble des composantes physiques.

Cet ouvrage présente les concepts de base de la programmation, à l'aide du langage de programmation Python. Avant d'aborder la programmation et Python, certaines notions de base des ordinateurs sont nécessaires afin de mieux situer la programmation dans son contexte.

1.1 Composantes matérielles d'un ordinateur (*hardware*)

La description d'un ordinateur faite dans cette section est une introduction à un sujet complexe. Parfois, certains aspects sont simplifiés ou omis afin de s'en tenir à l'essentiel pour la compréhension des concepts de base de la programmation. Un ordinateur est constitué de plusieurs *composantes matérielles* qui collaborent entre elles afin de produire un traitement. L'*architecture matérielle* d'un ordinateur est la description de ses composantes matérielles et de leurs interrelations. Les composantes matérielles d'un ordinateur typique sont illustrées à la Figure 1 : processeur central, mémoire centrale, bus et unités périphériques.

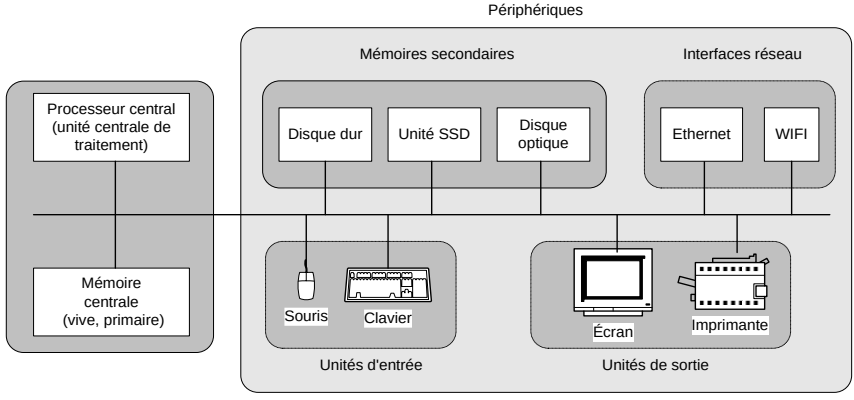


Figure 1. Composantes matérielles (*hardware*) d'un ordinateur typique.

Bus

Le bus permet la communication entre les composantes de l'ordinateur. Une composante branchée au bus peut envoyer des informations à une autre composante ou recevoir des informations d'une autre composante branchée au même bus.

Le diagramme semble suggérer que le bus est simplement un fil. En réalité, un bus est une unité matérielle qui permet les échanges ordonnés entre les différentes composantes qui y sont branchée. Le bus empêche les conflits entre plusieurs composantes qui veulent communiquer en même temps. Les composantes doivent respecter des règles précises pour établir la communication. L'ensemble des règles est appelé le *protocole de communication du bus*.

1.1.1 Processeur et mémoire

Le cœur d'un ordinateur est composé d'un *processeur central* et d'une *mémoire centrale*. Avant qu'un programme ne puisse être exécuté, celui-ci doit être placé dans la mémoire centrale de l'ordinateur.

Mémoire centrale (*main memory*, *primary storage*), principale, vive, primaire ou volatile

La mémoire centrale contient temporairement les *instructions* et *données* d'un programme en cours de traitement. L'ordinateur exécute des instructions placées en mémoire centrale. Ces instructions manipulent des données qui doivent aussi résider en mémoire centrale. Nous verrons plus loin qu'un

programme est habituellement chargé en mémoire centrale à partir d'une unité périphérique (souvent une mémoire secondaire tel que le disque dur) avant d'être exécuté. Les données doivent aussi être chargée en mémoire centrale avant d'être traitée par le programme. La mémoire centrale est constituée d'une séquence de *cases* (*cellules*, *mots*) de taille fixe.

Adresse-mémoire

Une case de la mémoire centrale est identifiée par une *adresse* (*adresse-mémoire*). Dans le cas le plus simple, l'adresse est un entier dans un intervalle de 0 à *n*-1[2], où *n* est la taille de la mémoire centrale. La taille d'une case peut varier selon le processeur utilisé.

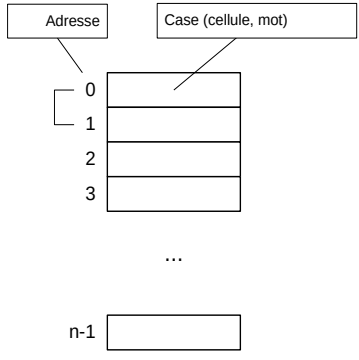


Figure 2. Mémoire centrale.

Les cases de la mémoire centrale peuvent contenir des instructions ou des données. Le contenu des cases peut être modifié par les instructions des programmes. La mémoire centrale est caractérisée par sa rapidité d'accès d'une part et sa volatilité (non-permanence) d'autre part. La vitesse est importante pour que le processeur puisse accéder rapidement aux instructions et aux données en mémoire centrale lors de l'exécution d'un programme. On désigne aussi la mémoire centrale par le terme *Random Access Memory* (RAM) à cause de cette capacité à pouvoir accéder rapidement à n'importe quelle case à tout moment. Sauf pour une petite partie appelée le ROM (*Read Only Memory*), le contenu de la mémoire centrale n'est pas permanent. Il est perdu lorsque le courant électrique qui alimente l'ordinateur est interrompu. Le ROM et les mémoires secondaires permanentes, tel que le disque, peuvent être utilisées pour conserver

[2] Par exemple, si la taille de la mémoire est *n*=16, les cases seront numérotées de 0 à 15. En réalité, le schéma d'adressage peut être plus compliqué...

l'information en permanence au-delà des interruptions de courant. Il faut comprendre que les interruptions de courant ne sont pas toujours volontaires et peuvent provenir, par exemple, d'une panne d'électricité. Il est donc important de placer, en mémoire secondaire ou en ROM, les éléments qui doivent être conservés de manière *persistante*, i.e. survivre aux programmes ou aux anomalies de fonctionnement.

Processeur central (*Central Processing Unit* - CPU), ou Unité Centrale de Traitement (UCT)

Le *processeur central* est la composante qui coordonne l'exécution d'un programme. Il effectue inlassablement le traitement suivant :

Chercher la prochaine instruction en mémoire centrale[3]
Exécuter l'instruction
Chercher la prochaine instruction en mémoire centrale
Exécuter l'instruction
Etc.

Chacune des instructions d'un programme produit un traitement relativement simple. Par exemple, une instruction peut additionner le contenu de deux cases de la mémoire centrale dont les adresses sont *x* et *y*, et placer le résultat dans une troisième case dont l'adresse est *z*. Les données à traiter sont ultimement identifiées par les adresses mémoire mais les langages de programmation utilisent plutôt des identifiants symboliques pour faire référence aux données.

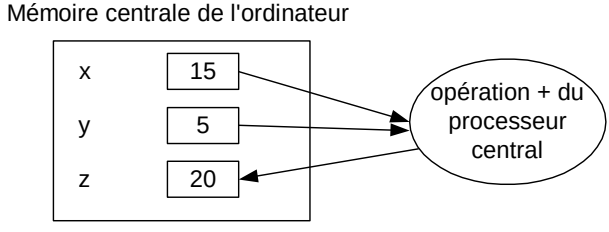


Figure 3. Instruction effectuée par le processeur central.

Le processeur central est lui-même typiquement constitué de deux composantes : l'Unité de Contrôle (UC) et l'Unité Arithmétique et Logique (UAL). L'UC est responsable de déterminer la séquence d'exécution des

[3] La réalité est un peu plus complexe. L'accès à la mémoire centrale peut être géré et accéléré par l'intermédiaire de divers mécanismes tels que la mémoire virtuelle, l'antémémoire (*cache memory*) et les registres.

instructions. L'UAL effectue les calculs arithmétiques et logiques tel que l'addition illustrée ci-haut. La puissance d'un ordinateur vient de sa capacité à exécuter un très grand nombre d'opérations simples à une vitesse extrême.

Un aspect important du fonctionnement de l'ordinateur est la manière de déterminer la *prochaine instruction* à exécuter. Il y a trois structures de contrôle fondamentales :

1. *Séquence*. L'adresse en mémoire centrale de la *prochaine instruction* est normalement celle qui suit l'adresse de l'instruction précédente. Donc, par défaut, les instructions sont exécutées en séquence. Cependant, si c'était toujours le cas, l'unité centrale de traitement exécuterait les instructions jusqu'à ce qu'elle aboutisse à un cul de sac, soit la dernière adresse de la mémoire centrale et ne pourrait continuer !
2. *Répétition (boucle)*. Pour éviter le cul de sac de la séquence, il y a des instructions qui peuvent modifier au besoin l'adresse de la prochaine instruction à exécuter. Par exemple, une instruction peut provoquer le saut à une adresse précédente. En particulier, ceci permet de répéter un ensemble d'instructions.
3. *Décision* (*sélection* ou *choix*). Certaines instructions peuvent choisir l'adresse de la prochaine instruction en fonction d'une condition. Par exemple, si le contenu de la case d'adresse x est 0, sauter à l'adresse y sinon continuer normalement en séquence. C'est ce genre d'instruction qui permet à l'ordinateur de « prendre des décisions » et de modifier son comportement au besoin.

Ces trois manières d'organiser l'exécution des instructions, la séquence, la répétition et la décision, sont des mécanismes de base de la plupart des langages de programmation.

1.1.2 Unités périphériques

Les unités périphériques permettent d'échanger des données entre la mémoire centrale de l'ordinateur et le monde extérieur.

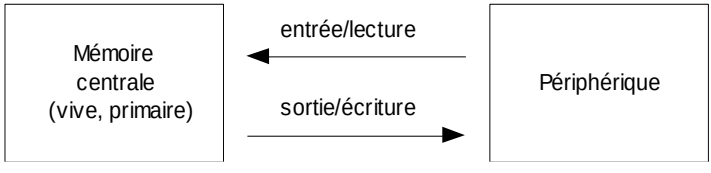


Figure 4. Opération d'entrée/lecture et de sortie/écriture

Entrée, lecture (*input*)

Du point de vue de la mémoire centrale, une opération de transfert d'une information d'une unité périphérique vers la mémoire centrale est appelée une *opération d'entrée* ou de *lecture*.

Sortie, écriture (*output*)

Un transfert inverse de la mémoire centrale vers une unité périphérique est une *opération de sortie* ou d'*écriture*.

Des instructions d'entrées/sorties sont prévues pour déclencher les opérations d'entrée/sortie.

Périphérique d'entrée

Les *périphériques d'entrée* permettent les opérations d'entrée. Par analogie avec l'humain, ce sont en quelque sorte les sens de l'ordinateur. En particulier, certains périphériques permettent à l'ordinateur de recevoir des informations des utilisateurs humains. La *souris*, le *clavier, l'écran tactile, le microphone et la caméra* sont des unités d'entrée bien connus. Au-delà des périphériques d'entrée pour l'humain, toutes sortes de capteurs existent pour saisir des données de diverses natures.

Périphérique de sortie

Les *périphériques de sortie* permettent à l'ordinateur de transmettre de l'information au monde extérieur et en particulier à l'humain. L'*écran* et l'*imprimante* sont des périphériques de sortie typiques. Les *haut-parleurs* ou les *écouteurs* sont aussi très répandus. Des interfaces de toutes sortes existent pour interagir avec des composantes externes de diverse nature.

Les *interfaces réseau* (*modem*, *carte réseau*, ...) et les *mémoires secondaires* (*disque dur*, *mémoire* SSD, *disque optique*, ...) sont aussi des périphériques très répandus. Plusieurs de ces périphériques permettent à la fois les entrées et les sorties. Par exemple, il est possible de lire et d'écrire des informations sur le disque dur. Il est possible d'envoyer et de recevoir des informations par un réseau.

Mémoire secondaire (*secondary storage*), de masse, auxiliaire, permanente, externe, stable, non volatile ou persistante

Une *mémoire secondaire* est une mémoire habituellement plus lente que la mémoire centrale mais qui a la caractéristique d'être permanente. Son contenu ne disparaît pas lorsque le courant électrique est interrompu. Les informations (données et programmes) qui doivent être conservées en permanence seront donc toujours placées en mémoire secondaire. Ce type de mémoire est aussi appelé mémoire *de masse*, *auxiliaire*, *permanente, externe*, *stable*, *non volatile* ou *persistante*.

Il est possible de transférer des informations (données ou programmes) entre une mémoire secondaire et la mémoire centrale par des instructions d'entrées/sorties prévues à cet effet. La mémoire secondaire la plus répandue est l'*unité de disque magnétique* (*disque dur*, *disque rigide* ou tout simplement le *disque*). Le *disque électronique* à semi-conducteurs (SSD - *Solid State Drive*) devient de plus en plus populaire à cause de sa rapidité en lecture par rapport au disque.

Interface réseau

Une interface réseau permet à l'ordinateur de communiquer avec d'autres ordinateurs par l'intermédiaire d'un réseau d'ordinateurs. Il y a différents types de réseaux d'ordinateurs et la manière de brancher un ordinateur à un réseau varie en fonction du type de réseau.

Réseau local (*Local Area Network* - LAN), domestique, d'entreprise, Internet

Un réseau dit *local* permet la communication entre un ensemble limité d'ordinateurs situés à proximité[4] les uns des autres. Un *réseau local domestique* permet la communication entre ordinateurs d'une résidence. Un *réseau local d'entreprise* permet la communication entre les ordinateurs à l'intérieur d'une entreprise. Le réseau *Internet* est un réseau planétaire qui permet la communication avec des ordinateurs répartis autour de la planète. Le réseau *Internet* permet la communication entre les réseaux locaux. C'est donc un

[4] La distance possible entre les ordinateurs d'un réseau local varie en fonction du matériel utilisé.

réseau de réseaux ! Ainsi, un ordinateur branché à un réseau local peut accéder à Internet lorsque le réseau local est lui-même branché à Internet.

Deux interfaces populaires à un réseau local sont l'interface avec fil *Ethernet* ou l'interface sans fils (WIFI). Les appareils mobiles peuvent passer par le réseau cellulaire pour l'accès à Internet.

Le branchement entre les périphériques et le bus suit des conventions basées des normes bien établies tel que ISA, PCI, PCMCIA, SCSI, etc. La pièce maîtresse de l'ordinateur sur laquelle sont installées les différentes composantes de l'ordinateur est la *carte mère* (*mother board*).

1.2 Le logiciel

Un ordinateur fonctionne en exécutant des programmes. On utilise le terme *logiciel* (*software*) pour désigner les programmes. Lorsqu'on démarre un ordinateur[5], il y a un premier programme qui est automatiquement exécuté, appelé le *programme de démarrage* (*boot program*). Ce premier programme est habituellement dans la mémoire centrale à une adresse fixe, connue à l'avance. À cet effet, il y a une petite partie de la mémoire centrale qui est permanente, appelée mémoire morte (*Read Only Memory* - ROM), qui contient le programme de démarrage. Le programme de démarrage a pour rôle essentiel de charger en mémoire centrale un plus gros programme appelé le *système d'exploitation*[6], à partir d'une mémoire secondaire, habituellement le disque dur[7].

[5] On allume l'ordinateur en enfonçant le bouton ON qui est parfois bien caché pour que les non-initiés éprouvent un sentiment d'humiliation la première fois qu'ils essaient de le faire fonctionner. Curieusement, il faut parfois enfoncer le bouton ON pour éteindre certains ordinateurs…

[6] Pour être plus précis, le programme de démarrage inclut déjà certaines parties du système d'exploitation nécessaires pour accéder aux unités périphériques. Dans les ordinateurs PC compatibles, cette portion du système d'exploitation est appelée le *BIOS* (*Basic Input Output System*).

[7] Il est aussi possible de charger le système d'exploitation à partir d'une autre mémoire secondaire (clé USB, disque optique, etc.) mais ceci est habituellement effectué dans des circonstances spéciales, par exemple, lorsqu'un problème survient avec le disque dur.

Système d'exploitation (*Operating System* - OS)

Le système d'exploitation est un programme dont le rôle est de faciliter l'utilisation de l'ordinateur et en particulier des périphériques. Windows, Unix et MacOS sont des exemples de systèmes d'exploitation. Sans système d'exploitation, un ordinateur est aussi utile qu'une aiguille dans une botte de foin[8].

Le système d'exploitation organise l'exécution d'autres programmes appelés *programmes d'application* ou plus simplement *applications*. Après le chargement du système d'exploitation en mémoire centrale, plusieurs tâches d'initialisation peuvent être exécutées afin de préparer l'utilisation du système et démarrer diverses applications. Par la suite, le système se met en état d'attente d'une commande de l'utilisateur. À ce moment, l'utilisateur peut commander l'exécution d'un programme d'application ou interagir avec une application préalablement démarrée.

Interface système (*shell*), interface en ligne de commande (*Command Line Interface*- CLI), interface à l'utilisateur graphique (*Graphical User Interface* - GUI)

Les systèmes d'exploitation offrent habituellement deux types d'interface permettant d'invoquer leurs fonctions. Les *interfaces en ligne de commande* (*Command Line Interface* – CLI) ou les *interfaces à l'utilisateur graphiques* conviviales (*Graphical User Interface* - GUI) basées sur l'utilisation de fenêtres, de menus, de boutons, etc. Dans le cas de Windows, l'interface en ligne de commande est aussi appelée *fenêtre de commande Windows* (*Command Prompt*) et Windows est l'interface graphique. Dans les systèmes d'exploitation UNIX, par analogie avec une noix, il est d'usage d'employer le terme *shell* (*coquille*) pour désigner une interface système qui permet de faire exécuter des commandes du *noyau* (*kernel*) du système d'exploitation.

[8] Pourquoi la cherche-t'on ?

Programme d'application (ou simplement application)

Les programmes d'applications accomplissent des tâches diverses. Par exemple, parmi les applications d'usage courant, il y a les *programmes de traitement de textes* (e.g. Bloc-notes, Wordpad de Windows, Word de Microsoft Office, Emacs de UNIX), les *chiffriers électroniques* (e.g. Excel de Microsoft Office), les *fureteurs WEB* (e.g. Google Chrome, Safari, Microsoft Edge), les *systèmes de courriel* (e.g. Outlook de Microsoft, Google Gmail), les *réseaux sociaux* (e.g. Facebook, Twitter, Instagram, Snapchat, TikTok).

Cet ouvrage traite du développement de programmes d'application en Python. Concrètement, avant qu'il ne soit chargé en mémoire centrale pour être exécuté, un programme d'application se trouve habituellement en mémoire secondaire, par exemple, sur disque dur. Un des rôles importants du système d'exploitation est de permettre d'organiser les programmes et les données placées en mémoire secondaire afin de pouvoir les retrouver.

Fichier (*file*), système de gestion de fichiers (*file system*)

La partie du système d'exploitation qui s'occupe de l'organisation des données et des programmes en mémoire secondaire est le *système de gestion de fichiers* (*file system*). Un *fichier* peut contenir un ou une partie d'un programme ou encore des données pour un programme.

Le terme *document* sert souvent à désigner un fichier de données. En effet, un fichier peut par exemple contenir un texte produit avec un programme de traitement de texte. Le fichier est alors un document contenant des données du point de vue de l'application de traitement de texte. Le système de gestion de fichier permet de regrouper des fichiers dans un *dossier* (aussi appelé *répertoire*). En plus des fichiers, un dossier peut lui-même regrouper d'autres dossiers, ainsi de suite, résultant en une *hiérarchie de dossiers*.

Chemin (*path*) absolu et relatif

Le *chemin* (*path*) d'un fichier ou d'un dossier permet de l'identifier à l'intérieur de la hiérarchie de dossiers. La notion de chemin est aussi rencontrée dans le contexte plus général du Web par exemple, pour identifier une ressource telle qu'une page Web avec une URL (*Uniform Resource Locator*). Le chemin peut être absolu ou relatif. Un chemin absolu part de la racine de la hiérarchie et indique la séquence des dossiers qui mènent à l'élément visé en traversant la hiérarchie. Un chemin relatif part

d'un dossier de travail courant dans le contexte d'une interaction avec le système.

La racine d'un chemin absolu peut être identifiée de différentes manières et les conventions peuvent varier en fonction du système visé. Avec Windows, le cas le plus fréquent identifie la racine par une unité de stockage tel qu'un disque dur ou un disque SSD. La syntaxe « C:\ » identifie la racine de la hiérarchie des dossiers sur l'unité de stockage nommé C. Habituellement, c'est l'unité de stockage de base de Windows. Il peut y avoir plusieurs unités qui sont chacune nommées par une lettre majuscule. Une unité de stockage physique peut aussi être partitionnée en plusieurs volumes, chacun identifié par une lettre.

Le chemin de fichier ou de dossier est constitué de l'identificateur de la racine où se situe l'élément, suivi de la séquence des dossiers à parcourir dans la hiérarchie des dossiers. Les noms des dossiers sont séparés par des barres obliques inverses `\` avec Windows. Ainsi, dans l'exemple de chemin suivant, pour identifier le fichier nommé `HelloWorld.py`, le point de départ est la racine identifiée par `C:\`, suivi du dossier `Users`, suivi ensuite du dossier `Robert`, et enfin du nom du fichier `HelloWorld.py`. Ce fichier contient un programme Python comme nous le verrons par la suite.

```
C:\Users\Robert\HelloWorld.py
```

Le nom d'un fichier est habituellement composé de deux parties séparées par un « `.` ». La partie qui suit le point appelée extension permet de désigner un type de fichier. Dans notre exemple, l'extension `.py` désigne un fichier qui contient du code Python.

Dans le cas d'un système Unix ou IOS, la racine est identifiée par `/` et les noms des dossiers à parcourir sont séparés par des `/`. Ainsi, le chemin précédent serait représenté par :

```
/Users/Robert/HelloWorld.py
```

Un *chemin relatif* part de l'hypothèse qu'un dossier courant est déjà déterminé par suite d'une interaction précédente avec le système. Par exemple, supposons que le dossier courant sous Windows est :

```
C:\Users
```

Le chemin relatif ne spécifie que le chemin qui reste à parcourir à partir du dossier courant, soit :

```
Robert\HelloWorld.py
```

Dans un chemin relatif, le « . » représente le dossier courant. On peut donc utiliser la formulation équivalente :

```
.\ Robert\HelloWorld.py
```

Les systèmes de gestion de fichier modernes permettent le stockage dans des serveurs distants en donnant l'impression d'une unité de stockage locale. De plus en plus, le stockage est effectué dans des serveurs distants partagés en passant par un réseau. D'autres conventions sont employées pour identifier la racine dans ces circonstances. Le concept de nuage informatique (infonuagique) désigne l'ensemble des services distants offerts incluant des services de stockage. De grandes entreprises offrent des services de stockage dans le nuage qui incorporent des mécanismes de sécurité, de fiabilité et de partage plus élaborés que le stockage dans des unités locales.

Parallélisme

Les ordinateurs permettent de faire exécuter plusieurs programmes en même temps. Ceci est possible même dans le cas où il n'y a qu'un seul processeur ! En réalité, dans un système monoprocesseur (à un seul processeur), il n'y a effectivement qu'un seul programme à la fois qui est exécuté. Le système d'exploitation fournit l'illusion d'une exécution simultanée de plusieurs programmes en allouant à chacun des programmes à tour de rôle une petite partie du temps du processeur central. Mais ceci se fait tellement rapidement que l'utilisateur ne peut le percevoir même s'il est un « petit vite ». Le terme *pseudo-parallélisme* désigne cette illusion. Au-delà du pseudo-parallélisme, les ordinateurs modernes sont pourvus de plusieurs processeurs qui permettent donc l'exécution en parallélisme réel de plusieurs traitements. Les unités périphériques sont aussi souvent pourvues de processeurs simples qui peuvent exécuter des opérations d'entrée/sortie en parallèle avec le traitement du processeur central. Pour coordonner les différents traitements entre tous ces processeurs, des mécanismes d'interruption sont prévus afin qu'un processeur puisse signaler à un autre processeur qu'une tâche est terminée, et que ce dernier puisse réagir à cet évènement. Ceci entraîne une interruption du cours normal d'exécution de la boucle de traitement de base. La synchronisation et coordination des différentes tâches parallèles d'un ordinateur peut devenir très complexe et le système d'exploitation fournit des services à cet effet.

1.2.1 Le binaire, le langage machine et le code octet

Les instructions qui composent un programme en mémoire centrale sont exprimées sous forme d'un code appelé le *langage machine*. Ce langage varie

en fonction du type de processeur. Une instruction en langage machine est composée d'une suite de symboles binaires. Un symbole binaire est soit un « 0 » ou un « 1 ». On désigne par *langage binaire*, un langage qui se limite à l'utilisation de deux symboles binaires. Chacun des symboles d'une séquence de symboles binaires (0 ou 1) est appelé un *bit*. Une séquence de 8 bits est appelée un *octet* (*byte*). Chacune des cases de la mémoire peut contenir un nombre fixe de bits, habituellement, 8 bits (1 octet). La taille d'une instruction est un multiple de la taille d'une case mémoire. Il peut même y avoir des tailles d'instruction variables pour un même processeur.

Les données manipulées par un programme sont aussi codées en mémoire sous forme binaire. Par exemple, un nombre entier est souvent représenté par une séquence de 32 ou 64 bits (4 ou 8 octets). Le nombre entier est codé selon un système de numération binaire (base 2). Par exemple, l'entier 25 en décimal est représenté sur huit bits (un octet) par :

$$25_{10} = 00011001_2$$
$$= 0 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$
$$= 0 + 0 + 0 + 16 + 8 + 0 + 0 + 1.$$

Chacun des bits correspond à un exposant en base 2. Pour une représentation plus compacte des nombres binaires dans le contexte de la représentation textuelle, on a souvent recours au système de numération octal (base 8) ou hexadécimal (base 16). Par exemple, l'entier 25 en décimal est représenté en hexadécimal par :

$$25_{10} = 19_{16} = 1 \times 16^1 + 9 \times 16^0 = 16 + 9$$

Les dix chiffres de 0 à 9 ne suffisent pas au système hexadécimal étant donné qu'il faut 16 caractères différents pour chacun des chiffres. La séquence de 0 à 9 est étendue en ajoutant les 6 premières lettres de l'alphabet, soit A à F. Le A correspond à 10 décimal, le B à 11, etc. Par exemple :

$$27_{10} = 1B_{16} = 1 \times 16^1 + 11 \times 16^0 = 16 + 1$$

Les mémoires modernes ont des tailles impressionnantes qui sont habituellement mesurées en employant les conventions du système international détaillées dans le tableau suivant.

Nom	Symbole	Nombre d'octets
kilooctet	Ko	10^3
mégaoctet	Mo	10^6
gigaoctet	Go	10^9
téraoctet	To	10^{12}
pétaoctet	Po	10^{15}
exaoctet	Eo	10^{18}
zettaoctet	Zo	10^{21}
yottaoctet	Yo	10^{24}

En anglais, on emploi les acronymes KB, MB, GB, TB, PB où B correspond à *Byte*. À cause de l'emploi des nombres binaires en informatique, on a traditionnellement employé le préfixe *kilo* pour désigner 2^{10}, *méga* pour 2^{20} et *giga* pour 2^{30}. Depuis 1998, la norme CEI 60027-2 propose plutôt les préfixes du tableau suivant pour les nombres en base 2. Cependant, ces conventions sont plus ou moins respectées en pratique.

Nom	Symbole	Nombre d'octets
kibioctet	Kio	2^{10} = 1 024
mébioctet	Mio	2^{20} = 1 048 576
gibioctet	Gio	2^{30} = 1 073 741 824
tébioctet	Tio	2^{40} = 1 099 511 627 776
pébioctet	Pio	2^{50} = 1 125 899 906 842 624
exbioctet	Ei	2^{60} = 1 152 921 504 606 846 976
zébioctet	Zi	2^{70} = 1 180 591 620 717 411 303 424
yobioctet	Yi	2^{80} = 1 208 925 819 614 629 174 706 176

Une chaîne (suite) de caractères est typiquement représentée en utilisant une séquence d'octets. Les caractères sont codés en suivant une norme de codage. Le code ASCII est une des premières normes employées pour représenter les caractères de base par des codes de la taille d'un octet. L'exemple suivant montre le codage d'une chaîne de caractères. La première ligne montre le code binaire. La deuxième ligne indique les caractères correspondants.

```
01100001 01100010 01100011 00001101 00001010 00110001 00110010 00001101 00001010
```

```
a       b       c       \r      \n      1       2       \r  \n
```

Sous Windows, la fin de ligne est représentée par la séquence des caractères spéciaux ASCII, *retour de chariot* (`\r`) et *saut de ligne* (`\n`)[9]. Le code ASCII ne permet pas de traiter les caractères de toutes les langues. La norme *Unicode* (www.unicode.org) présentée à la section 4.4 est une norme plus générale qui permet d'encoder les caractères d'un grand nombre de langues en employant plus d'un octet au besoin.

Une image est typiquement représentée par un quadrillage de pixels (*picture element*). Pour une image en noir et blanc, chacun des pixels est représenté par un bit qui correspond à la couleur noir ou blanc (0 ou 1). Pour une image en couleur, le système RVB (RGB en anglais) encode chacun des pixels par trois entiers entre 0 et 255 qui correspondent aux trois couleurs, Rouge (*Red*), Vert (*Green*) et Bleu (*Bleu*).

Les premiers ordinateurs étaient programmés directement en langage machine, ce qui était très fastidieux et peu productif. Des langages de programmation plus évolués, tel que Python, sont maintenant employés. Ces langages cachent le codage binaire des données. Les données sont représentées dans une forme interprétable par l'humain. Par exemple, les nombres entiers sont habituellement représentés en décimal.

Cependant, comme l'ordinateur ne comprend pas directement le langage Python, il faut un processus qui permet d'exécuter le code Python sur une machine qui ne comprend que le langage machine binaire. Deux méthodes sont employées : la compilation ou l'interprétation.

Interprète (interpréteur)

Un *interprète* est un programme qui exécute directement les instructions du langage évolué, une à la fois au fur et à mesure qu'elles sont lues.

C'est l'approche qui est privilégiée en apparence avec le langage Python dans son implémentation habituelle. Il faut installer un interprète Python sur un ordinateur avant de pouvoir exécuter un programme Python. Dans un premier temps, il n'est pas nécessaire d'entrer dans le détail du fonctionnement de l'interprète pour programmer en Python. Pour les

[9] La manière de représenter les fins de ligne peut différer en fonction de la plate-forme. Unix, par exemple, emploie uniquement le *saut de ligne* ('\n'). Ceci cause souvent des problèmes lors du transfert de fichiers de texte entre systèmes différents.

curieux, l'encadré suivant donne des détails au sujet de la compilation, de l'interprétation et de l'aspect hybride de Python.

Processus d'interprétation et compilation en code-octet

Un *compilateur* est un programme qui traduit un programme, écrit en un langage évolué, en un programme en langage machine ou autre. Le programme en langage évolué est appelé le *programme source* ou encore *code source*. Et le programme résultant de la traduction est appelé le *programme objet* ou encore le *code objet*. Le code objet peut par la suite être exécuté sur l'ordinateur correspondant. Cette approche en deux étapes est privilégiée pour plusieurs langages de programmation tel que C ou C++.

L'implémentation de l'exécution d'un programme par un *interprète* permet de simplifier le processus d'exécution en évitant l'étape intermédiaire de compilation. L'inconvénient est une exécution moins rapide que dans l'approche avec compilation qui permet d'optimiser l'exécution du code d'une manière plus globale. Entre les deux options de compilation et d'interprétation pures, il y a diverses combinaisons hybrides possibles comme c'est le cas de Python.

Code-octet (*bytecode*) python, machine virtuelle Python

L'implémentation de référence de Python est un hybride entre la compilation et l'interprétation parce que l'interprète Python emploie une étape intermédiaire de compilation. Le *compilateur en code-octet* traduit le code Python en une forme intermédiaire appelée le *code-octet* (*bytecode*). Le code-octet est un langage proche du langage machine qui correspond à une machine abstraite appelée la *Machine Virtuelle Python* (MVP, *Python Virtual Machine* – PVM). La MVP est une machine théorique qui n'existe pas vraiment, du moins pour l'instant. Il manque donc quelque chose pour faire exécuter le programme objet en *code-octet* sur une machine réelle ! Comme le code-octet est proche du langage machine des machines réelles, le processus de traduction du code-octet en code machine est relativement simple à réaliser.

Il y a trois manières d'exécuter les programmes en *code-octet Python* sur la machine réelle visée (voir Figure 5) :

- *Interprète de code-octet*. Un petit programme spécial, appelé *interprète de code-octet*, simule une *MVP* en exécutant le code-octet directement. En d'autres mots ce petit programme donne l'illusion d'une machine dont le langage est le *code-octet*. L'interprète de code-octet est un programme

en langage machine de la machine réelle. C'est cette approche qui est employée par l'implémentation de référence du langage Python (*CPython*) et qui est la plus répandue.

- *Compilateur de code-octet*. Un second niveau de compilation traduit le code-octet en instructions du langage machine de la machine réelle. Un compilateur JIT (*Just-In-Time*) effectue la traduction au moment d'exécuter le code-octet. Cette voie commence à être plus répandue pour Python (e.g. *PyPy*).

- *Processeur Python*. La troisième approche consisterait à construire un processeur dont le langage est le *code-octet* (*code-octet*) Python.

Un avantage du code-octet est l'indépendance du code objet vis-à-vis la machine réelle utilisée. Le même programme objet en code-octet Python peut être exécuté sur n'importe quelle machine réelle pourvu qu'une implémentation de la machine virtuelle Python soit disponible pour l'ordinateur visé.

En conclusion, on dit parfois que Python est un langage interprété mais la réalité est plus complexe. Même lorsque les instructions sont exécutées une à la fois par l'interprète Python, il y a une étape intermédiaire de compilation en code-octet Python. Python est donc à la fois interprété et compilé dans son implémentation de référence. Mais il y a aussi d'autres implémentations sans interprétation.

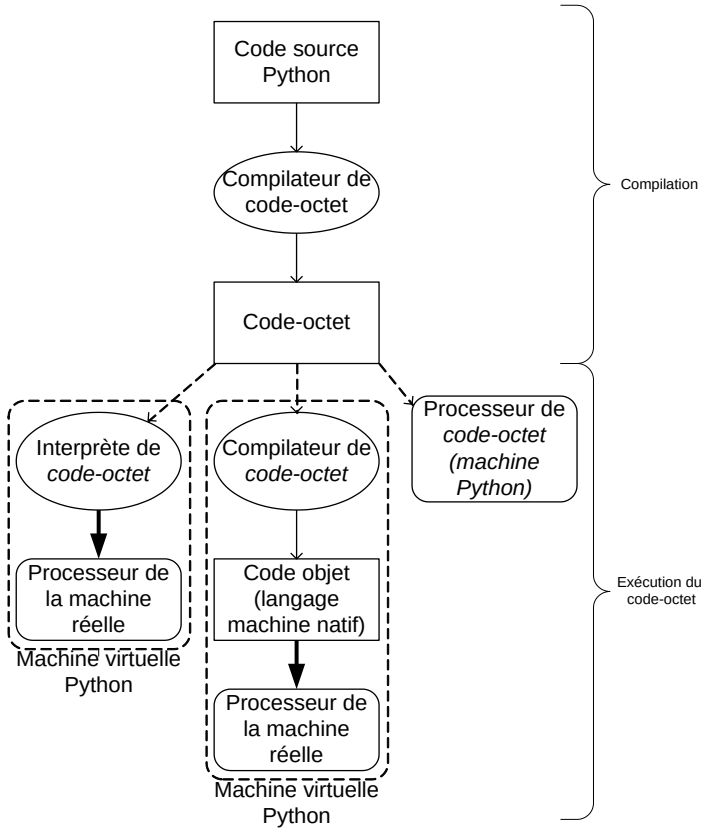


Figure 5. Compilation et exécution d'un programme Python.

1.2.2 Étapes de création et d'exécution d'un programme Python avec Windows

Pour illustrer les concepts précédents de manière concrète, cette section montre comment écrire et faire exécuter un petit programme Python très simple avec le système d'exploitation Windows.

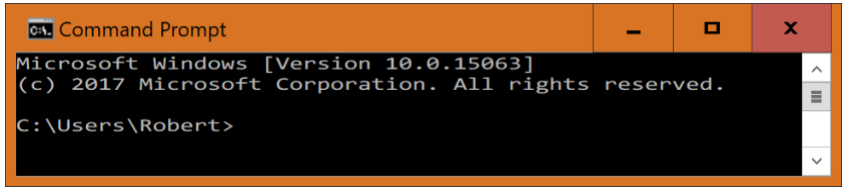
Installation du logiciel Python

Ce scénario suppose que la version 3 de Python est installée sous Windows. Pour plus d'information au sujet des versions disponibles, consultez le site www.python.org. Pour un autre système d'exploitation ou une autre version de Python, les détails des manipulations peuvent varier. La version 2 de Python est souvent déjà installée par défaut sur plusieurs versions des

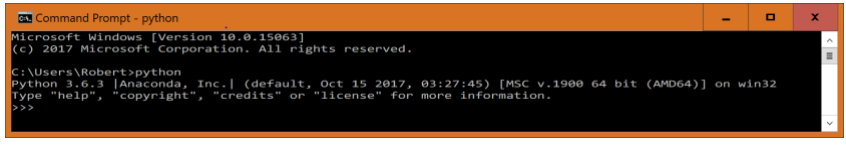
systèmes d'exploitation Linux et macOS. Sous macOS, l'installation du système de Python sur macOS est similaire à Windows : il suffit de vous rendre sur python.org. Cependant il est aussi possible d'installer Python en passant par brew (`https://brew.sh`). Il est recommandé d'installer la version 3 pour éviter les problèmes de compatibilité avec le code de ce livre. Les instructions d'installation sont parfois un peu mystérieuses pour un débutant. Si c'est votre cas, il peut être nécessaire de recourir aux services d'un informaticien expérimenté ou de faire preuve d'un peu de patience ! Un guide de base pour l'installation se trouve à l'adresse suivante : `https://wiki.python.org/moin/BeginnersGuide/Download`

Invocation d'une interface (*shell*) Python (Windows)

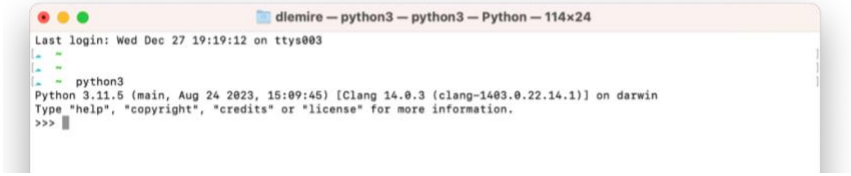
Après avoir installé Python, il est possible de démarrer une interface à l'interprète Python (*Python shell*) en passant par une *fenêtre de commande Windows* (*Command Prompt*).



Il y a différentes manières d'invoquer cette fenêtre de commande, dont le menu des programmes Windows. L'invite de commande Windows est ensuite affichée. Elle est composée du chemin du dossier courant (ici `C:\Users\Robert`) suivi de `>`. Ensuite, une interface Python est démarrée en tapant `python` :

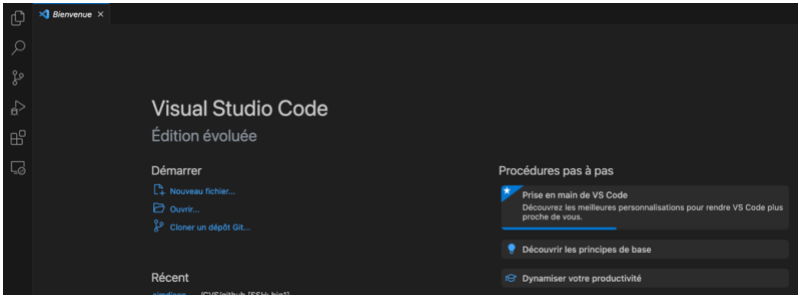


Sous macOS, vous pouvez lancer l'application Terminal qui se trouve dans Finder sous Applications, Utilitaires. Le résultat est similaire à Windows. Notez cependant que pour invoquer la version 3 de Python, sous macOS, il peut être nécesssaire de taper `python3` plutôt que `python`. C'est vrai aussi sous d'autres systèmes où la version 2 de Python est installée par défault.



L'interface Python affiche un message décrivant la version employée.

Pour écrire des programmes en Python, l'interface de la fenêtre de commande n'est pas très pratique. Plusieurs développeurs professionnels utilisent Visual Studio Code comme environnement de développement Python. C'est pour cette raison qu'une fois l'installation de Python complétée, nous vous invitons à installer l'éditeur Visual Studio Code de Microsoft. Il suffit de rendre sur le site https://code.visualstudio.com/ et d'appuyer sur le bouton Télécharger (*Download*). Une fois l'installation de Visual Studio Code effectuée, lancez l'application.



À gauche de la fenêtre principale de Visual Studio Code, vous devriez voir des icônes correspondant à la recherche (loupe) ou à l'ajout d'extensions (carrés). Rendez-vous dans le menu de Visual Studio Code et choisissez « Nouveau Terminal ». Vous devriez alors voir apparaître une sous-fenêtre contenant une console. Une fois la console ouverte, tapez `python` (ou `python3` au besoin) et la touche de retour à la ligne (ou `<fin de ligne>`). Au lancement de Python, vous pourrez consulter la version de Python qui est disponible, assurez-vous qu'il s'agit bien de la version 3. Sur certains systèmes, il peut être nécessaire de taper `python3`.

Note au sujet de l'installation de Python
Si un message d'erreur indique que « python » n'est pas reconnu et que vous êtes sous Windows, il est possible que ce soit parce que le chemin qui mène au programme *python.exe* ne soit pas inclus dans la variable d'environnement *Path* de Windows. Dans ce cas, une option consiste à spécifier le chemin complet du programme Python. Une autre option consiste à inclure le chemin dans la variable d'environnement *Path* qui identifie des chemins de dossiers que le système d'exploitation Windows parcoure afin de retrouver les programmes à exécuter. Cette option est habituellement disponible dans le programme d'installation de Python.

Si la commande n'est pas reconnue, mais que vous avez bien installé Python sous Windows, nous vous suggérons de consulter le site *Utiliser Python sous Windows* :

https://docs.python.org/fr/3/using/windows.html.

Consultez un technicien au besoin.

L'invite de commande `>>>` indique que le *shell* Python est en attente d'un énoncé du langage Python. Le code Python de notre exemple de programme est très simple :

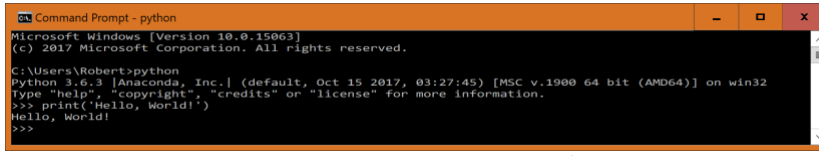
```
print('Hello, World!')
```

Ce genre de programme est une tradition typique dans l'apprentissage d'un nouveau langage de programmation. Il ne fait qu'afficher la phrase « Hello, World! ». Le code Python peut être entré directement suivi de `<fin de ligne>`.

Notation `<fin de ligne>`

La notation `<fin de ligne>` est employée pour représenter la fin de ligne dans le texte de cet ouvrage. La représentation exacte de la fin de ligne peut différer selon le codage employé pour les caractères.

Ceci conduit à l'exécution du code par l'interprète Python :



Le texte « Hello, World ! » est affiché dans la fenêtre. Et voilà un premier programme Python ! L'invite de commande `>>>` est affichée à nouveau indiquant que l'interface est en attente d'une autre commande. La commande `exit()` permet d'arrêter l'interface Python et de revenir à l'invite de commande de Windows.

Il y a d'autres manières d'invoquer l'interprète en fonction de l'installation employée et du système d'exploitation. Des interprètes accessibles par un fureteur Web sont aussi disponibles. Par exemple, le lien https://www.python.org/shell/ invoque une interface à l'interprète Python qui peut être employée pour expérimenter sans devoir l'installer.

Exercice. Saisissez le code de notre exemple « Hello, World ! » en invoquant le *shell* à l'adresse :
https://www.python.org/shell/

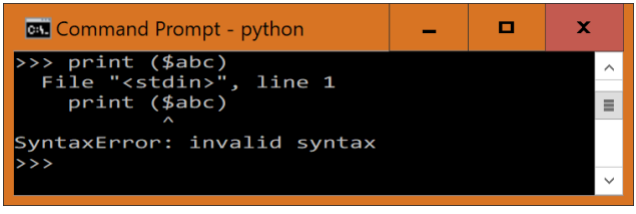
Comme expliqué précédemment, l'exécution passe par une première étape de compilation en code-octet. Le compilateur en code-octet vérifie que le code Python respecte les règles de syntaxe du langage.

Erreur de de syntaxe
Si le programme source est incorrect selon les règles de syntaxe du langage Python, des messages d'erreur sont affichés afin de faciliter le repérage et la correction des erreurs de syntaxe.

Attention !

Les messages d'erreur ne sont pas toujours faciles à déchiffrer et peuvent porter à confusion …

Dans l'exemple suivant, le compilateur a reconnu que la syntaxe du langage Python n'est pas respectée :



Outre les erreurs de syntaxe, il est possible que d'autres types d'erreurs soient rencontrées.

Erreur d'exécution, logique, sémantique (bogue)
Dans un scénario typique de développement d'un programme, en plus des erreurs de syntaxe, il y a souvent des erreurs d'exécution, aussi appelées erreurs logiques ou sémantique (*bogues*) qui peuvent conduire à l'interruption du programme ou à la production d'un résultat incorrect. Il faut alors corriger les erreurs dans le code source (*débogage*).

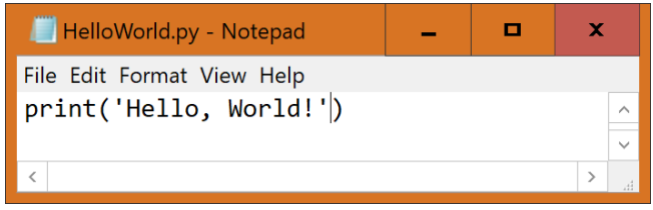
La production d'un résultat correct est un aspect fondamental de la qualité du logiciel et plusieurs aspects de cette facette du développement sont présentés tout au long de cet ouvrage.

1.2.3 Exécution d'un script Python

L'exécution directe de code par l'interprète n'est pas appropriée au développement de logiciels d'une taille non triviale. Dans ce cas, il faut sauvegarder le code source du programme Python dans un fichier avec une

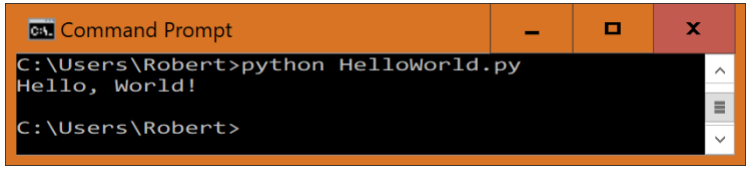
extension « .py » et faire exécuter le code à partir du fichier. Un tel fichier est appelé un *script* Python. Vous pouvez vous y prendre de différentes manières.

Si vous êtes un utilisateur de Windows, vous pouvez utiliser l'outil Notepad. Dans la figure suivante, le texte du script Python qui contient le même code que notre exemple est édité avec l'éditeur de texte *Notepad* de Windows. Le fichier est sauvegardé dans le dossier courant de la fenêtre de commande, `C :\Users\Robert` sous le nom : `HelloWorld.py`



Le script Python est exécuté en tapant à l'invite de commande Windows `python` suivi du nom du fichier :

```
python HelloWorld.py
```

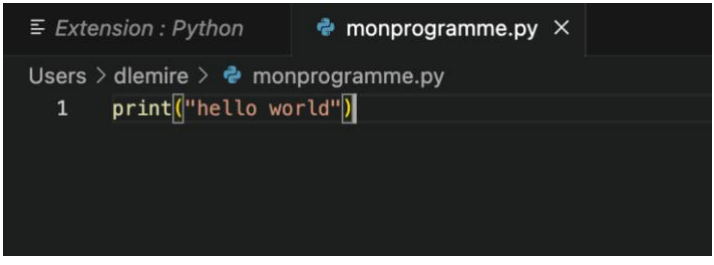


Ceci conduit à l'exécution du script du fichier `HelloWorld.py`. Le texte « Hello, World ! » est affiché. Après l'exécution du script, l'invite de commande Windows est affichée à nouveau. Cette manière de procéder est plutôt rudimentaire et il existe de nombreux outils logiciels qui facilitent le développement des programmes Python. Voir par exemple : https ://realpython.com/tutorials/tools/.

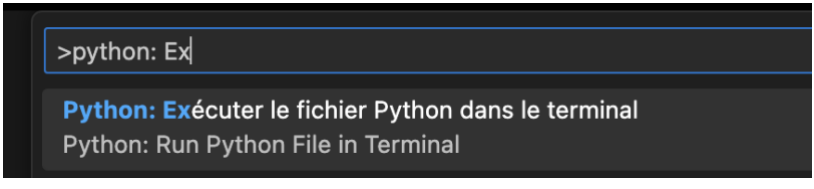
Dans un premier temps, il est suffisant d'employer les outils précédents pour l'apprentissage. Par la suite, il est d'usage d'employer des outils plus sophistiqués dont les environnements de développement intégrés (*Integrated Development Environment* - IDE) qui simplifient la tâche du programmeur en facilitant l'installation des logiciels, le processus d'édition, de compilation, d'exécution, de débogage, de suivi des versions et de partage des programmes Python.

Nous vous conseillons d'utiliser Visual Studio Code comme point de départ. Pour rendre l'opération aisée, installez l'extension Python (de Microsoft) au sein de Visual Studio Code. Choisissez dans le menu d'icônes de gauche l'icône « Extension » (composé généralement de 4 carrés). Une boîte de saisie devrait apparaître, tapez `Python`. Dans les premiers résultats, vous devriez trouver l'extension Python publiée par Microsoft. Après l'avoir sélectionnée, tapez `Installer (Install)`.

Nous allons maintenant exécuter un fichier. Auparavant, il faut créer le fichier. Dans le menu de Visual Studio Code, choisissez « Nouveau fichier texte » ou l'équivalent. Une fenêtre vous permettant de saisir du texte devrait apparaître. Tapez l'expression `print('Hello, World!')` puis choisissez « Enregistrer » dans le menu de Visual Studio Code. Choisissez un nom pour votre fichier se terminant en `.py` comme `monprogramme.py`.



Pour exécuter le nouveau programme, tapez F1 alors que vous êtes dans Visual Studio Code. Sur certains claviers, il peut être nécessaire de taper sur une touche function ou fn au même moment. Une boîte de saisie devrait alors apparaître, tapez « Python : Exécuter le fichier ». Sélectionnez avec votre souris la commande suggérée correspondant à votre saisie. Vous devriez alors voir votre programme s'exécuter dans la console.



Vous pouvez aussi utiliser la console et la commande `python monprogramme.py` pour exécuter votre programme.

1.2.4 Expressions simples avec un interprète Python

Le *shell* python peut être employé comme une simple calculatrice qui évalue une expression mathématique. Par exemple, si l'utilisateur entre

l'expression `3+2` suivie de `<fin de ligne>`, le système répond en affichant le résultat de l'addition des deux entiers qui est `5`.

```
>>> 3+2
5
```

L'affichage du résultat est automatique lorsqu'une expression est saisie.

La soustraction est exprimée par - :

```
>>> 5-4
1
```

La multiplication par * :

```
>>> 3*4
12
```

La division par / :

```
>>> 20/5
4.0
```

L'opérateur // calcule la division entière dont le résultat est un entier et le % calcule le reste (modulo) :

```
>>> 11//2
5
>>> 11%2
1
```

Exercice. Faites quelques expériences d'exécution d'expressions simples avec l'interface Python de votre choix.
https://www.python.org/shell/

1.2.5 Objet, classe, type et littéral

Dans les expressions précédentes, les données 1, 2, 5, 20 sont des nombres entiers. En Python, chacune des données est représentée par un ***objet***. Un objet a une ***classe***, une ***identité*** et une ***valeur***. L'objet occupe une place dans la mémoire de l'ordinateur qui sert à stocker sa valeur. Le classe de l'objet qui représente l'entier 20 est `int`. La classe de l'objet qui représente 4.0 est `float`. L'emploi du point décimal permet de déterminer que c'est un `float`. Le type du résultat de l'opération de division `/` est toujours un `float`.

La fonction `type()`[10] retourne la classe d'un objet :

[10] La notation `nom_fonction()` avec les parenthèses qui suivent le nom d'une fonction est une convention qui sert à désigner une fonction

```
>>> type(20)
<class 'int'>
>>> type(4.0)
<class 'float'>
```

Dans un premier temps, *type* et *classe* peuvent être considérés comme des synonymes. Des nuances seront introduites plus loin au sujet de la distinction entre ces concepts.

La fonction `id()` retourne l'identité d'un objet :

```
>>> id(20)
1509712640
```

L'identité d'un objet sert à lui faire référence et n'a pas de lien direct avec sa valeur. Dans notre exemple, la valeur de l'objet est le nombre entier 20 et l'identité de l'objet, qu'on peut assimiler à une adresse mémoire, est 1509712640. L'importance de la notion d'objet sera détaillée par la suite. Dans un premier temps, on peut simplement penser aux données en faisant abstraction des détails des mécanismes d'objets.

Le type d'un objet détermine sa **représentation interne** et les **traitements** qui lui sont applicables. Par exemple, une donnée de type `int` est représentée par un entier binaire de 32 bits (4 octets) si possible. Sinon, dans le cas où l'entier est trop gros, la représentation est changée automatiquement par Python pour accommoder une précision plus grande. Lorsqu'on utilise le type `int`, il n'est pas nécessaire de connaître ces détails d'implémentation. Les **opérations arithmétiques** (addition, soustraction, multiplication, division, etc.) sont permises sur les données de type `int`. Les fonctions et les méthodes sont d'autres formes d'opérations qui seront étudiées.

Une donnée de type `float` est représentée en virgule flottante qui est une représentation approximative d'un nombre réel. En effet, la représentation interne est composée d'une partie *mantisse* et d'une partie *exposant* qui sont d'une précision limitée. Cette limite de précision peut devenir contraignante pour des applications qui manipulent des nombres de très grande ou très petite taille relativement à la précision permise.

Le langage Python propose plusieurs types prédéfinies (*natifs*) avec un grand nombre de traitements associés. Il y a des types pour la manipulation des nombres entiers, réels, complexes, des Booléens, des chaînes de caractères, des chaînes d'octets, des listes, ensembles, tableaux, dictionnaires, etc.

Dans le code Python, une donnée particulière est représentée par une suite (*chaîne*) de caractères appelée un *littéral*. Par exemple, la chaîne des deux caractères « `20` » est un littéral qui représente l'entier 20. Lorsqu'un littéral est lu par l'interprète Python, il est converti sous forme d'un objet du type approprié. Python s'occupe de la représentation interne sous forme d'un entier binaire et de l'allocation de mémoire à l'objet. Le programmeur n'a pas à se soucier de ces détails dans la plupart des cas. Le tableau suivant montre quelques types de base et des exemples de littéraux.

Type	Description
bool	Deux valeurs : False ou True (équivalent à 0 ou 1 du type int)
int	Entier d'une précision illimitée (borné par la mémoire disponible). Exemples : 12, -46, 0o14 (octal), 0x31 (hexadécimal)
float	Nombre réel double précision (précision de 64 bits IEEE 754-1985) entre $-1.7*10^{308}$ et $1.7*10^{308}$ (15 chiffres significatifs). Exemples : 5., 5.0, 3.1416, 15.56e4 ou 15.56E4 (équivalent à 155600.0), 245.1e-5 (équivalent à 0.002451)
complex	Nombre complexe. Exemple : 5.23+67.241j
str	Chaîne de caractères. Suite de caractères Unicode entre apostrophes (peut inclure guillemets) ou guillemets (peut inclure apostrophes) ou triples apostrophes/guillemets (permet de chevaucher les lignes). Exemples : 'aB3$"5Fde63', ''XY'12 $%a'Bc'', '''peut chevaucher plusieurs lignes'''

En plus des classes prédéfinies, il est possible de créer de nouvelles classes et ceci est un aspect fondamental d'un langage de programmation objet tel que Python. Souvent, il n'est pas nécessaire de créer de nouvelles classes parce qu'il y a beaucoup de bibliothèques de classes qui ont déjà été développées pour divers types d'applications.

Bibliothèque (*Library*)

Dans le contexte informatique, une bibliothèque est un ensemble d'éléments logiciels (fonctions, classes, modules, …) destinés à être réutilisés pour développer de nouvelles applications.

Un aspect important de l'apprentissage d'un langage est de découvrir les bibliothèques existantes.

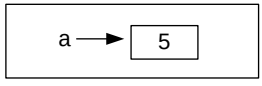
1.2.6 Notion de variable

Comme illustré dans le code suivant, il est possible d'affecter une valeur à une variable avec l'opérateur d'affectation `=` et d'employer le nom de

variable pour désigner cette valeur par la suite. La valeur à affecter est produite par une expression qui est à droite du = et la variable est à gauche.

```
>>> a=2+3
>>> a
5
>>> print(a)
5
```

Plus précisément, la variable fait référence à un objet qui représente la valeur. La référence est effectuée au moyen de l'identité de l'objet. Dans le diagramme suivant, la flèche représente une référence par l'identité d'objet. À un moment donné de l'exécution d'un programme, une variable ne fait référence qu'à un et un seul objet.



Il est à noter que le résultat de l'expression n'est pas affiché par le *shell* à la suite de l'exécution d'un énoncé d'affectation. D'autre part, si on tape une expression qui est simplement le nom d'une variable, le *shell* affiche la valeur correspondante, comme si on faisait appel à la fonction `print()`.

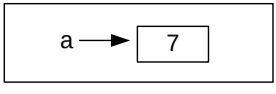
Après avoir affecté un objet à une variable, il est possible par la suite d'employer le nom de la variable pour faire référence à l'objet tel qu'illustré par l'expression suivante :

```
>>> 3*a
15
```

Étant donné que *a* fait référence à l'entier 5, l'expression devient 3*5.

Il est aussi possible de changer la valeur d'une variable :

```
>>> a=7
>>> a
7
```



Il est permis d'employer la même variable à gauche et à droite du « = ». Ceci ne ferait pas de sens en mathématiques où cette notation sert à définir une identité mathématique ou une équation à résoudre. En programmation, l'expression suivante est une affectation qui prend la valeur initiale de `a`, lui

additionne 1 et change la valeur de `a` pour la nouvelle valeur produite par l'expression :

```
>>> a=a+1
>>> a
8
```

La fonction `type()` appliquée à une variable retourne le type de l'objet auquel la variable fait référence.

```
>>> b = 5
>>> type(b)
<class 'int'>
>>> b = 3.4
>>> type(b)
<class 'float'>
```

Cet exemple illustre le fait qu'une variable Python peut faire référence à des objets de types différents au cours de l'exécution d'un programme.

Typage dynamique ou statique

En Python, une variable n'a pas de type comme tel. Elle peut faire référence à un objet de n'importe quel type à un point donné d'un script, et la variable peut être modifiée par la suite pour faire référence à un objet de n'importe quel autre type. Cette approche est parfois qualifiée de *typage dynamique*. D'autres langages (Java, C, etc.) emploient le typage *statique* où il faut spécifier explicitement le type d'une variable. Un avantage du typage statique est l'opportunité de vérifier et d'optimiser certains aspects du programme avant de l'exécuter. Le typage dynamique simplifie le code et permet plus de flexibilité à l'exécution.

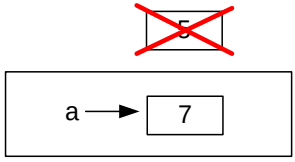
Depuis la version 3 de Python, diverses possibilités d'annotations de type ont été introduites. Ces mécanismes permettent de faire des vérifications pour obtenir du code plus robuste et valide. Des outils peuvent être employés pour vérifier la cohérence du code par rapport aux annotations de type. En revanche, l'exécution du code Python n'est pas affectée par les annotations de type. Ces aspects ne sont pas traités dans cet ouvrage. Voir :

`https://docs.python.org/fr/3/library/typing.html`

Ramasse-miettes (*garbage collector*)

Que devient l'objet qui correspond à 5 qui était référencé par *a* avant cette nouvelle affectation ? L'espace mémoire occupé par l'objet est récupéré

ultimement par un processus de ramasse-miettes (*garbage collection*) comme illustré par la figure suivante :



Ceci permet de réutiliser la mémoire par la suite afin de limiter l'encombrement mémoire du programme. Cet aspect devient important dans le cas d'un programme qui crée un grand nombre d'objets temporairement exploités et qui sont ne sont plus référencés par la suite. Le détail de la réalisation du ramasse-miettes est assez complexe et il est même possible de changer son comportement au besoin (https://docs.python.org/3/library/gc.html). Ceci peut devenir un enjeu important dans des applications avancées à haute consommation de mémoire.

Exercice. Reproduisez les opérations précédentes. Essayez quelques opérations arithmétiques et quelques affectations. Affichez le type de l'objet auquel la variable fait référence.

Les expressions suivent des règles d'évaluation semblables aux règles mathématiques de priorité et seront étudiées de manière plus détaillée au chapitre 4.

Exercice. Vérifiez le résultat de l'expression 2+3*4.

1.3 Références

Un grand nombre de ressources Web sont disponibles pour l'apprentissage de Python. En particulier, le site suivant maintient plusieurs liens vers des ressources pour débutants :

```
https://wiki.python.org/moin/BeginnersGuide
```

Plusieurs publications sur divers aspects de Python sont disponibles sur le site :

```
https://www.freecodecamp.org/news/tag/python/
```

2 Principes de base de la programmation Python

Ce chapitre introduit les principes de base de la programmation avec Python à partir d'un exemple simple. Le script Python suivant permet de saisir deux nombres entiers par le clavier de l'ordinateur et d'en afficher la somme.

CodePython/chapitre2/Exemple1.py[11]

```
1. ''' Exemple1.py
 2. Ce programme saisit deux entiers et en affiche la somme '''
 3.
 4. # Saisir les deux chaînes de caractères qui représentent des
nombres entiers
 5. chaine1 = input("Entrez un premier nombre entier :")
 6. chaine2 = input("Entrez un second nombre entier :")
 7.
 8. # Convertir les deux chaînes de caractères en entiers
 9. entier1 = int(chaine1)
10. entier2 = int(chaine2)
11.
12. # Calculer la somme des deux entiers
13. somme = entier1 + entier2
14.
15. # Afficher la somme
16. print("La somme des deux entiers est ",somme)
17.
```

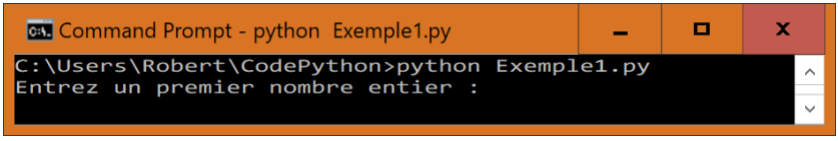
Exercice. Editez ce script, sauvegardez-le dans un fichier nommé `Exemple1.py` et faites-le exécuter.

Le scénario suivant montre un exemple d'exécution du script. Le script est invoqué par la commande :

```
python Exemple1.py
```

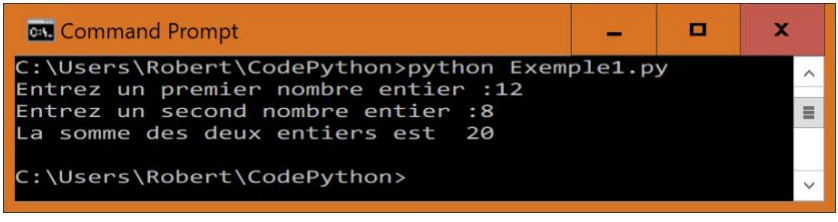
Ceci provoque l'exécution de ce qui est communément appelé le *programme principal* (*main program*) du script. Le script affiche d'abord le message qui invite l'utilisateur à entrer un premier nombre entier.

[11] Ce titre est un lien vers le répertoire `GitHub` qui contient le code des exemples et des exercices. Le préfixe `CodePython` correspond à `https://github.com/RobertGodin/CodePython`



L'utilisateur entre `12` suivi de `<fin de ligne>`. Le script affiche ensuite un message qui invite l'utilisateur à entrer un second entier.

L'utilisateur entre `8` suivi de `<fin de ligne>`. Le script affiche ensuite le résultat : `20`. Le script se termine et l'invite de commande Windows est affiché.



Les prochaines sections examinent le programme en détails.

2.1 Commentaire Python

Le script `Exemple1.py` débute par un commentaire :

```
''' Exemple1.py
Ce programme saisit deux entiers et en affiche la somme '''
```

Toute portion du code source qui débute par trois apostrophes (`'''`) ou trois guillemets (`"""`) et se termine de la même manière est considérée comme un commentaire en Python et n'a aucun effet du point de vue de l'exécution du programme. Ainsi, tous les commentaires peuvent être supprimés sans changer le fonctionnement du programme. L'objectif principal d'un commentaire est de faciliter la compréhension du

programme par les humains (programmeurs)[12]. Un commentaire de ce type peut s'étendre sur une ou plusieurs lignes.

Une autre manière de faire un commentaire consiste à le débuter par # comme dans :

```
# Saisir les deux chaînes de caractères qui représentent des nombres entiers
```

Le texte qui suit le # est considéré comme un commentaire. Un tel commentaire se termine automatiquement à la fin de la ligne courante et ne peut donc pas chevaucher plusieurs lignes. Pour l'affichage d'une longue ligne, dans ce manuel ou au sein d'un éditeur de texte, la ligne est souvent visuellement répartie sur plusieurs lignes afin de respecter les marges de la page ou une limite maximale de caractères par ligne. Cet effet est souvent appelé un retour automatique à la ligne. Le retour à la ligne automatique est un effet visuel qui n'affecte pas le contenu du code informatique et qui ne fait pas partie de la syntaxe du programme. Le langage Python ne se préoccupe que des lignes véritables créées par le programmeur à l'aide d'un retour de charriot.

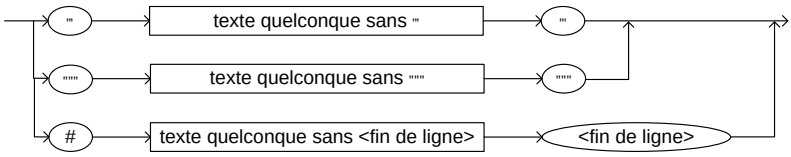
Diagramme syntaxique

Un diagramme syntaxique est un graphe qui permet de représenter les règles de syntaxe d'un langage de programmation. Le sens des flèches indique comment enchaîner les différents éléments. Un ovale contient des caractères spécifiques. Un rectangle fait référence à une autre règle de syntaxe.

Le *diagramme syntaxique* suivant montre les trois alternatives de commentaire Python. Le rectangle contenant le titre `texte quelconque sans '''` représente une séquence de caractères quelconque qui ne peut contenir la suite de trois apostrophes `'''`. Une règle représentée par un rectangle peut être détaillée dans un autre diagramme syntaxique. Dans cet exemple, les règles détaillées sont omises.

[12] Le module doctest (https://docs.python.org/3/library/doctest.html) exploite aussi les commentaires pour les tests

commentaire :



2.2 Appel de fonction

La ligne suivante représente un appel (*invocation*) de la fonction `input()` et l'affectation du résultat de la fonction à la variable nommée `chaine1` :

```
chaine1 = input("Entrez un premier nombre entier :")
```

Nom de variable

Les noms de variables doivent respecter des règles précises qui s'appliquent à tous les identificateurs Python. Un identificateur doit débuter par une lettre ou un soulignement _ suivi d'une suite d'un nombre quelconque de caractères limités aux lettres, chiffres, et au soulignement _. Les lettres peuvent être majuscules ou minuscules et la casse est significative. Par exemple, « somme » et « Somme » sont deux noms différents. Certains identificateurs sont dits réservés et ne peuvent servir de nom de variable. Les identificateurs réservés ont un sens prédéfini en Python (voir `https://docs.python.org/3/reference/lexical_analysis.html#keywords`).

Il est important de choisir les noms de manière à faciliter la compréhension des programmes. Le nom d'une variable devrait donner une idée du rôle de la variable dans le programme. Au-delà des règles de syntaxe obligatoires, des conventions de style sont décrites le guide de style suivant[13] : https://www.python.org/dev/peps/pep-0008/

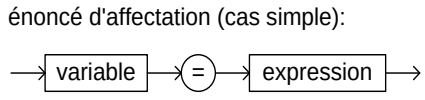
En particulier, pour le nom des variables, il est proposé de séparer les mots par un soulignement _ tel que dans :

```
un_nom_de_variable
```

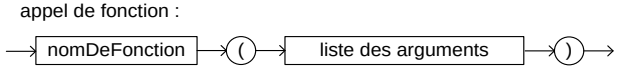
[13] Ce guide est un PEP (Python Enhancement Proposal). Les PEPs sont des propositions pour l'évolution du langage faite à la communauté qui permettent de gérer l'évolution de manière consensuelle.

Le nom de variable formé d'un soulignement _ seul peut servir à désigner un nom de variable dont la valeur est sans importance mais qui doit être présent dans l'énoncé.

Tel qu'expliqué au chapitre précédent, la forme générale d'une affectation est :



L'expression dans notre exemple est un simple appel à la fonction `input()`. L'appel à une fonction a la forme générale suivante :



Dans l'appel d'une fonction, il faut préciser les valeurs des arguments (aussi appelés *paramètres réels*) entre parenthèses après le nom de la fonction. Un argument représente une valeur qui est utilisée par la fonction.

Par opposition aux mathématiques, un argument de fonction n'est pas limité à une valeur numérique. Le texte

```
"Entrez un premier nombre entier"
```

est la valeur de l'argument de la fonction `input()` dans notre exemple. Il représente un message qui est affiché par la fonction sur l'unité de sortie.

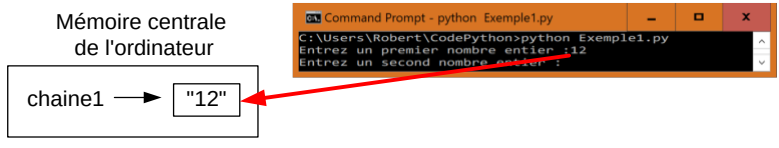
Chaîne de caractère : type `str`

En Python, une séquence de caractères entre apostrophes (`'`) ou guillemets (`"`) est interprétée comme une chaîne de caractères, c'est-à-dire un littéral de type `str`. Comme pour un commentaire, les apostrophes triples (`'''`) ou les guillemets triples (`"""`) doivent être utilisés si la chaîne de caractères chevauche plusieurs lignes.

La fonction `input()` affiche d'abord l'argument `"Entrez un premier nombre entier :"` sur l'unité de sortie et attend qu'une suite de caractères suivie de `<fin de ligne>` soit entrée avec le clavier de l'ordinateur. La suite de caractères saisie excluant le `<fin de ligne>` est retournée par la fonction sous forme d'un objet de type `str`. Comme expliqué plus loin, une fonction peut ou non retourner une valeur. Pour apparaître dans une

expression de la partie droite d'une affectation, il faut que la fonction retourne une valeur. Dans notre exemple, la valeur retournée par la fonction est affectée à la variable `chaine1`.

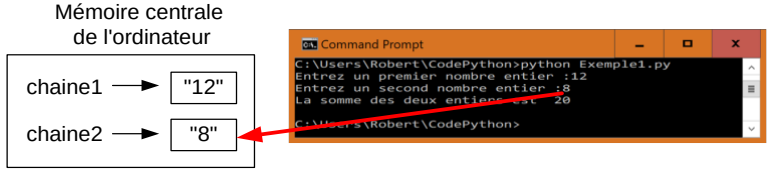
Insistons sur le fait que la séquence de caractères lue n'est pas interprétée comme un nombre entier à ce point-ci mais comme une simple chaîne de caractères. Il faut par la suite convertir cette chaîne en un entier. La figure suivante illustre l'effet du code dans le contexte de ce scénario. Le nom de variable `chaine1` est créé, et la fonction `input()` crée un objet qui prend comme valeur la chaîne de caractère `"12"`. Dans la figure, la boîte représente l'objet en mémoire. Le nom de variable `chaine1` fait référence à l'objet qui représente la chaîne de caractère `"12"`. La flèche représente la référence à l'objet par la variable. Concrètement, la référence correspond à l'identité de l'objet. L'interprète Python doit donc mémoriser pour chacune des variables, la valeur de l'identité d'objet de la donnée à laquelle la variable fait référence. La *table des noms* représente cette correspondance que l'interprète Python doit maintenir. Mais ceci est invisible au programmeur qui n'a pas à se préoccuper de la valeur de l'identité de l'objet. Ainsi, le nom de variable `chaine1` devient en quelque sorte un alias pour la chaîne de caractère `"12"` jusqu'à ce que la variable soit modifiée.



Ensuite, la ligne suivante saisie une seconde chaine de caractère :

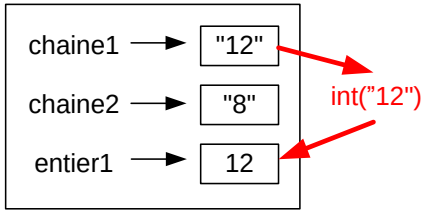
```
chaine2 = input("Entrez un second nombre entier :")
```

Voici l'effet avec notre scénario :



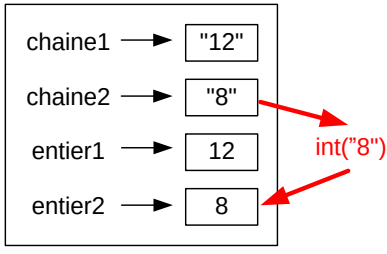
La ligne suivante convertie `chaine1` en un entier `entier1` par l'appel à la fonction `int()` qui retourne un objet représentant l'entier correspondant.

```
entier1 = int(chaine1)
```



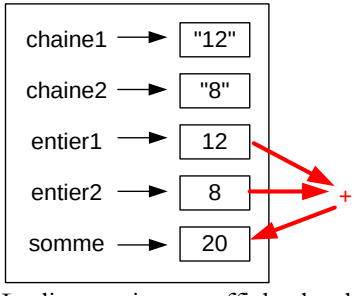
La ligne suivante convertie `chaine2` en un entier `entier2` par l'appel à la fonction `int()`.

```
entier2 = int(chaine2)
```



La ligne suivante affecte la somme des deux entiers `intier1` et `entier2` à la variable `somme`.

```
somme = entier1 + entier2
```



La ligne suivante affiche la chaîne de caractère `La somme des deux entiers est` suivie de la valeur de `somme` par l'appel à la fonction `print()`.

```
print("La somme des deux entiers est ",somme)
```

Un outil intéressant pour l'apprentissage de Python est Python Tutor (https://pythontutor.com/python-compiler.html#mode=edit) qui permet d'éditer du code Python et de visualiser l'effet du code. Cet outil peut être employé pour aider à la compréhension du code présenté tout au long de cet ouvrage.

Exercice. Éditez le code de l'exemple précédent avec Python Tutor et visualisez l'exécution du code ligne par ligne en cliquant sur le bouton « *Vizualize Execution* » pour déclencher la visualisation et en cliquant sur le bouton « *Next* » à répétition pour exécuter le code ligne par ligne.

Cet exemple montre qu'un énoncé du langage peut être un simple appel à une fonction. Le langage Python offre un grand nombre de fonctions prédéfinies. Plusieurs de ces fonctions seront présentées tout au long de cet ouvrage. La liste des fonctions prédéfinies est énumérée à :

https://docs.python.org/3/library/functions.html#built-in-functions

La fonction `help()` de Python permet d'obtenir de l'aide interactive en affichant la documentation au sujet d'une fonction ainsi que pour d'autres éléments du langage.

Exemple.

Le programmeur peut aussi créer une fonction. Cet aspect important de la programmation est présenté plus loin.

2.3 Exceptions

Un programme Python peut lever une exception si une condition anormale se présente :

```
C:\Users\Robert\CodePython>python Exemple1.py
Entrez un premier nombre entier :1
Entrez un second nombre entier :e
Traceback (most recent call last):
  File "Exemple1.py", line 10, in <module>
    entier2 = int(chaine2)
ValueError: invalid literal for int() with base 10: 'e'
```

Dans l'exemple, l'utilisateur a entré la lettre `"e"` par erreur pour le second nombre entier. Lorsque la fonction `int()` est appelée avec l'argument `"e"`, une exception est levée par la fonction parce qu'elle a détecté que l'argument n'est pas une chaîne de caractère qui représente un nombre entier. L'exception provoque un arrêt du programme et un message

explicatif est affiché. Le message indique la ligne de code qui a provoqué l'exception. Il précise aussi le type d'exception, ici `ValueError` suivi d'une explication plus détaillée du problème.

Dans le processus de développement d'un programme, il est fréquent de rencontrer des exceptions. Le message aide le programmeur à comprendre la nature du problème pour y trouver une solution. Le chapitre 9 explique en détails les exceptions et la manière de les traiter dans un programme.

2.4 Génie logiciel et spécification du logiciel

Le développement de logiciel est un sujet très vaste et le *génie logiciel* (*software engineering*) étudie les méthodes et processus du développement de logiciel. Lorsqu'on développe un programme, il est important de préciser ce que le programme doit faire, sous forme d'une *spécification du logiciel* (*software specification*). Dans le cas de gros logiciels, la spécification peut devenir très complexe et des approches systématiques ont été développées à cet effet. Dans les exemples et exercices simples étudiés ici, la spécification se limite à un court texte.

Exercice. Modifiez le programme `Exemple1` afin qu'il lise trois entiers (plutôt que deux) et en affiche la somme.

Solution. La solution suivante est une adaptation directe de `Exemple1.py`. La nouvelle variable `chaine3` est introduite pour y lire la troisième chaîne de caractère ainsi qu'une variable `entier3` pour l'entier correspondant.

CodePython/chapitre2/Exercice1.py

```
''' Exercice1.py
Ce programme saisit trois entiers et en affiche la somme '''

# Saisir les deux chaînes de caractères qui représentent des nombres entiers
chaine1 = input("Entrez un premier nombre entier :")
chaine2 = input("Entrez un second nombre entier :")
chaine3 = input("Entrez un troisième nombre entier :")


# Convertir les trois chaînes de caractères en entiers
entier1 = int(chaine1)
entier2 = int(chaine2)
entier3 = int(chaine3)

# Calculer la somme des trois entiers
somme = entier1 + entier2 + entier3

```

```
18. # Afficher la somme
19. print("La somme des trois entiers est ",somme)
```

Exercice. Écrivez un programme qui effectue le même traitement que le précédent mais en utilisant une seule variable de type `str` (plutôt que trois) et une variable `int` plutôt que quatre !

Solution. La solution suivante réutilise la même variable `chaine` pour lire chacune des chaînes de caractère qui représente un entier et la variable `somme` pour cumuler les valeurs intermédiaires de la somme.

CodePython/chapitre2/Exercice2.py

```
 1. ''' Exercice2.py
 2. Ce programme saisit trois entiers et en affiche la somme en
employant une seule
 3. variable de type str et une de type int'''
 4.
 5. chaine = input("Entrez un premier nombre entier :")
 6. somme = int(chaine)
 7.
 8. chaine = input("Entrez un second nombre entier :")
 9. somme = somme + int(chaine)
10.
11. chaine = input("Entrez un troisième nombre entier :")
12. somme = somme + int(chaine)
13.
14. # Afficher la somme
15. print("La somme des trois entiers est ",somme)
16.
```

Qualité du logiciel : mémoire consommée, temps de calcul, clarté du programme

Lorsqu'on développe un programme, il est important d'en garantir la qualité. La qualité du logiciel comporte plusieurs facettes qui sont abordées tout au long de cet ouvrage. Un aspect à considérer dans le développement d'un programme est la quantité de mémoire consommée. La solution *Exercice2* est préférable à *Exercice1* de ce point de vue. Cependant, ceci n'est pas le seul aspect à considérer. Parfois, le fait de minimiser la mémoire consommée de manière absolue peut introduire d'autres défauts tel qu'un temps de traitement plus élevé ou un manque de clarté d'un point de vue de la lisibilité du programme. Il faut donc tenter de considérer tous ces facteurs. Parfois, il faut faire des compromis entre les différentes facettes de la qualité du logiciel.

2.5 Disposition du code Python

Les règles de Python concernant la disposition du code sont assez flexibles au sens où les éléments du langage peuvent être séparés par une suite quelconque d'espaces blancs. Le terme *espace blanc* désigne un *espace*, une marque de *tabulation* ou une *fin de ligne*. Cependant, certaines instructions doivent respecter des règles précises concernant l'indentation tel qu'expliqué au prochain chapitre.

Énoncé multiligne dans le shell

Dans le contexte du shell Python, si une expression n'est pas complète et que l'utilisateur tape `<fin de ligne>`, le système répond par « … » pour indiquer qu'il attend le reste de l'énoncé :

```
>>> print(
... 'abc'
... )
abc
>>>
```

Qualité du logiciel : clarté, lisibilité du code

Un aspect important de la qualité d'un programme est sa lisibilité. Les règles de Python concernant l'indentation favorisent une meilleure lisibilité du code. Il est important de considérer le fait que le code doit souvent être lu et relu par un ou plusieurs programmeurs afin de le corriger, le modifier ou le réutiliser. Cet aspect peut souvent sembler peu important pour un novice dans le contexte de petits programmes à utilisation restreinte mais il devient très important dans le développement de logiciels de plus grande envergure partagés à grande échelle.

Pour aider le lecteur, n'hésitez pas à mettre des commentaires et à faire un choix judicieux concernant le nom de vos variables et de vos fonctions.

3 Structures de contrôle

Il y a trois manières fondamentales d'enchaîner les opérations d'un programme : la *séquence*, la *répétition* et l'*alternative* (*choix*, *décision*). Ce chapitre présente les énoncés de base Python qui permettent d'exprimer ces trois types d'enchaînement.

3.1 La séquence

Les exemples vus jusqu'à présent sont des séquences. Une séquence d'énoncés est de la forme générale suivante :

```
énoncé1
énoncé2
…
énoncén
```

Chacun des énoncés débute au premier caractère de la ligne s'il n'est pas dans un bloc comme nous le verrons plus loin. Il se termine normalement par la `<fin de ligne>`.

Si une ligne se termine par un \, elle se poursuit sur la ligne suivante. Un énoncé peut ainsi chevaucher plusieurs lignes au besoin :

```
Partie 1 d'un énoncé \
Partie 2 énoncé \
…
Partie n énoncé
```

Les énoncés placés en séquence les uns après les autres et sont exécutés dans cet ordre.

Diagramme UML d'une séquence

La Figure 1 montre une représentation graphique d'une séquence avec un *diagramme d'activité* du Langage de Modélisation Unifié (*Unified Modeling Language* - UML). Un rectangle aux coins arrondis représente une activité. Dans notre cas, une activité correspond à un énoncé Python. Les flèches indiquent l'ordre d'exécution des activités. Le point noir représente le début et le point noir encerclé la fin de l'exécution. UML est un langage normalisé pour la représentation de logiciels. Le diagramme d'activité permet de visualiser l'enchaînement des énoncés.

Les représentations graphiques sont des outils fréquemment employés dans le développement de logiciels. La visualisation permet d'améliorer la compréhension d'éléments complexes de diverses natures. Le

développement peut souvent être démarré par la production de divers artefacts incluant des diagrammes. Le langage UML inclut plusieurs types de diagrammes à cet effet.

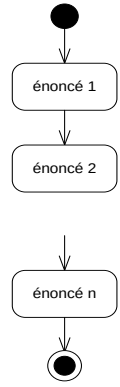


Figure 6. Diagramme d'activité UML pour une séquence

Si les énoncés d'une séquence sont courts, il est aussi possible d'en placer plusieurs sur la même ligne en les séparant par des `;`.

```
énoncé1; énoncé2; … ; énoncén
```

L'exemple suivant reprend le programme vu précédemment en regroupant trois des énoncés sur une seule ligne.

CodePython/chapitre3/ExempleSequenceSurUneLigne.py

```
 1. # Saisir les deux chaînes de caractères qui représentent des
nombres entiers
 2. chaine1 = input("Entrez un premier nombre entier :")
 3. chaine2 = input("Entrez un second nombre entier :")
 4.
 5. # Convertir les deux chaînes de caractères en entiers et
calculer la somme
 6. entier1 = int(chaine1); entier2 = int(chaine2); somme = entier1
+ entier2
 7.
 8. # Afficher la somme
 9. print("La somme des deux entiers est ",somme)
```

Dans le cas d'une séquence d'affectations, il est possible d'employer une *affectation multiple* selon la syntaxe suivante :

```
variable1, variable2, … , variablen = expression1, expression2, … , expressionn
```

CodePython/chapitre3/ExempleAffectationMultiple.py

```
1. # Saisir les deux chaînes de caractères qui représentent des nombres entiers
2. chaine1 = input("Entrez un premier nombre entier :")
3. chaine2 = input("Entrez un second nombre entier :")
4.
5. # Convertir les deux chaînes de caractères en entiers et calculer la somme
6. entier1, entier2, somme = int(chaine1), int(chaine2), entier1 + entier2
7.
8. # Afficher la somme
9. print("La somme des deux entiers est ",somme)
```

3.2 La répétition avec l'énoncé while

Imaginons que l'on veuille afficher les entiers de 1 à 5. Le programme suivant peut produire ce résultat :

CodePython/chapitre3/ExempleAfficher1a5.py

```
1. """
2. Afficher les entiers de 1 à 5
3. """
4.
5. print(1)
6. print(2)
7. print(3)
8. print(4)
9. print(5)
10.
```

S'il fallait afficher les entiers de 1 à 1 000 000 de cette manière, le programme serait long à écrire … Pour éviter de répéter les énoncés dans le programme, on peut employer une répétition (aussi appelée *boucle* ou *itération*). L'énoncé `while` Python est un des énoncés qui permet d'effectuer une répétition. `ExempleWhile` illustre la notion de répétition avec compteur en employant un énoncé `while`. Ce programme a le même effet que le précédent.

CodePython/chapitre3/ExempleWhile.py

```
1. """
2. Exemple d'une boucle while avec un compteur
3. """
4.
5. compteur = 1
6. while compteur <=5:
```

```
7.      print(compteur)
8.      compteur = compteur + 1
9.
```

Voici la syntaxe d'un énoncé *while* :

L'expression entre parenthèses doit être une *expression booléenne*, aussi appelée *condition*, dont la valeur est de type `bool` (*vrai* (`True`) ou *faux* (`False`)). Si cette condition est respectée (i.e. la valeur retournée par l'*expression* est `True`), le bloc après le `while` est répété en boucle jusqu'à ce que la condition ne soit plus respectée (i.e. la valeur retournée par l'expression est `False`). Il est donc important que le code à l'intérieur de la boucle rende la condition *False* après un certain nombre d'itérations, sinon, la boucle continue de se répéter sans fin (*boucle infinie*).

Indentation d'un bloc de code Python

Un *bloc de code* Python est une séquence d'un ou plusieurs énoncés indentés de manière identique par rapport à une entête qui se termine par « : ». Un entête peut-être un `while` ou d'autres types d'énoncés. La convention suggérée est d'indenter systématiquement avec 4 espaces mais ce n'est pas obligatoire. Il est aussi permis d'indenter avec les tabulations. Cependant, il ne faut pas mélanger les deux. Le guide de style PEP8 recommande d'employer systématiquement 4 espaces pour l'indentation. Ceci peut être facilité en paramétrant l'éditeur de code à cet effet.

Il est important de comprendre que la syntaxe d'un `while` permet d'imbriquer un autre `while` dans le bloc et ceci à volonté en fonction des besoins. La figure suivante illustre l'enchaînement des énoncés du programme par un diagramme d'activité UML. Un losange représente une condition. Les flèches qui partent de la condition montrent les deux enchaînements possibles selon le résultat de la condition. Le diagramme montre bien le concept de répétition par un cycle dans l'enchaînement des énoncés.

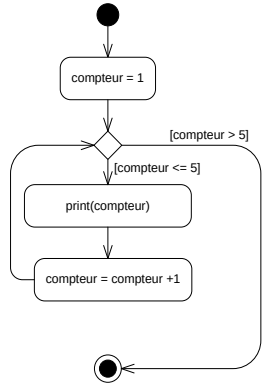


Figure 7. Diagramme d'activité UML pour le programme.

S'il y a plus d'un énoncé à répéter, comme c'est le cas de notre exemple, il faut les regrouper dans un bloc, et les indenter de manière identique.

Dans notre exemple, la condition est :

```
compteur <= 5
```

Ainsi, tant que cette condition s'avère vraie (tant que le compteur est plus petit ou égal à 5), les énoncés indentés suivants sont exécutés en boucle :

```
    print(compteur)
    compteur = compteur + 1
```

Dès que le compteur dépasse la valeur de cinq, le programme passe à l'énoncé suivant la répétition, qui est la fin du programme dans notre exemple.

Incrémentation / décrémentation avec += et -=

L'incrémentation du compteur peut s'exprimer de manière équivalente avec la syntaxe : `compteur += 1`

Le `-=` sert à la décrémentation. En fait, il est souvent mal vu d'utiliser l'expression `compteur = compteur + 1` dans un contexte professionnel.

Exercice. Faites exécuter le code précédent avec Python Tutor.

Exercice. Vérifiez ce qui se passe si la ligne d'incrémentation du compteur est omise ! NB Vous pouvez interrompre un script avec le caractère d'interruption, `<ctrl-C>`.

3.3 Expression de comparaison

Le nombre de répétition du `while` est contrôlé par une expression booléenne qui retourne `True` ou `False`. Une expression booléenne peut être formée en comparant des valeurs à l'aide des opérateurs de comparaison du tableau suivant. Il y a quelques différences avec la notation mathématique usuelle pour accommoder les ensembles de caractères des claviers simples.

Opérateur de comparaison	Signification
<	Plus petit que
<=	Plus petit ou égal à
>	Plus grand que
>=	Plus grand ou égal à
==	Égal à
!=	N'est pas égal à

Il est possible de former des expressions booléennes plus complexes avec les opérateurs booléens (voir chapitre 4). Le test d'égalité entre deux valeurs est exprimé par `==` pour marquer la distinction avec un énoncé d'affection exprimé par `=`.

Exercice. Modifiez l'exemple précédent afin qu'il affiche les entiers 0, 2, 4, 6, 8, 10.

Solution. CodePython/chapitre3/ExerciceWhile1.py

```
1. """
2. Afficher 0,2,4,6,8,10
3. """
4.
5. compteur = 0
6. while compteur <=10:
7.     print(compteur)
8. compteur = compteur + 2
```

Exercice. Modifiez l'exemple afin d'afficher 5, 4, 3, 2, 1, 0, -1, -2, -3, -4, -5.

Solution. CodePython/chapitre3/ExerciceWhile2.py

```
1. """
2. Afficher les valeurs de compteur 5,4,3,2,1,0,-1,-2,-3,-4,-5
3. """
4.
```

```
5. compteur = 5
6. while compteur >= -5:
7.     print(compteur)
8.     compteur = compteur -1
9.
```

Exercice. Reprenons l'exemple de lecture d'entiers afin d'en afficher la somme. Nous avons vu le cas de deux et de trois entiers. Dans cet exercice, il faut additionner 10 entiers ! Une solution consiste à répéter dix fois la lecture, la conversion et l'accumulation dans `somme`. Dans cet exercice, il faut plutôt utiliser une boucle `while` pour lire dix entiers et en afficher la somme.

Solution. CodePython/chapitre3/ExerciceWhile3.py

```
 1. """
 2. Lire dix entiers et en afficher la somme avec un while
 3. """
 4.
 5. somme=0
 6. compteur = 1
 7. while compteur <= 10:
 8.     chaine = input("Entrez un nombre entier :")
 9.     somme = somme + int(chaine)
10.     compteur = compteur+1
11. print("La somme des dix entiers est:",somme)
12.
```

Exercice. Maintenant supposons que le nombre d'entiers à lire est inconnu à l'avance. Une technique souvent employée pour arrêter la répétition est l'utilisation d'une valeur spéciale appelée *sentinelle* qui provoque l'arrêt de la répétition. Par exemple, supposons que le nombre 0 représente la sentinelle. Vous devez écrire un programme qui lit une série d'entiers jusqu'à ce que l'entier 0 soit entré et qui produit la somme de ces entiers.

Solution. CodePython/chapitre3/ExerciceWhileSentinelle.py

```
 1. """
 2. Lire une suite d'entiers jusqu'à ce que l'entier 0 soit entré et
afficher la somme des entiers lus.
 3. """
 4.
 5. somme=0
 6. chaine = input("Entrez un nombre entier, 0 pour terminer :")
 7. while chaine != '0':
 8.     somme = somme + int(chaine)
 9.     chaine = input("Entrez un nombre entier, 0 pour terminer :")
10. print("La somme des entiers est:",somme)
```

Une solution plus compacte et élégante est possible avec l'opérateur morse (*walrus*) `:=` introduit avec Python 3.8 qui permet une affectation à l'intérieur

d'une expression. Ceci permet de tester une condition de fin de boucle à partir d'une expression et de réutiliser par la suite la valeur de l'expression sans devoir la réévaluer.

Exemple. CodePython/chapitre3/ExempleOperateurMorse.py

```
"""
Lire une suite d'entiers jusqu'à ce que l'entier 0 soit entré et afficher la somme des entiers lus.
Solution avec l'opérateur morse :=
"""

somme=0
while (chaine := input("Entrez un nombre entier, 0 pour terminer :")) != '0':
    somme = somme + int(chaine)
print("La somme des entiers est:", somme)

```

Exercice. Afficher le résultat suivant :

```
1
12
123
1234
12345
123456
1234567
12345678
123456789
```

Indice : il est permis d'imbriquer une boucle `while` dans une autre boucle `while`.

NB Il est possible d'afficher la variable `une_donnee` sans faire un saut de ligne par la forme suivante de la fonction `print()` :

```
print(une_donnee, end='')
```

Solution. CodePython/chapitre3/ExerciceWhileImbriques.py

```
"""
Exercice deux while imbriqués
"""
compteur1 = 1
while compteur1 <= 9:
    compteur2 = 1
    while compteur2  <= compteur1:
        print(compteur2, end='')
        compteur2 = compteur2 + 1
    print('')
    compteur1 = compteur1 + 1
```

3.4 La répétition avec l'énoncé for

L'utilisation d'une répétition avec compteur est très fréquente. La boucle `for` simplifie l'écriture de telles boucles. Le programme suivant produit le même effet que `ExempleWhile` en affichant les entiers de 1 à 5. Dans un énoncé `for`, l'initialisation du compteur, l'expression de fin de répétition et la mise-à-jour du compteur sont remplacés par l'itération sur une séquence générée par la fonction `range()`.

CodePython/chapitre3/ExempleForRange.py

```
1. """
2. Exemple d'une boucle for avec la fonction range()
3. """
4.
5. for compteur in range(1,6):
6.     print(compteur)
```

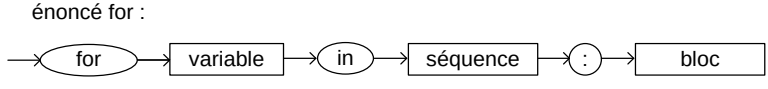
Après le mot-clé `for`, il faut spécifier une variable, ici `compteur`, qui prendra successivement les valeurs d'une séquence spécifiée après le mot-clé `in`. L'appel à `range(1,6)` génère la séquence 1, 2, 3, 4, 5. Le premier argument est la valeur initiale. La valeur du deuxième argument est la borne supérieure mais elle n'est pas incluse. L'incrément est 1 par défaut.

L'appel à `range(5)` génère la séquence 0,1,2,3,4. Par défaut, la valeur initiale est 0. Et `range(4,16,3)` génère 4,7,10,13 (le 16 est exclus). L'incrément peut être négatif comme dans `range(5,-10,-4)` qui génère 5,1,-3,-7. Comme pour un incrément positif, la dernière valeur est exclue : ainsi `range(8,4,-2)` génère la séquence 8,6. Le 4 est exclu.

Attention !

La fonction `range()` ne produit pas la liste des valeurs de la séquence en mémoire. Les valeurs sont générées à la demande, pendant le processus d'itération, par un objet itérateur associé à la fonction. Les détails de ce mécanisme d'itérateur sont expliqués à la section 7.5. De plus la fonction `range()` de la version 3 de Python remplace la fonction `xrange()` de la version 2. Et la fonction `range()` de la version 2 qui génère en mémoire une liste de valeurs n'est plus supportée.

La syntaxe du `for` est :



Dans l'exemple précédent, la séquence est produite par la fonction `range()`. D'autres possibilités sont explorées plus loin.

Exercice. Affichez la séquence des entiers 0, 2, 4, 6, 8, 10 avec un `for`.

Exercice. Affichez 5, 4, 3, 2, 1, 0 avec un `for`.

Exercice. Affichez 5, 4, 3, 2, 1, 0, -1, -2, -3, -4, -5 avec un `for`.

Exercice. Affichez 4, 2, 0, -2, -4, -6 avec un `for`.

Exercice. Utilisez un `for` pour lire dix entiers et en afficher la somme.

Exercice. Afficher le résultat suivant avec un `for` imbriqué dans un autre:

```
1
12
123
1234
12345
123456
1234567
12345678
123456789
```

Solution. CodePython/chapitre3/ExerciceForImbriques.py

```
1. """
2. Exercice for imbriqué
3. """
4. for compteur1 in range(1,10):
5.     for compteur2 in range(1,compteur1+1):
6.         print(compteur2, end='')
7.     print('')
```

3.5 L'alternative (choix, décision) avec if

L'énoncé `if` permet au programme de prendre une décision au sujet des actions à exécuter en fonction d'une condition à évaluer. Le programme suivant lit un entier (`un_int`) et répond en affichant un message qui indique s'il est plus grand que 10 ou non. Pour déterminer le message à afficher, une décision est prise en comparant l'entier lu à 10 dans une condition (`un_int > 10`). Selon le résultat de la condition, le `if` permet de choisir entre les deux blocs alternatifs à exécuter. La première alternative suit le `:` après la condition et elle est exécutée si la condition est évaluée à vrai (`True`). Sinon, i.e. la valeur de l'expression est `False`, le bloc qui suit `else :` est exécuté.

CodePython/chapitre3/ExempleIf.py

```
"""
Exemple simple illustrant l'énoncé if
"""

un_int = int(input("entrez un nombre entier: "))
if un_int > 10:
    print(un_int,"est plus grand que 10")
else :
    print(un_int,"est plus petit ou égal à 10")
```

Dans notre exemple, si la condition

```
un_int > 10
```

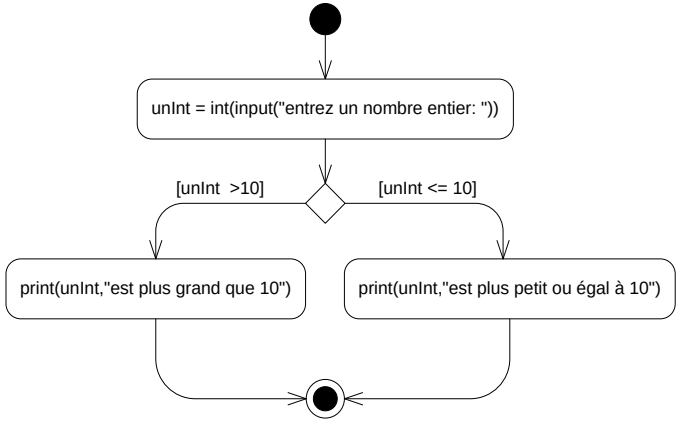
se révèle vraie, l'énoncé

```
    print(un_int,"est plus grand que 10")
```

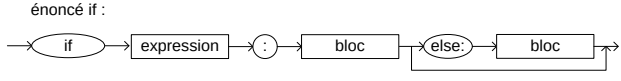
sera exécuté. Si l'expression se révèle fausse, l'énoncé qui sera exécuté est :

```
    print(un_int,"est plus petit ou égal à 10")
```

Le diagramme d'activité suivant illustre cet enchaînement.

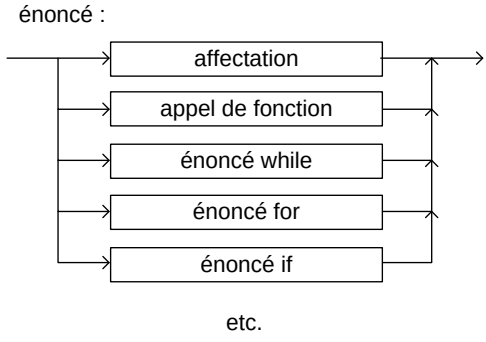


La syntaxe du `if` est :



La partie `else` est optionnelle. En son absence, lorsque l'expression de condition est fausse, rien n'est exécuté et l'interprète passe à l'énoncé suivant le `if`. Il faut faire attention de bien indenter le `else` afin de produire l'effet désiré.

Les trois manières d'enchaîner les énoncés ont été introduites : séquence, répétition et alternative. Le diagramme syntaxique suivant résume les différents cas d'énoncés vus jusqu'à présent :



Il faut se rappeler que dans les structures de contrôle, il est permis d'imbriquer d'autres structures dans leurs blocs. On obtient donc la possibilité d'imbriquer autant de structures que nécessaires.

L'exemple suivant illustre un cas de `if` sans `else` et d'un `if` imbriqué.

CodePython/chapitre3/ExempleIfImbrique.py

```
"""
Exemple if sans else et if imbriqué
"""

un_int = int(input("entrez un nombre entier: "))
if un_int > 10:
    if un_int > 20:
        print(un_int,"est plus grand que 20")
    else :
        print(un_int,"est plus grand que 10 et plus petit ou égal à 20")
```

Pour que l'interprétation soit correcte, il est important que le `else` soit imbriqué au même niveau que le `if` imbriqué. L'exemple suivant illustre le problème d'une mauvaise imbrication.

CodePython/chapitre3/ExempleImbricationIncorrecte.py

```
"""
Exemple if sans else et if imbriqué
```

```
3. """
4.
5. un_int = int(input("entrez un nombre entier: "))
6. if un_int > 10:
7.     if un_int > 20:
8.         print(un_int,"est plus grand que 20")
9. else :
10.     print(un_int,"est plus grand que 10 et plus petit ou égal à 20")
```

Exercice. Faites exécuter le code précédent et vérifiez l'effet d'entrer le nombre 15.

Exercice. Lire deux entiers et afficher la division entière du premier par le deuxième. Si le diviseur est 0, afficher un message à cet effet.

Solution. CodePython/chapitre3/ExerciceIfDiviseur0.py

```
1. """
2. Exercice if division. Si le diviseur est nul, afficher un message
3. """
4.
5. dividende = int(input("entrez un nombre entier, le dividende : "))
6. diviseur = int(input("entrez un nombre entier, le diviseur : "))
7. if diviseur != 0:
8.     print("Le résultat de ", dividende, "divisé par",diviseur,"est :", dividende//diviseur)
9. else :
10.     print("Le diviseur ne peut être nul (0)")
```

Exercice. Lire deux entiers et afficher le maximum des deux. N.B. S'ils sont égaux, il n'y a qu'à afficher un des deux !

Solution. CodePython/chapitre3/ExerciceMax2Entiers.py

```
1. """
2. Exemple afficher le maximum de deux entiers
3. """
4.
5. entier1 = int(input("entrez un nombre entier: "))
6. entier2 = int(input("entrez un nombre entier: "))
7. if entier1 > entier2:
8.     print("Le maximum des deux entiers est :", entier1)
9. else :
10.     print("Le maximum des deux entiers est :", entier2)
```

Exercice. Lire trois entiers et afficher le maximum des trois.

Solution. CodePython/chapitre3/ExerciceMax3Entiers.py

```
1. """
2. Exemple afficher le maximum de trois entiers
3. """
4.
5. entier1 = int(input("entrez un nombre entier: "))
6. entier2 = int(input("entrez un nombre entier: "))
7. entier3 = int(input("entrez un nombre entier: "))
```

```
 8.
 9. if entier1 > entier2:
10.     if entier1 > entier3:
11.         print("Le maximum des trois entiers est :", entier1)
12.     else:
13.         print("Le maximum des trois entiers est :", entier3)
14. else:
15.     if entier2 > entier3:
16.         print("Le maximum des trois entiers est :", entier2)
17.     else:
18.         print("Le maximum des trois entiers est :", entier3)
```

Cette approche devient de plus en plus lourde et compliquée au fur et à mesure que le nombre d'entiers à comparer augmente. Dans l'exercice suivant, utilisez une répétition pour simplifier le code.

Exercice. Lire 5 entiers et afficher la valeur maximale.

Solution. CodePython/chapitre3/ExerciceMax5Entiers.py

```
 1. """
 2. Afficher le maximum de 5 entiers lus
 3. """
 4.
 5. le_maximum_actuel = int(input("entrez un nombre entier: "))
 6. for compteur in range(4):
 7.     entier_lu = int(input("entrez un nombre entier: "))
 8.     if entier_lu > le_maximum_actuel:
 9.         le_maximum_actuel = entier_lu
10. print("Le maximum des 5 entiers lus est:",le_maximum_actuel)
```

Exercice *. Afficher les nombres premiers plus petits que 100.

Exercice. Lire une note entre 0 et 100 inclusivement et afficher la lettre correspondante selon le barème suivant :

0<=note<60	E
60<=note<70	D
70<=note<80	C
80<=note<90	B
90<=note<=100	A

Solution. CodePython/chapitre3/ExerciceNoteABCDE.py

```
1. """
2. Afficher la note littérale correspondant à la note numérique
3. """
4.
5. note = int(input("entrez une note entre 0 et 100 : "))
6. if note < 0:
7.     print("La note doit ne peut être inférieure à 0")
```

```
 8. else:
 9.     if note < 60:
10.         print("E")
11.     else :
12.         if note < 70:
13.             print("D")
14.         else:
15.             if note < 80:
16.                 print("C")
17.             else:
18.                 if note < 90:
19.                     print("B")
20.                 else:
21.                     if note <= 100:
22.                         print("A")
23.                     else:
24.                         print("La note ne peut être supérieure à 100")
```

La clause **elif** qui est une abréviation de **else if** permet de produire le même effet que dans la solution précédente en limitant l'indentation. Dans un **if**, il peut y avoir plusieurs **elif** et à la fin un seul **else** optionnel. L'effet est analogue à l'énoncé **CASE** ou **SWITCH** offert dans d'autres langages.

CodePython/chapitre3/ExempleElif.py

```
 1. """
 2. Afficher la note littérale correspondant à la note numérique
avec la clause elif
 3. """
 4.
 5. note = int(input("entrez une note entre 0 et 100 : "))
 6. if note < 0:
 7.     print("La note doit ne peut être inférieure à 0")
 8. elif note < 60:
 9.     print("E")
10. elif note < 70:
11.     print("D")
12. elif note < 80:
13.     print("C")
14. elif note < 90:
15.     print("B")
16. elif note <= 100:
17.     print("A")
18. else:
19.     print("La note ne peut être supérieure à 100")
```

La version 3.10 de Python introduit l'énoncé **match** qui permet de simplifier la syntaxe de comparaisons multiples en ajoutant des fonctionnalités supplémentaires qui en font une structure très puissante et versatile. L'exemple précédent peut être exprimé avec **match** de la manière suivante :

```
 1. """
 2. Afficher la note littérale correspondant à la note numérique
avec match (python 3.10)
 3. """
 4.
 5. note = int(input("entrez une note entre 0 et 100 : "))
 6. match note :
 7.     case note if note < 0:
 8.         print("La note doit ne peut être inférieure à 0")
 9.     case note if note < 60:
10.         print("E")
11.     case note if note < 70:
12.         print("D")
13.     case note if note < 80:
14.         print("C")
15.     case note if note < 90:
16.         print("B")
17.     case note if note <= 100:
18.         print("A")
19.     case _:
20.         print("La note ne peut être supérieure à 100")
```

3.6 Interruption d'une répétition

Il est parfois utile d'interrompre le déroulement normal d'une répétition à l'intérieur du bloc de la répétition. Revoyons l'exemple qui consiste à lire une série d'entier jusqu'à ce que l'entier 0 soit entré et à en produire la somme. Dans la solution précédente avec un `while`, mais sans l'opérateur morse, deux énoncés de lecture étaient employés, un avant le `while` et un autre comme dernier énoncé du bloc du `while`. Dans la solution suivante, un seul énoncé de lecture est suffisant. Lorsque le 0 est lu à l'intérieur du `while`, la répétition est interrompue par l'énoncé `break`.

CodePython/chapitre3/ExempleWhileBreak.py

```
 1. """
 2. Lire une suite d'entiers jusqu'à ce que l'entier 0 soit entré et afficher la
somme
 3. des entiers lus. Exemple du break.
 4. """
 5.
 6. somme=0
 7. while True:
 8.     chaine = input("Entrez un nombre entier, 0 pour terminer :")
 9.     if chaine != '0' :
10.         somme = somme + int(chaine)
11.     else:
12.         break
13. print("La somme des entiers est:",somme)
```

L'énoncé `continue` permet de terminer l'itération en cours en passant directement à l'itération suivante sans arrêter la répétition comme le `break`. L'exemple suivant consiste à lire une série d'entier jusqu'à ce que l'entier 0

soit entré et produire la somme de ces entiers. Il faut omettre les entiers négatifs du calcul. Dans la répétition, si l'entier lu est négatif, le `continue` passe à l'itération suivante.

CodePython/chapitre3/ExempleContinue.py

```
"""
Lire une suite d'entiers jusqu'à ce que l'entier 0 soit entré et afficher la
somme
des entiers lus. Omettre les entiers négatifs. Exemple du continue.
"""

somme=0
while True:
    entier_lu = int(input("Entrez un nombre entier, 0 pour terminer :"))
    if entier_lu > 0 :
        somme = somme + entier_lu
    elif entier_lu < 0:
        continue
    else:
        break
print("La somme des entiers non négatifs est:",somme)
```

Exercice. Cherchez une solution plus compacte avec l'opérateur morse.

Solution. CodePython/chapitre3/ExerciceSommeNonNegMorse.py

```
"""
Lire une suite d'entiers jusqu'à ce que l'entier 0 soit entré et afficher la somme
des entiers lus. Omettre les entiers négatifs. Exemple avec opérateur morse.
"""

somme=0
while (entier_lu := int(input("Entrez un nombre entier, 0 pour terminer :"))) != 0:
    if entier_lu > 0 :
        somme = somme + entier_lu
print("La somme des entiers non négatifs est:",somme)
```

3.7 Qualité du logiciel, tests et débogage

Un aspect fondamental de la qualité d'un programme est la validité des résultats produits par rapport à sa spécification. Une pratique courante en développement de logiciel est de vérifier que les résultats corrects sont produits pour un ensemble de cas de tests.

Pour tester le programme précédent, on pourrait employer plusieurs combinaisons de données en entrée et vérifier que pour chacun des cas de tests, le bon résultat est produit, tel qu'illustré par le tableau suivant :

Numéro de test	Input	Output
1	15 120 30 0	165
2	10 -5 20 0	30
3	0	0
4	2a	Exception

Ce genre de test est dit *fonctionnel* étant donné qu'il vérifie que le fonctionnement du programme est valide par rapport à sa spécification fonctionnelle. Dans des programmes plus complexes, d'autres aspects peuvent aussi être mesurés tel que le temps de calcul, la mémoire consommée ou d'autres aspects dits *non fonctionnels*.

Il existe des méthodes et logiciels qui visent à automatiser le processus de vérification par des tests. Généralement, même si tous les tests produisent le résultat correct, ceci ne garantit pas que le programme fonctionne correctement dans tous les cas possibles. Il est habituellement impraticable de tester tous les cas. Cependant en choisissant les cas de tests d'une manière judicieuse, on peut obtenir un grand niveau de confiance au sujet du fonctionnement du programme. Différentes stratégies peuvent être employées à cet effet.

Dans l'approche de test par boîte noire ou opaque (*black box testing*), les tests sont choisis sans examiner le code lui-même. On cherche à choisir les cas de tests de manière à produire différentes combinaisons d'input qui couvrent les différentes possibilités prévues dans la spécification du programme. Dans l'approche de test par boîte blanche ou transparente (*white box testing*, *glass box testing*), les tests sont conçus en tenant compte du code. En particulier, il faut tenter de parcourir toutes les parties du code par l'ensemble des tests.

Lorsqu'un test ne produit pas le résultat voulu, il y a un ou plusieurs bogues (*bug*) dans le programme. Le débogage est le processus d'élimination des bogues. Le débogage peut ressembler à un travail de fin limier qui consiste à trouver le code coupable en examinant les indices produits par l'exécution du programme. Il existe des outils débogueurs qui facilitent le dépistage des bogues, par exemple, en permettant d'exécuter les énoncés pas à pas et en inspectant les valeurs des variables. Une autre stratégie consiste à ajouter

des énoncés qui affichent l'état de certaines variables à différents endroits du programme pour en suivre le déroulement d'une manière plus détaillée.

4 Types et expressions

Les types de base ont été introduits à la section 1.2.5. Ce chapitre approfondit les notions de type et d'expression en Python.

4.1 Types numériques et expressions

Quelques exemples d'expressions arithmétiques simples ont été introduits dans les premiers chapitres. Il est possible de former des expressions arithmétiques plus complexes en combinant les opérations au besoin. Comme pour les conventions mathématiques usuelles, lorsqu'il y a plusieurs opérations dans une expression, celles à plus grande priorité sont effectuées en premier. À priorité égale, les opérations sont effectuées de gauche à droite. Le tableau suivant montre la priorité des opérateurs en ordre décroissant de priorité. Le + et - *unaires* servent à préciser le signe des nombres comme dans +3 ou -15, le + étant toujours facultatif. Les parenthèses permettent de modifier cet ordre d'évaluation au besoin.

Opération	Priorité
+, - binaires	0 Basse
*, /, //, %	1
+, - unaires	2
**	3
(,)	4 Haute

L'expression suivante

```
3 + 2 * 6 - 3 - 2 * 4
```

est équivalente à

```
(((3 + (2 * 6)) - 3) - (2*4))
```

dont le résultat est 4. L'évaluation procède donc selon les étapes suivantes :

```
(((3 + (2 * 6)) - 3) - (2*4))
(((3 + 12) - 3) - (2*4))
(((3 + 12) - 3) - 8)
((15 - 3) - 8)
(12 - 8)
4
```

Exercice. Quel est le résultat de l'expression suivante :

```
2+4**2*5+10/2
```

Réécrire l'expression avec des parenthèses qui reflètent la priorité d'évaluation des opérations.

Conseil de génie logiciel

Ne vous fiez pas à la priorité, et mettez des parenthèses en cas de doute !

Lorsque des opérandes de types différents sont combinés, Python effectue des conversions (`cast`) automatiques. L'expression

```
3.4 + 7
```

fait intervenir le `float` 3.4 et le `int` 7. Le `int` sera converti automatiquement en un `float` avant d'effectuer l'opération. La conversion cherche à éviter la perte d'information en faisant une *promotion* à un type plus général. Le tableau suivant montre les promotions valides en Python pour les types numériques.

Type	Promotions automatiques valides
complex	aucune
float	complex
int	float ou complex
bool	int ou float ou complex

Conseil de génie logiciel

Il est préférable de spécifier explicitement les conversions dans les expressions pour clarifier le comportement désiré.

L'expression suivante

```
3.4 + 7
```

est équivalente à

```
3.4 + float(7)
```

La deuxième formulation est préférable car elle rend explicite la conversion désirée.

Manipulation de très grands entiers

Un avantage de Python est la facilité à manipuler de très grands nombres. Si possible, Python emploie une représentation native 32 bits dont la valeur est entre -2 147 483 648 (-2^{31}) et 2 147 483 647 ($2^{31}-1$), pour obtenir une meilleure performance. Mais cet aspect n'est pas visible d'un point de vue

du programme. La conversion en une représentation pour des entiers plus larges est effectuée automatiquement au besoin.

Le calcul de 2^{500} soulèverait une exception dans la plupart des langages !

```
>>> 2**500
327339060789614187001318969682759915221664204604306478948329136809613379640467455488327009232590415715088668412756007100921725654588539305332852758937 6
```

4.2 Module, package et modules numériques prédéfinis

Le mécanisme de module permet de découper le code en morceaux qui sont plus faciles à gérer et à réutiliser.

Notion de module et de package

Un module contient un ensemble d'éléments (fonctions, classes, variables, …) qui se trouvent dans un fichier nommé « *nomModule.py* ». L'énoncé `import nomModule` permet d'employer les éléments du module à l'intérieur d'un programme Python ou d'un autre module. La notation `nomModule.nomElement` permet de faire référence à un élément nommé `nomElement` dans le module `nomModule`. Il est aussi possible de définir un alias pour le nom d'un module par :

```
import nomModule as nomAlias
```

Dans ce cas, on peut désigner l'élément avec `nomAlias.nomElement`. La variante `from nomModule import *` permet d'éviter d'employer le préfixe `nomModule`. Mais ceci peut engendrer des collisions de noms lorsque les mêmes noms sont déjà utilisés. Il est préférable de spécifier explicitement les noms des éléments à importer avec la syntaxe :

```
from nomModule import nomElement
```

Par la suite, on peut désigner l'élément sans le préfixer du nom du module.

Les modules peuvent être regroupés en *packages* potentiellement à plusieurs niveaux si désirés. La notation `nomPackage.nomModule` désigne un module `nomModule` dans un package `nomPackage`,. Un package est représenté concrètement par un dossier qui contient les fichiers de ses modules. Un fichier spécial nommé `__init__.py` inclus dans le dossier du package peut contenir du code qui est exécuté pour initialiser le package.

Python offre plusieurs modules prédéfinis. Le module `math` contient les fonctions mathématiques usuelles (arrondissement, logarithmes, trigonométrie, …) du langage C.

Voici un exemple d'importation du module `math` et d'utilisations de quelques fonctions courantes.

```
>>> import math
```

La fonction `math.ceil(x)` du module `math` retourne le plus petit entier plus grand que *x*.

```
>>> math.ceil(3.4)
4
```

Après avoir importé le module, il est possible d'importer le nom de la fonction explicitement pour éviter de spécifier le préfixe :

```
>>> from math import ceil
>>> ceil(3.4)
4
```

La fonction `math.floor(x)` retourne le plus grand entier plus petit que *x*.

```
>>> math.floor(3.4)
3
```

La fonction `math.sqrt(x)` retourne la racine carrée de *x*.

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

La fonction `math.log(x,[base])` retourne $\log_{base}(x)$ et `ln(x)` si l'argument `base` est absent.

```
>>> math.log(8,2)
3.0
```

La variable `math.e` contient le nombre `e`. Ainsi l'appel suivante correspond à `ln(e)`.

```
>>> math.log(math.e)
1.0
```

La fonction `math.cos(x)` retourne le `cos(x)` où `x` est en radian. NB $\pi\ radian = 180\ degrés$.

```
>>> math.cos(1)
0.5403023058681398
```

La variable `math.pi` contient le nombre π.

```
>>> math.cos(math.pi)
-1.0
```

La fonction `math.sin(x)` retourne `sin(x)` où `x` est en radian.

```
>>> math.sin(math.pi/2)
1.0
```

En plus des types numériques de base, Python offre le type `Decimal` qui représente des nombres décimaux exacts par opposition à `float` dont la précision est limitée. Il y a aussi le type `Fraction` qui représente les nombres rationnels. Le module `random` permet de produire des nombres pseudo aléatoires selon différentes distributions. Le module `statistics` permet de calculer diverses fonctions statistiques de base (moyenne, médiane, mode, écart-type, etc.). Le projet `SciPy` contient plusieurs modules employés dans des applications numériques avancés dont le très populaire `NumPy` pour le calcul matriciel.

4.3 Expressions booléennes

Le type `bool` comprend deux littéraux, `True` et `False`, qui correspondent aux entiers 1 et 0. Ce type est utilisé dans les conditions des énoncés `if` et des énoncés `while`. Des expressions booléennes simples ont été vues avec les opérateurs de comparaison (`<,<=, ==, >, >=, !=`) dans le chapitre précédent. Il est aussi possible de formuler des expressions booléennes complexes avec les opérateurs logiques du tableau suivant qui sont ordonnés par la priorité, du moins au plus prioritaire :

Opérateur logique	Signification
or	ou
and	et
not	négation

Les opérateurs de comparaison ont une plus grande priorité que les opérateurs logiques. Il est aussi permis d'employer les parenthèses pour modifier l'ordre de priorité au besoin. Voici quelques exemples d'expressions booléennes.

```
>>> a = 10
>>> a > 5 and a < 15
True
>>> a > 5 and a < 8
False
>>> a > 15 or a < 12
True
>>> a < 0 or a > 20
False
>>> a%5 == 0 or a%7 == 0
True
>>> a%3 == 0 or a%4 == 0
False
```

```
>>> 4*5+7 == 3*9 and 8/2*6 == 2*12 and not 4 < 6
False
```

Comme le `and` a une plus grande priorité que le `or`, le `and` est exécuté avant dans l'exemple suivant :

```
>>> True or True and False
True
```

Comme pour les expressions arithmétiques, les parenthèses permettent de modifier l'ordre d'exécution au besoin :

```
>>> (True or True) and False
False
```

Évaluation court-circuitée des opérateurs logiques

Le `and` n'évalue pas la deuxième partie si la première est `False`. On dit que l'évaluation est court-circuitée. En effet, si la première partie est fausse, le résultat de la condition sera nécessairement faux. Il n'est donc pas nécessaire d'évaluer la deuxième partie. L'exemple suivant montre une utilisation avantageuse de cette manière de procéder. Le code vérifie que `a` est un diviseur de 20. La deuxième condition n'a pas besoin d'être évaluée si `a` est 0 ! Si elle était évaluée, une exception serait levée à cause de division par 0.

```
>>> a = 10
>>> a != 0 and 20%a == 0
True
>>> a = 0
>>> a != 0 and 20%a == 0
False
>>> 20%0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

Dans le cas du `or`, si la première partie est vraie, la deuxième partie de la condition n'est pas évaluée, car le résultat doit être vrai. La condition suivante vérifie que 10 ou 0 est un diviseur de 20. Si l'évaluation n'était pas court-circuitée, une erreur d'exécution serait produite.

```
>>> 20%10 == 0 or 20%0 == 0
True
```

Conventions pour les valeurs de vérité des objets

Python adopte des conventions particulières qui attribuent des valeurs `True` ou `False` aux objets dans le contexte d'une expression booléenne. Par défaut les objets d'une classe retournent la valeur `True`, à moins que la

classe n'implémente la méthode `__bool__()` qui peut retourner la valeur `False` pour certains objets ou encore la méthode `__len__()` dont la valeur 0 se traduit par `False`.

Les valeurs suivantes retournent `False` pour les types prédéfinis :

- `None` et `False`.
- La valeur zéro pour chacun des types numériques : `0, 0.0, 0j, Decimal(0), Fraction(0, 1)`
- Les séquences et collections vides : `'', (), [], {}, set(), range(0)`

De plus les opérateurs booléens `and` et `or` ont des comportements très inhabituels lorsqu'ils sont appliqués aux objets (voir `https://realpython.com/python-and-operator/`).

4.4 Type str : littéraux et opérations

Tel qu'introduit précédemment, une séquence de caractères entre apostrophes (') ou guillemets (") est interprétée comme une chaîne de caractères, c'est-à-dire un littéral de type `str`. Les apostrophes triples (`'''`) ou les guillemets triples (`"""`) doivent être utilisés si la chaîne de caractères chevauche plusieurs lignes. La barre oblique inversée `\` (*backslash*), peut servir à échapper à l'interprétation prévue d'un caractère dans le contexte d'une chaîne. Par exemple, pour inclure une apostrophe dans une chaîne délimitée par des apostrophes, il faut précéder cette apostrophe d'un `\` :

```
>>> print('C\'est un exemple de séquence d\'échappement')
C'est un exemple de séquence d'échappement
```

L'expression *séquence d'échappement* désigne une suite de caractères dont l'interprétation est altérée. Une autre option est d'employer les guillemets pour éviter l'ambiguïté d'inclure une apostrophe :

```
>>> print("C'est un exemple sans séquence d'échappement")
C'est un exemple sans séquence d'échappement
```

Comme le `\` a une interprétation spéciale, pour l'inclure dans la chaîne, il faut le doubler !

```
>>> print("Une chaîne avec \\")
Une chaîne avec \
```

Le tableau suivant contient des exemples de séquences d'échappement en Python :

Séquence d'échappement	Description
\ b	backspace BS
\ t	tabulation HT
\ n	saut de ligne LF
\ f	saut de page FF
\ r	retour de chariot CR
\ "	guillemets "
\ '	apostrophe '
\ \	barre oblique inversée (backslash) \
\<fin de ligne>	Omet le <fin de ligne>
\ooo	Code ooo du caractère en octal
\xhh	Code hh du caractère en hexadécimal

Dans l'exemple suivant, une `<fin de ligne>` est insérée dans la chaîne de caractère avec la séquence d'échappement `\n`.

```
>>> print("Voici un saut de ligne\n et la suite")
Voici un saut de ligne
 et la suite
```

L'exemple suivant affiche `ABC` en employant le code octal des caractères. La séquence `\101` représente la lettre « `A` » avec son code octal `101`.

```
>>> print("\101\102\103")
ABC
```

L'exemple suivant affiche `ABC` en employant le code hexadécimal des caractères. La séquence `\x41` représente la lettre `A` avec son code hexadécimal 41_{16}.

```
>>> print("\x41\x42\x43")
ABC
```

Codage UTF-8 de la norme Unicode

Les caractères d'une chaîne sont représentés en binaire dans la mémoire de l'ordinateur et il y a plusieurs normes qui ont été développées à cet effet. La norme *American Standard Code for Information Interchange* (ASCII) a longtemps prévalu parce qu'elle est compacte. Elle représente les caractères par 128 codes de 7 bits. Le code ASCII permet de représenter les chiffres arabes, les lettres de l'alphabet occidental majuscules et minuscules, des symboles mathématiques de base et la ponctuation. Cependant, la norme est limitée aux lettres sans accents. La norme Latin-1 est une extension sur 8 bits incluant les caractères accentués de la plupart des langues d'Europe de l'Ouest. La norme *Unicode* (www.unicode.org) a été développée pour représenter tous les caractères de toutes les langues et elle est de plus en

plus répandue. Les codes Unicode, appelés *points de code*, sont énumérés dans :#

http://www.unicode.org/Public/UNIDATA/UnicodeData.txt.

Chacun des points de code correspond à un entier qui est représenté par quatre chiffres hexadécimaux. Par exemple, la lettre « A » correspond au nombre hexadécimal à quatre chiffres 0041_{16}.

Différents codages (*Unicode Transformation Format*) sont définis afin de satisfaire à différents contextes d'utilisation (UTF-8, UTF-16, UTF-32, …). Par défaut, Python exploite le codage UTF-8 de taille variable qui représente chacun des points de code par une séquence de 1 à 4 octets. Ce codage emploie un octet (8 bits) dans le cas d'un grand nombre de caractères usuels de la langue anglaise. Les caractères de la norme ASCII ont la même représentation dans UTF-8, ce qui permet une compatibilité avec cette norme plus ancienne. UTF-8 est devenu un codage très populaire pour le Web et la programmation.

Dans un littéral, il est possible d'employer le code Unicode du caractère selon la séquence d'échappement `\uyyyy` pour les points de code à 16 bits ou `\Uyyyyyyyy` pour les points de code à 32 bits où `y` est un chiffre hexadécimal qui correspond à quatre bits.

L'exemple suivant affiche `ABC` en employant la séquence d'échappement Unicode à 16 bits.

```
>>> print("\u0041\u0042\u0043")
ABC
```

UTF-8 est le codage employé par défaut pour le code source Python depuis la version 3. Auparavant, c'était le code ASCII. Il est possible d'altérer le codage employé par défaut en mettant un commentaire spécial à la première ou deuxième ligne du code Python. Le commentaire suivant force l'utilisation du codage UTF-8 :

```
# -*- coding: utf-8 -*-
```

Le commentaire suivant force l'utilisation du codage ASCII.

```
# -*- coding: ASCII -*-
```

Pour empêcher le `\` d'être interprété comme séquence d'échappement dans le contexte d'une chaîne, on peut employer une chaîne brute (*raw string*) en préfixant la chaîne de `r` :

```
>>> print(r"\u0041\u0042\u0043")
\u0041\u0042\u0043
```

4.4.1 Opérations du type str

Le type `str` offre un grand nombre d'opérations prédéfinies. Python concatène automatiquement une suite de chaînes.

```
>>> 'partie1 ' 'partie2 ' 'partie3'
'partie1 partie2 partie3'
```

L'opération **+** produit aussi le même effet.

```
>>> 'partie1 ' + 'partie2 ' + 'partie3'
'partie1 partie2 partie3'
```

Ainsi, l'opération **+** n'a pas le même effet sur une chaîne de caractère que sur un nombre. Cependant, la concaténation sur les chaînes est en quelque sorte analogue à l'addition sur les nombres.

Surcharge d'une opération

Le fait d'employer le même symbole pour désigner plusieurs opérations différentes mais analogues distinguées par la nature des données sur lesquelles l'opération porte, est appelée la *surcharge* du symbole. Ce concept est applicable aux opérations mais aussi à d'autres éléments du langage.

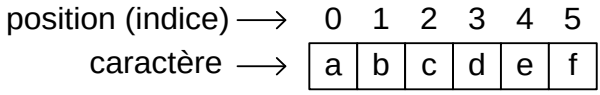
La multiplication (*) d'une chaine par un entier a pour effet de répéter la chaîne :

```
>>> 3*'abc'
'abcabcabc'
>>> 3*'abc'*2
'abcabcabcabcabcabc'
>>> 3*'abc'+2*'def'
'abcabcabcdefdef'
```

La fonction `len()` retourne la longueur d'une chaîne.

```
>>> len('abcdef')
6
```

Le type `str` fait partie des *séquences Python*. Python offre plusieurs opérations communes à tous les types séquence, dont l'accès par un indice à un élément de la séquence. La notation `une_chaine[i]` retourne le caractère en position `i` de la chaîne où `i` est un entier dans l'intervalle de 0 à *n*-1, *n* est la longueur de la chaîne. Dans ce contexte, `i` joue le rôle d'un *indice*.



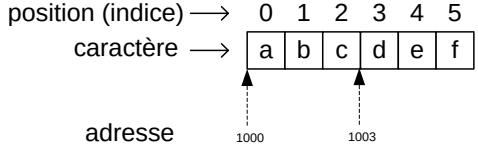
Un objet de type `str` est représenté par une suite contiguë de cases mémoires.

Tableau (*array*)

La représentation par une séquence de cases contiguës qui permet un accès rapide par un indice est traditionnellement appelé un tableau.

Ceci permet de représenter une chaîne de caractère d'une manière compacte et l'accès à un caractère particulier en passant par un indice est très efficace étant donné que la position en mémoire est facile à calculer en ajoutant la position du début du tableau à l'indice multiplié par le nombre d'octets par caractère. En supposant que chacun des caractères occupe un octet, la figure suivante montre que l'adresse du caractère en position 3 est calculé simplement par :

$$adresse\ début + indice * taille\ d'un\ caractère = 1000 + 3 * 1 = 1003$$



Dans l'exemple suivant, `une_chaine[3]` est affecté à la variable `le_caractere` :

```
>>> une_chaine = 'abcdef'
>>> le_caractere = une_chaine[3]
>>> le_caractere
'd'
>>> type(le_caractere)
<class 'str'>
```

Le caractère retourné est en fait une chaîne de longueur 1 de type `str`. En Python, par opposition à d'autres langages, il n'y a pas un type particulier pour le cas d'un seul caractère.

Objet muable (*mutable*) ou immuable (*immutable*)

En Python, il y a deux catégories d'objets importantes à distinguer : les objets *muables* et *immuables*. Un objet muable peut être modifié par opposition à un objet immuable.

Un `str` est immuable. Dans l'exemple suivant, une erreur est soulevée par la tentative de modification d'un caractère de la chaîne :

```
>>> une_chaine[1] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Exercice. Lire une chaîne et vérifier si la lettre `a` est dans la chaîne lue.

Solution. CodePython/chapitre4/ExerciceCherchera.py

```
"""
Lire une chaîne et vérifier si 'a' est dans la chaîne
"""

une_chaine = input("Entrez une chaîne de caractères :")
indice = 0
trouve = False
while indice < len(une_chaine) and not(trouve) :
    if une_chaine[indice] == 'a' :
        trouve = True
    indice = indice + 1
print(trouve)
```

Cette manière de procéder est une implémentation de la recherche linéaire qui sera étudiée plus en détails au chapitre 12. Une autre solution serait d'interrompre le `while` par un `break` lorsqu'un `a` est trouvé. Ceci évite de tester la variable booléenne `trouve` à chacune des itérations.

CodePython/chapitre4/ExempleBreak.py

```
"""
Lire une chaîne et vérifier si 'a' est dans la chaîne
"""

une_chaine = input("Entrez une chaîne de caractères :")
indice = 0
trouve = False
while indice < len(une_chaine) :
    if une_chaine[indice] == 'a' :
        trouve = True
        break
```

```
12.     indice = indice + 1
13. print(trouve)
```

4.4.2 Itération avec un *for* sur une séquence

Le `for` permet d'itérer sur n'importe quelle séquence en parcourant chacun de ses éléments un à un avec une syntaxe simple à coder et à comprendre. Cette possibilité d'itérer sur les éléments est possible parce qu'un `str` Python est un objet *itérable*. Cet aspect est étudié en détails à la section 7.5. L'exemple suivant montre l'emploi du `for` pour itérer sur les éléments d'une chaîne de caractères.

CodePython/chapitre4/ExempleForInChaine.py

```
 1. """
 2. Lire une chaîne et vérifier si 'a' est dans la chaîne avec un
for
 3. """
 4.
 5. une_chaine = input("Entrez une chaîne de caractères :")
 6. trouve = False
 7. for un_caractere in une_chaine :
 8.     if un_caractere == 'a' :
 9.         trouve = True
10.         break
11. print(trouve)
12.
```

La variable `un_caractere` prend successivement comme valeur chacun des caractères de la chaîne. Cet exemple montre que le `break` peut aussi être employé avec le `for`.

Il serait aussi possible de parcourir la chaîne de caractères en passant par un indice qui parcoure les éléments comme dans l'exemple suivant.

```
 1. """
 2. Lire une chaîne et vérifier si 'a' est dans la chaîne avec un
for et un indice
 3. """
 4.
 5. une_chaine = input("Entrez une chaîne de caractères :")
 6. trouve = False
 7. for indice in range(len(une_chaine)) :
 8.     if une_chaine[indice] == 'a' :
 9.         trouve = True
10.         break
11. print(trouve)
12.
```

Cette approche est typique de l'utilisation du `for` dans la plupart des langages de programmation. Cependant, cette solution est moins conforme

à l'approche Python, dite *pythonique*, qui permet d'itérer directement avec une variable qui parcoure la séquence itérable. De plus, le passage par un indice est moins performant.

D'autre part, il est possible d'itérer sur les éléments et leurs indices de manière pythonique en passant par la fonction `enumerate()`. L'exemple suivant vérifie si le caractère `a` est présent et retourne sa position dans la chaîne si c'est le cas :

CodePython/chapitre4/ExempleForEnumerate.py

```
"""
Lire une chaîne et vérifier si 'a' est dans la chaîne.
Afficher la position en passant par un for avec enumerate.
"""

une_chaine = input("Entrez une chaîne de caractères :")
trouve = False

for indice, un_caractere in enumerate(une_chaine) :
    if un_caractere == 'a' :
        trouve = True
        indice_a = indice
        break
if trouve :
    print("Le caractère a est présent à la position :",
indice_a)
else :
    print("Le caractère a n'est pas dans la chaîne")
```

Le `enumerate()` permet d'itérer sur les couples `(indice, élément)` pour chacun des éléments de la séquence.

4.4.3 Recherche de sous-chaîne avec *in*

Le `in` permet de vérifier si une chaîne est une sous-séquence d'une autre :

```
>>> 'a' in 'tarte'
True
>>> 'b' in 'tarte'
False
>>> 'art' in 'tarte'
True
```

4.4.4 Extraction d'une tranche (*slice*) d'une chaîne

La notation `s[i:j]` retourne la sous-séquence des éléments d'indice `k`, tel que `i<=k<j`.

```
>>> s ='0123456789'
>>> s[3:7]
'3456'
```

La notation `s[i:]` retourne la sous-séquence des éléments d'indice `k`, tel que `i<=k<len(s)`.

```
>>> s[4:]
'456789'
```

La notation `s[:j]` retourne la sous-séquence des éléments d'indice `k`, tel que `0<=k<j`.

```
>>> s[:5]
'01234'
```

La notation `s[i:j:p]` retourne la sous-séquence des éléments d'indice `k`, tel que `i<=k+np<j, n >= 0`.

```
>>> s[2:9:2]
'2468'
```

Il est permis d'employer un pas négatif :

```
>>> s[5:2:-1]
'543'
```

Lorsque `i` et `j` sont absents, avec un pas `k` négatif, c'est comme si on employait les valeurs `i=-1, j=-(len(s)+1)`. L'interprétation d'une valeur d'indice négatif, signifie que l'indice est compté en partant de la fin de la chaine. On obtient ainsi la séquence : `-1,-2,…,-len(s)`. A noter que la dernière valeur `-(len(s)+1)` est exclue de la séquence. L'effet est donc d'inverser la chaîne.

```
>>> s[::-1]
'9876543210'
>>> s[-1:-11:-1]
'9876543210'
```

Il est aussi possible d'employer des indices positifs avec un incrément négatif :

```
>>> s[8:2:-2]
'864'
```

Exercice. Quel est le résultat de `s[len(s):0:-1]`

Solution.

```
'987654321'
```

Exercice. Quel est le résultat de `s[len(s):-1:-1]`

Solution.

```
''
```

Exercice. Quel est le résultat de `s[len(s):-2:-1]`

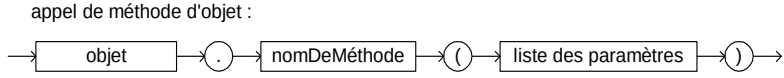
Solution.
```
'9'
```

Exercice. Quel est le résultat de `s[-1:-9:-1]`

Solution.
```
'98765432'
```

4.4.5 Méthodes du type str

Chacun des types a un ensemble de méthodes associées. Une méthode d'objet est une fonction qui s'applique à un objet. Une méthode est semblable à une fonction mais elle est appelée avec la syntaxe suivante :



La méthode `str.find()` retourne la position de la première occurrence d'une sous-chaîne dans la chaîne correspondant à l'objet désigné.

```
>>> un_str = 'abracadabra'
>>> un_str.find('a')
0
>>> un_str.find('c')
4
```

Si la sous-chaîne est absente, la méthode retourne -1.

```
>>> un_str.find('e')
-1
```

Les deuxième et troisième arguments optionnels permettent de limiter la recherche à une tranche de l'objet. L'appel suivant fait la recherche de `a` dans la tranche `un_str[4 :7]` :

```
>>> un_str.find('a',4,7)
5
```

La méthode `str.count()` compte le nombre d'occurrences d'une sous-chaîne dans l'objet désigné.

```
>>> un_str = 'abracadabra'
>>> un_str.count('a')
5
>>> un_str.count('c')
1
>>> un_str.count('e')
0
>>> un_str.count('bra')
```

```
2
>>> un_str.count('a',4,7)
1
>>> un_str.count('a',4,9)
2
```

Plusieurs autres méthodes sont disponibles, voir :

```
https://docs.python.org/fr/3/library/stdtypes.html#str
```

4.5 Type list

Le type `str` permet de représenter une séquence de caractères alors que le type `list` permet de représenter une séquence d'éléments de type quelconque. Le traitement effectué sur une chaîne ou une liste est souvent similaire et plusieurs aspects se ressemblent. Ces deux types font partie des séquences Python. Un littéral de type `list` est composé d'un crochet ouvrant `[` suivi d'une liste d'éléments séparés par des virgules et terminé par un crochet fermant `]`.

```
>>> liste_de_int = [5,2,7,3,10,-4]
>>> liste_de_noms = ['Pierre','Jean','Jacques']
>>> liste_de_int
[5, 2, 7, 3, 10, -4]
>>> liste_de_noms
['Pierre', 'Jean', 'Jacques']
```

La fonction `len()` donne la taille de la liste.

```
>>> len(liste_de_int)
6
```

La fonction `list()` permet de produire une liste à partir d'un autre type de séquence tel qu'un `range()` :

```
>>> list(range(4))
[0, 1, 2, 3]
```

Une liste permet aussi de représenter une collection *hétérogène* formée d'éléments de types différents. En particulier, un élément d'une liste peut être lui-même une liste ! Ceci permet de construire des structures de données complexes avec des éléments enchâssés les uns dans les autres à volonté.

```
>>> liste_melangee = ['Paul',150,30,['golf','tennis','hockey']]
>>> liste_melangee
['Paul', 150, 30, ['golf', 'tennis', 'hockey']]
```

Un tableau à deux dimensions peut être représenté par une liste de listes.

```
>>> t = [[1,2,3],[4,5,6]]
>>> t
```

```
[[1, 2, 3], [4, 5, 6]]
```

Il est possible de répéter un élément à plusieurs reprises dans une liste.

```
>>> liste_avec_repetition = [4,2,2,6,2,4]
>>> liste_avec_repetition
[4, 2, 2, 6, 2, 4]
```

Les opérations d'addition et de multiplication ont un effet analogue au cas des `str`. Le littéral `[]` représente une liste vide. Pour ajouter des éléments d'une liste à une autre liste, l'opération d'addition peut être employée :

```
>>> liste_fruits = []
>>> liste_fruits = liste_fruits + ['orange']
>>> liste_fruits
['orange']
>>> liste_fruits = liste_fruits + ['pomme']
>>> liste_fruits
['orange', 'pomme']
```

4.5.1 Accès par indice

Comme une liste est aussi une séquence, les opérations communes aux séquences vues pour les `str` sont aussi applicables aux listes. Il est ainsi possible d'accéder à un élément avec la notation indicée `nom_liste[indice]` :

```
>>> liste_de_int[2]
7
>>> liste_de_noms[1]
'Jean'
>>> liste_melangee[3]
['golf', 'tennis', 'hockey']
```

Les objets d'une liste sont ordonnés en ce sens qu'ils sont en ordre de l'indice. Cependant, l'ordre dépend de la manière dont les objets sont placés dans la liste soit à sa création ou par les opérations de modification, et non pas l'ordre des objets eux-mêmes. Par exemple, dans `liste_de_int`, les entiers ne sont pas en ordre numérique. La méthode `sort()` permet de trier les objets en fonction de l'ordre défini sur ces objets (voir plus loin).

Dans le cas d'une liste de listes, il est possible d'employer une suite d'indices qui sont appliqués en série.

```
>>> t
[[1, 2, 3], [4, 5, 6]]
>>> t[1]
[4, 5, 6]
>>> t[1][2]
6
```

4.5.2 Itération avec *for*

Le `for` permet de parcourir les éléments d'une liste de manière analogue à l'itération sur une chaîne. Dans l'exemple suivant, le `for` parcoure les éléments de la liste un par un et les affiche.

```
>>> liste_de_int = [5,2,7,3,10,-4]
>>> for un_int in liste_de_int :
...     print(un_int)
...
5
2
7
1
3
10
-4
```

La réalisation d'une telle boucle `for` est intimement liée au mécanisme d'itérateur Python qui sera expliquée plus loin. De manière analogue au cas des chaînes de caractère, le `enumerate()` permet l'accès aux indices avec les éléments correspondants :

```
>>> liste_de_int = [5,2,7,3,10,-4]
>>> for indice, un_int in enumerate(liste_de_int) :
...     print(indice, un_int)
...
0 5
1 2
2 7
3 3
4 10
5 -4
```

Exercice. Lire 5 entiers, afficher la moyenne et les entiers supérieurs à la moyenne.

CodePython/chapitre4/ExerciceSuperieurAMoyenne.py

```
 1. """
 2. Lire 5 entiers, afficher la moyenne et les entiers supérieurs à
la moyenne.
 3. """
 4. somme = 0
 5. liste_de_int = []
 6. for indice in range(5):
 7.     un_int = int(input('Entrez un entier:'))
 8.     somme = somme + un_int
 9.     liste_de_int = liste_de_int + [un_int]
10. moyenne = somme / 5
11. print('La moyenne est :',moyenne)
12. print('Liste des entiers lus supérieurs à la moyenne :')
13. for un_int in liste_de_int:
```

```
14.     if un_int> moyenne :
15.         print(un_int)
```

Un aspect important à considérer dans cet exercice concerne l'utilisation d'une liste pour mémoriser les données lues afin de pouvoir les parcourir à nouveau après le calcul de la moyenne.

Exercice. Ajouter l'affichage des indices des éléments à l'exemple précédent en exploitant le `enumerate()`.

4.5.3 Test d'appartenance à la liste avec *in*

Le *in* permet de vérifier l'appartenance d'un élément à la liste :

```
>>> liste_de_int = [5,2,7,3,10,-4]
>>> 3 in liste_de_int
True
>>> liste_de_noms = ['Pierre','Jean','Jacques']
>>> 'Paul' in liste_de_noms
False
```

4.5.4 Extraction d'une tranche d'une liste

La syntaxe pour l'extraction d'une tranche d'une liste est la même que pour `str`.

```
>>> liste_de_int = [5,2,7,3,10,-4]
>>> liste_de_int[2:4]
[7, 3]
>>> liste_de_int[3:]
[3, 10, -4]
>>> liste_de_int[:2]
[5, 2]
>>> liste_de_int[0:5:2]
[5, 7, 10]
```

4.5.5 Liste : séquence muable (*mutable*)

Par opposition au type `str`, une liste est muable. L'élément `liste_de_int[2]` est modifié dans l'exemple suivant :

```
>>> liste_de_int
[5, 2, 7, 3, 10, -4]
>>> liste_de_int[2] = 5
>>> liste_de_int
[5, 2, 5, 3, 10, -4]
```

Il est permis de modifier une tranche d'une liste en remplaçant celle-ci par une liste dont la taille peut être différente.

```
>>> liste_de_int[2:4] = [8,6]
>>> liste_de_int
[5, 2, 8, 6, 10, -4]
>>> liste_de_int[2:5] = [9]
>>> liste_de_int
```

```
[5, 2, 9, -4]
>>> liste_de_int[1:3] = []
>>> liste_de_int
[5, -4]
```

4.5.6 Méthodes et fonctions du type list

Cette section montre quelques exemples de méthodes et de fonctions d'usage courant applicables au type `list`. La méthode `list.index()` retourne la position de la première occurrence de l'élément désigné.

```
>>> liste_de_int = [4,2,3,3,5,1,8,3,4,3]
>>> liste_de_int.index(3)
2
```

La méthode `list.count()` compte le nombre d'occurrences de l'élément désigné.

```
>>> liste_de_int.count(3)
4
```

Comme les listes sont muables, plusieurs méthodes permettent de les modifier. La méthode `list.append()` ajoute un nouvel élément à la fin de la liste.

```
>>> liste_de_int.append(2)
>>> liste_de_int
[4, 2, 3, 3, 5, 1, 8, 3, 4, 3, 2]
```

La possibilité de faire croître la dimension d'une liste est un aspect important qui la distingue d'un tableau qui est de taille fixe. Les tableaux de taille fixe ne sont pas natifs en Python contrairement à la plupart des langages de programmation. Le module `NumPy` est prévu à cet effet.

La méthode `list.pop()` supprime un élément à la position désignée et retourne l'élément supprimé.

```
>>> liste_de_int.pop(4)
5
>>> liste_de_int
[4, 2, 3, 3, 1, 8, 3, 4, 3, 2]
```

La méthode `list.sort()` trie les éléments de la liste en fonction de l'ordre des objets. Les éléments sont triés sur place, et la liste est modifiée.

```
>>> liste_de_int = [4,2,3,3,5,1,8,3,4,3]
>>> liste_de_int.sort()
>>> liste_de_int
[1, 2, 3, 3, 3, 3, 4, 4, 5, 8]
```

En revanche, la fonction `sorted()` trie la liste et retourne une nouvelle liste triée. La liste de départ demeure intacte.

```
>>> liste_de_int = [4,2,3,3,5,1,8,3,4,3]
>>> liste_triee = sorted(liste_de_int)
>>> liste_de_int
[4, 2, 3, 3, 5, 1, 8, 3, 4, 3]
>>> liste_triee
[1, 2, 3, 3, 3, 3, 4, 4, 5, 8]
```

4.5.7 Représentation interne du type list

La flexibilité du type `list` vient de la représentation interne qui est un tableau dynamique de références à des objets de types quelconques.

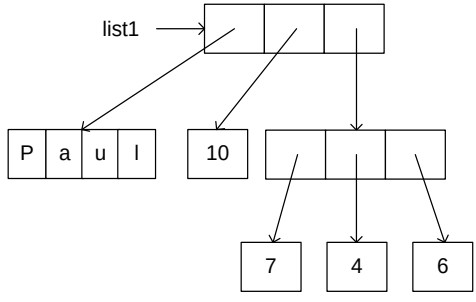
Attention ! Liste ou tableau ?

L'emploi du terme `list` en Python peut porter à confusion parce que traditionnellement dans plusieurs langages, une liste désigne des objets reliés sous forme d'une chaîne représentée par des références, ce qui n'est pas efficace pour un accès par un indice. Le type `list` Python est représenté par un tableau au sens plus traditionnel. Le tableau est dynamique parce que Python alloue un tableau en mémoire de taille supérieure au besoin pour accommoder les insertions futures et opère une réorganisation dans un tableau plus large lorsqu'un débordement se produit.

Dans l'exemple suivant, la liste `list1` est un tableau de références à trois objets. La figure ne montre que les cases des trois éléments mais il est possible qu'il en ait d'autres déjà alloués. Le premier objet est un `str`, le second un `int` et le dernier est un autre objet `list`.

```
>>> list1 = ['Paul',10,[7,4,6]]
```

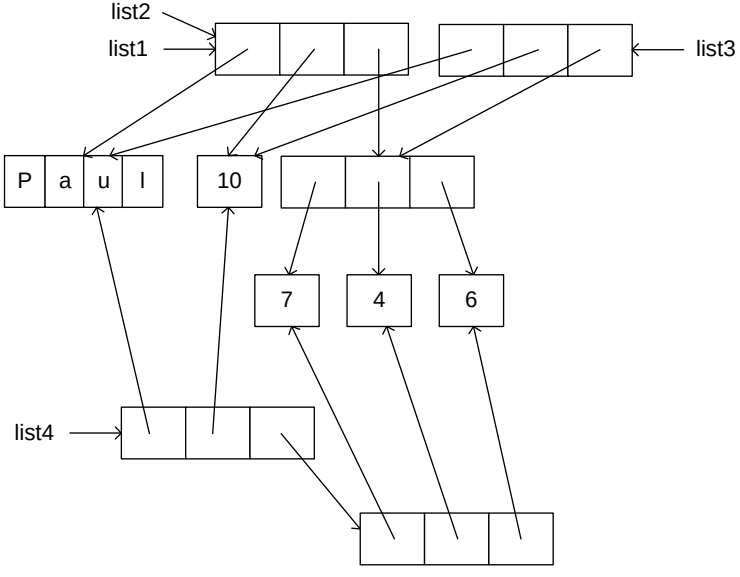
La figure suivante montre le résultat de l'affectation :



Comme toutes cases des références sont de taille identique, la recherche d'un élément particulier est très rapide. Lorsqu'on affecte une liste à une variable, c'est une référence qui est copiée. C'est le cas de `list2` dans l'exemple suivant qui réfère à la même adresse (`id`) que `list1`. Il est

possible d'altérer le comportement d'une affectation, en utilisant le module `copy`. La fonction `copy.copy()` produit une *copie de surface* de la liste (*shallow copy*), en créant une nouvelle liste qui contient les mêmes références que la liste originale, tel qu'illustré par `list3` dans l'exemple. Enfin, la fonction `copy.deepcopy()` produit une *copie profonde*, comme c'est le cas de `list4`. La copie profonde crée un nouvel objet qui est une copie pour tous les objets imbriqués qui sont muables. Ici la liste imbriquée `[7,4,6]` dans `liste1` est copiée dans une nouvelle liste dans *list4*. En revanche, les objets immuables comme le `str 'Paul'` et le `int` 10 ne sont pas dupliqués. Cette stratégie évite de maintenir en mémoire plusieurs copies d'objets identiques qui ne changent jamais parce qu'ils sont immuables.

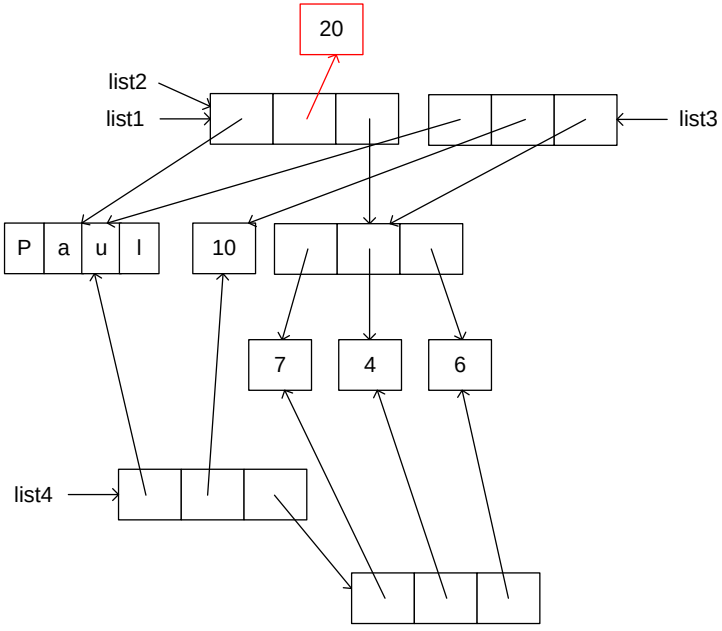
```
>>> list2 = list1
>>> id(list1)
2677777686400
>>> id(list2)
2677777686400
>>> import copy
>>> list3 = copy.copy(list1)
>>> id(list3)
2677777746240
>>> list4=copy.deepcopy(list1)
>>> id(list4)
2677777784128
```



Ceci amène des comportements qui peuvent être surprenants. L'exemple suivant montre l'affectation d'une nouvelle valeur 20 à `list1[1]`. Comme l'objet `int` 10 est immuable, ceci provoque la création d'un nouvel objet

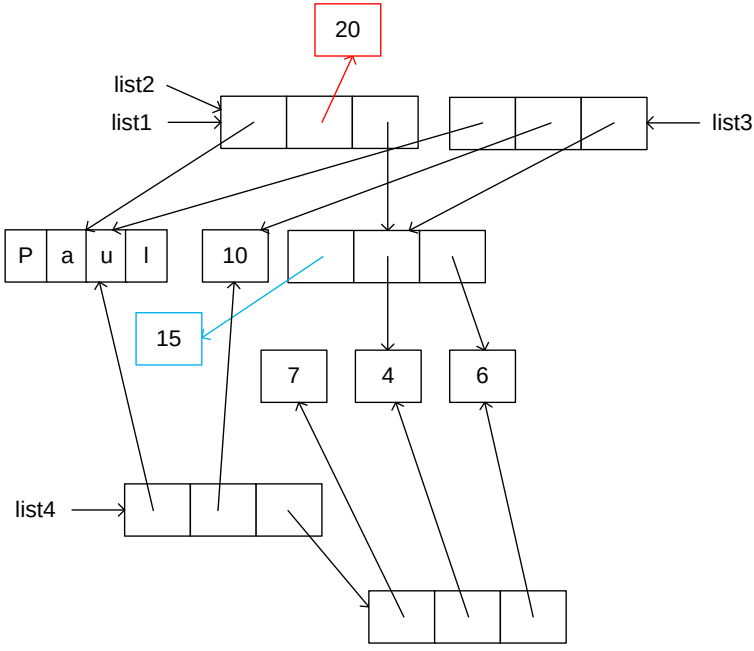
int 20 et le remplacement de la référence à l'objet 10 par celle à l'objet 20. Comme *list1* et *list2* font référence à la même structure, ils sont modifiés. En revanche, *list3* et *list4* ne sont pas affectés.

```
>>> list1[1]=20
>>> list1
['Paul', 20, [7, 4, 6]]
>>> list2
['Paul', 20, [7, 4, 6]]
>>> list3
['Paul', 10, [7, 4, 6]]
>>> list4
['Paul', 10, [7, 4, 6]]
```



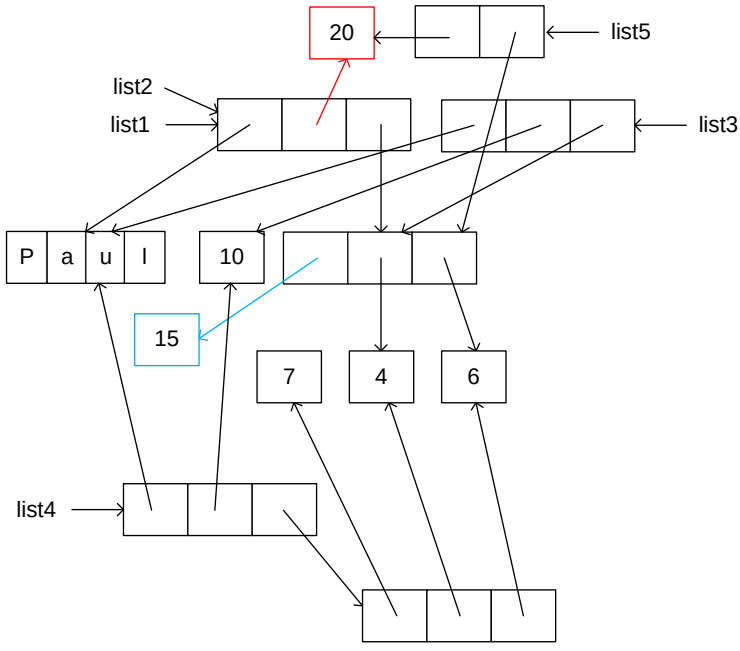
L'exemple suivant montre une modification du premier élément de la sous-liste `list1[2]`. Le comportement est différent. Dans ce cas, comme `list3` réfère au même objet, il est modifié. Cependant, `list4` qui opère une copie profonde réfère à un autre objet et il n'est donc pas modifié !

```
>>> list1[2][0]=15
>>> list1
['Paul', 20, [15, 4, 6]]
>>> list2
['Paul', 20, [15, 4, 6]]
>>> list3
['Paul', 10, [15, 4, 6]]
>>> list4
['Paul', 10, [7, 4, 6]]
```



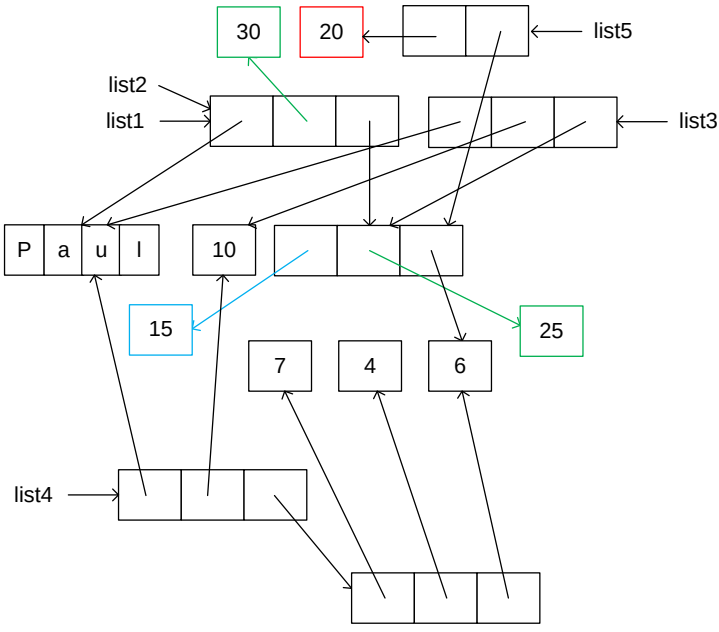
L'affectation d'une tranche d'une liste à une variable provoque l'équivalent d'une copie de surface de la sous-liste désignée :

```
>>> list5 = list1[1:3]
>>> list5
[20, [15, 4, 6]]
```




```
>>> list1[1]=30
>>> list5
[20, [15, 4, 6]]
>>> list1[2][1]=25
>>> list5
[20, [15, 25, 6]]
```

En conséquence, la modification du `int list1[1]` n'affecte pas `list5`. En revanche, la modification de la liste imbriquée `list1[2]` change aussi `list5`.



Exercice. Quelles sont les valeurs de `list1, list2, list3, list4, list5` après l'exécution du code suivant :

```
>>> list1 = ['Paul',150,30,['golf','tennis','hockey']]
>>> list2 = list1
>>> import copy
>>> list3 = copy.copy(list1)
>>> list4=copy.deepcopy(list1)
>>> list5=list1[1:4]
>>> list1[1]=100
>>> list1[3][2]= 'surf'
```

Solution:

```
>>> list1
['Paul', 100, 30, ['golf', 'tennis', 'surf']]
>>> list2
['Paul', 100, 30, ['golf', 'tennis', 'surf']]
>>> list3
['Paul', 150, 30, ['golf', 'tennis', 'surf']]
>>> list4
['Paul', 150, 30, ['golf', 'tennis', 'hockey']]
>>> list5
[150, 30, ['golf', 'tennis', 'surf']]
```

4.5.8 Liste en compréhension

Le mécanisme de liste en compréhension permet de construire une liste avec une syntaxe simplifiée. L'exercice suivant permet de motiver le concept.

Exercice. Construire une liste formée des entiers pairs de 0 à 10 inclusivement et afficher les éléments de la liste.

Solution.

```
>>> liste_de_int_pairs = []
>>> for indice in range(0,11,2) :
...     liste_de_int_pairs.append(indice)
...
>>> liste_de_int_pairs
[0, 2, 4, 6, 8, 10]
```

Le mécanisme de liste en compréhension (*list comprenhension*) en Python permet de produire le même effet mais avec une syntaxe simplifiée analogue avec la définition mathématique d'un ensemble en compréhension. La liste en compréhension est produite ici par l'indice d'un `for` :

```
>>> liste_de_int_pairs = [indice for indice in range(0,11,2)]
>>> liste_de_int_pairs
[0, 2, 4, 6, 8, 10]
```

Une autre façon de produire le même résultat illustre la possibilité d'employer une expression formée à partir de l'indice du `for` :

```
>>> liste_de_int_pairs = [2*indice for indice in range(6)]
>>> liste_de_int_pairs
[0, 2, 4, 6, 8, 10]
```

Il est possible d'inclure une clause `if` pour exclure des éléments du résultat. Ici, la liste retourne les entiers entre 0 et 9 qui ne sont pas des multiples de 3.

```
>>> liste_non_mult3 = [i for i in range(10) if i%3 != 0]
>>> liste_non_mult3
[1, 2, 4, 5, 7, 8]
```

Enfin, il est permis de combiner plusieurs `for` imbriqués. Ici, les *for* imbriqués produisent la liste des paires d'entiers `[i,j]` où `0<=i <3, 0<=j <3` :

```
>>> liste_de_paires = [[i,j] for i in range(3) for j in range(3) ]
>>> liste_de_paires
[[0, 0], [0, 1], [0, 2], [1, 0], [1, 1], [1, 2], [2, 0], [2, 1], [2, 2]]
```

4.6 Type tuple

Le type `tuple` (*n*-uplet) est très proche du type `list`. Comme pour le type `list`, le type `tuple` permet de représenter une séquence d'éléments de types variés possiblement hétérogène. Cependant, le type `tuple` est immuable.

Un littéral de type `tuple` est composé d'une parenthèse ouvrante `(` suivie d'une liste d'éléments séparés par des virgules et terminée par une parenthèse fermante `)` :

```
>>> nuplet_de_int = (5,2,7,3,10,-4)
>>> nuplet_noms = ('Pierre','Jean','Jacques')
>>> nuplet_de_int
(5, 2, 7, 3, 10, -4)
>>> nuplet_noms
('Pierre', 'Jean', 'Jacques')
>>> nuplet_melangee = ('Paul',150,30,('golf','tennis','hockey'))
>>> nuplet_melangee
('Paul', 150, 30, ('golf', 'tennis', 'hockey'))
```

Il est permis d'enchâsser une `list` dans un `tuple` ou l'inverse au besoin.

```
>>> t = ('Paul',150,30,['golf','tennis','hockey'])
>>> t
('Paul', 150, 30, ['golf', 'tennis', 'hockey'])
```

Le fonction `tuple()` permet de construire un `tuple` à partir d'un autre type de séquence tel qu'une liste ou un `range()` :

```
>>> tuple(range(4))
(0, 1, 2, 3)
>>> tuple([4,2,7])
(4, 2, 7)
```

Le traitement d'un `tuple` est analogue au type `list`. La fonction `len()`, les opérations `+` et `*`, l'accès par indice, l'itération avec `for`, le `in`, l'extraction de tranche, la définition en compréhension, les méthodes qui ne font pas de modification, fonctionnent de manière analogue.

4.6.1 Type *tuple* immuable (*immutable*)

Par opposition à un objet de type `list`, un tuple est immuable. Les opérations de modification des éléments d'un `tuple` sont interdites. Dans l'exemple suivant, l'élément `nuplet_de_int[2]` ne peut être modifié :

```
>>> nuplet_de_int
(5, 2, 7, 3, 10, -4)
>>> nuplet_de_int[2] = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

En revanche, si le `tuple` contient un objet muable comme une `list`, il est permis de modifier le contenu de la liste :

```
>>> t = ('Paul',150,30,['golf','tennis','hockey'])
>>> t
('Paul', 150, 30, ['golf', 'tennis', 'hockey'])
>>> t[3][1]=5
>>> t
('Paul', 150, 30, ['golf', 5, 'hockey'])
```

4.6.2 Tuple et affectation multiple

Lorsqu'on place une suite d'énoncés dans la partie droite d'une affectation multiple, elle est implicitement convertie en un `tuple`.

```
>>> t = 1,2+1,3*4
>>> t
(1, 3, 12)
```

On obtient donc le même effet que :

```
>>> t = (1,2+1,3*4)
>>> t
(1, 3, 12)
```

Lorsqu'une expression unique est suivie d'une virgule, elle est convertie en un n-uplet qui contient un élément (singleton).

```
>>> t = 2,
>>> t
(2,)
```

Note

Une particularité de Python est de permettre une virgule sans qu'un élément ne suive dans le contexte d'une suite d'éléments séparés par de virgules. Ainsi `(2,)` est équivalent à `(2)`.

Lorsqu'on emploie une affectation multiple, c'est le n-uplet de la partie droite qui est décomposé en plusieurs éléments qui sont assignés aux variables de la partie gauche.

```
>>> a,b,c = 1,2+1,3*4
>>> a,b,c
(1, 3, 12)
>>> a
1
>>> b
3
>>> c
12
```

On peut aussi employer la syntaxe suivante qui rend plus explicite le `tuple` sous-jacent.

```
>>> (a,b,c) = 1,2+1,3*4
>>> a
1
>>> b
3
>>> c
12
```

Les parenthèses dans la partie gauche deviennent nécessaires pour enchâsser les structures :

```
>>> t1,(a,b),x = (1,2),(3,4),5
>>> t1
(1, 2)
>>> a
3
>>> b
4
>>> x
5
```

4.7 Type set

Le type `set` représente une collection d'éléments hachables sans ordre et sans duplication. La sélection par indice n'est pas permise mais la recherche d'un élément est plus rapide qu'avec les autres collections. Un `set` peut être construit avec le constructeur `set()` appliqué à un argument qui est un itérable. Les doubles sont éliminés.

```
>>> s1 = set([1,2,3,2])
>>> s1
{1, 2, 3}
```

Il est aussi possible d'employer les accolades pour désigner un littéral `set`.

```
>>> s1 = {1,2,3,2}
>>> s1
{1, 2, 3}
```

En revanche, l'ensemble vide ne peut être représenté par les accolades :

```
>>> s3 = {}
>>> s3
{}
>>> type(s3)
<class 'dict'>
```

Ainsi `{}` est de type `dict` !

Il faut employer `set()` pour représenter l'ensemble vide :

```
>>> s4=set()
>>> s4
set()
>>> type(s4)
<class 'set'>
```

Un `set` peut être hétérogène, mais les éléments du `set` doivent être immuables. En conséquence, un `set` peut contenir un `tuple` mais pas un `list` ou un `set`.

```
>>> set_melangee=set(['Paul',150,30,('golf','tennis','hockey')])
>>> s5 = set(['Paul',150,30,['golf','tennis','hockey']])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> set_melangee
{'Paul', ('golf', 'tennis', 'hockey'), 150, 30}
```

Un `set` n'est pas une séquence Python. Les éléments ne sont pas ordonnés et l'adressage par indice n'est pas permis :

```
>>> s1
{1, 2, 3}
>>> s1[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
```

La recherche d'un élément dans un `set` avec `in` est définie :

```
>>> 2 in s1
True
>>> 5 in s1
False
```

La fonction `len()` donne la taille du `set` :

```
>>> len(s1)
3
```

Le `set` est itérable avec `for` :

```
>>> for element in s1:
...     print(element)
...
1
2
3
```

Les opérations ensemblistes sont définies pour le type set. L'union est exprimée par `|`, l'intersection par `&`, la différence par `-` et la différence symétrique par `^` :

```
>>> s1
{1, 2, 3}
>>> s2 = {4,3,2}
>>> s2
{2, 3, 4}
>>> s1|s2
{1, 2, 3, 4}
>>> s1&s2
{2, 3}
>>> s1-s2
{1}
>>> s1^s2
{1, 4}
```

Ces opérations peuvent aussi être effectuées par les méthodes *union*(), *intersection*(), *difference*() et *symmetric_difference*(). Les opérations ensemblistes ne sont pas permises sur les `list` :

```
>>> [1,2,3,2] |[4,3,2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for |: 'list' and 'list'
```

Les opérations d'inclusion ensembliste sont définies par les opérateurs `<, <=, >, >=` :

```
>>> {2,3}<s1
True
>>> s1<=s1
True
>>> s1<s2
False
```

Les méthodes `isdisjoint()`, `issubset()`, `issuperset()` sont aussi définies.

Un `set` est muable même si ses éléments ne sont pas muables, et plusieurs opérations et méthodes permettent de le modifier. Pour chacun des opérateurs ensemblistes (`|,&,-,^`), il y a un opérateur de modification correspondant. Le `|=` ajoute les éléments du set à droite de l'opération à la variable `set` à gauche, ce qui est l'équivalent d'affecter l'union des deux ensembles à la variable de gauche.

```
>>> s1 = {3,1,2}
>>> s2 = {3,4,2}
>>> s1 |= s2
>>> s1
{1, 2, 3, 4}
```

Les opérateurs d'affectation `&=, -=, ^=` sont définis de manière analogue.

```
>>> s1 = {3,1,2}
```

```
>>> s2 = {3,4,2}
>>> s1 &= s2
>>> s1
{2, 3}
>>> s1 = {3,1,2}
>>> s2 = {3,4,2}
>>> s1 -= s2
>>> s1
{1}
>>> s1 = {3,1,2}
>>> s2 = {3,4,2}
>>> s1 ^= s2
>>> s1
{1, 4}
```

Les méthodes `update()`, `intersection_update()`, `difference_update()` et `symmetric_difference_update()` permettent des mises-à-jour analogues.

Il est possible d'ajouter un élément avec la méthode `add()` :

```
>>> s1 = {3,1,2}
>>> s1.add(5)
>>> s1
{1, 2, 3, 5}
```

`Remove()` enlève un élément.

```
>>> s1 = {3,1,2}
>>> s1.remove(2)
>>> s1
{1, 3}
```

Une exception est levée si l'élément est absent :

```
>>> s1 = {3,1,2}
>>> s1.remove(7)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 7
```

`Discard()` fait comme `remove()` mais ne soulève pas d'exception si l'élément est absent. `Pop()` enlève un élément quelconque et `clear()` vide l'ensemble.

Le type `frozenset` est analogue à un `set` mais il est immuable.

```
>>> fs1=frozenset([3,1,2])
>>> fs1.add(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

Les opérateurs de mise à jour sont permis mais l'effet est de créer un nouvel objet. L'exemple suivant montre qu'après la mise-à-jour par l'union des deux ensembles, le id de fs1 a changé.

```
>>> id(fs1)
1313491197504
>>> fs1|= frozenset([4])
>>> fs1
frozenset({1, 2, 3, 4})
>>> id(fs1)
1313491197056
```

Un set est représenté avec une table de hachage (*hash table*). Le principe du hachage est d'appliquer une fonction de hachage à un élément qui détermine son adresse dans un tableau. Il est important que la fonction répartisse de manière uniforme les éléments dans le tableau. Lorsque différents éléments produisent la même adresse par la fonction de hachage, il y a collision et différentes stratégies peuvent être appliquées pour résoudre les collisions. Comme le calcul se fait rapidement, l'accès est très rapide. Les éléments doivent être hachables.

4.8 Type dict

Dans plusieurs applications, il faut retrouver des données dans une collection à partir d'une clé d'accès. Par exemple, dans une application de contacts téléphoniques, on peut vouloir retrouver un numéro de téléphone dans la collection des contacts à partir d'un nom. Un `dict` peut être vu comme une généralisation du type `list` où l'accès aux éléments est effectué à partir d'une clé qui n'est pas limitée à un indice entier comme c'est le cas de `list`. Plusieurs opérations internes de Python exploitent le type `dict`. Un `dict` est un ensemble de couples `(clé, valeur)`. L'implémentation est analogue avec celle du type `set` sous forme d'une table de hachage. La fonction de hachage est appliquée à la clé pour déterminer l'adresse dans la table de hachage où se trouve la valeur. La clé doit être d'un type hachable comme un `str`, un `int` ou un `float`. Plusieurs des opérations applicables à une liste sont aussi applicables à un `dict`. Comme pour `list`, un `dict` est muable.

Un littéral `dict` est encadré par des accolades et contient une suite de couples `clé :valeur` séparés par des virgules. L'exemple suivant exemple crée un répertoire téléphonique qui associe un numéro de téléphone (la valeur) à un nom (la clé) :

```
>>> repertoire_telephonique = {'Pierre' : '514-333-3333', 'Jean' : '514-
222-2222', 'Jacques' : '514-555-5555'}
>>> repertoire_telephonique
```

```
{'Pierre': '514-333-3333', 'Jean': '514-222-2222', 'Jacques': '514-555-
5555'}
```

Une valeur est sélectionnée par la syntaxe `nom_dict[cle]` :

```
>>> repertoire_telephonique['Jean']
'514-222-2222'
```

Si la clé est absente, une exception est levée :

```
>>> repertoire_telephonique['Paul']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Paul'
```

Le `in` permet de vérifier l'appartenance d'un élément au `dict` :

```
>>> 'Paul' in repertoire_telephonique
False
```

La méthode `keys()` retourne la liste des clés du `dict` sous forme d'une `view` :

```
>>> repertoire_telephonique.keys()
dict_keys(['Pierre', 'Jean', 'Jacques'])
```

La méthode `values()` retourne la liste des valeurs du dict sous forme d'une `view` :

```
>>> repertoire_telephonique.values()
dict_values(['514-333-3333', '514-222-2222', '514-555-5555'])
```

La fonction `len()` retourne la taille du `dict` :

```
>>> len(repertoire_telephonique)
3
```

Pour ajouter un nouveau couple `(clé,valeur)` à un `dict`, il suffit d'affecter la valeur à la nouvelle clé selon la syntaxe `nom_dict[cle] = valeur` :

```
>>> repertoire_telephonique['Paul']= '514-999-9999'
>>> repertoire_telephonique
{'Pierre': '514-333-3333', 'Jean': '514-222-2222', 'Jacques': '514-555-
5555', 'Paul': '514-999-9999'}
```

Si la clé existe déjà, l'affectation modifie la valeur correspondant à la clé :

```
>>> repertoire_telephonique['Jean']= '514-111-1111'
>>> repertoire_telephonique
{'Pierre': '514-333-3333', 'Jean': '514-111-1111', 'Jacques': '514-555-
5555', 'Paul': '514-999-9999'}
```

La syntaxe `del nom_dict[cle]` supprime le couple correspondant à la clé spécifiée :

```
>>> del repertoire_telephonique['Jean']
>>> repertoire_telephonique
{'Pierre': '514-333-3333', 'Jacques': '514-555-5555', 'Paul': '514-999-
9999'}
```

Le `for` permet d'itérer sur les clés du `dict` :

```
>>> for cle in repertoire_telephonique :
...     print(cle, ' : ', repertoire_telephonique[cle])
...
Pierre  :  514-333-3333
Jacques  :  514-555-5555
Paul  :  514-999-9999
>>>
```

4.9 Représentation du temps (time)

Les modules `time`, `datetime` et `calendar` de Python contiennent des fonctions sophistiquées pour la représentation du temps. Le module `time` est basé sur les fonctions du langage C de la plate-forme sous-jacente. La fonction `time()` du module `time` retourne le temps en secondes depuis l'*epoch* qui correspond au 1er janvier 1970 à 00 :00 :00 heures UTC (Temps Universel Coordonné) sous Windows et Unix, et ceci sous forme d'un nombre en point flottant. UTC est le temps de référence qui était autrefois appelé GMT (*Greenwich Mean Time*). :

```
>>> import time
>>> time.time()
1541431583.7022107
```

La fonction `gmtime()` représente le temps d'une manière plus usuelle sous forme d'un `struct_time` qui possède une interface de n-uplet nommé. Un n-uplet nommé permet d'accéder à ses éléments soit par indice ou par nom d'attribut. Le tableau suivant montre les éléments :

Indice	Attribut	Valeurs
0	tm_year	Année à quatre chiffres, 1993
1	tm_mon	Mois dans l'intervalle [1, 12]
2	tm_mday	Jour dans l'intervalle [1, 31]
3	tm_hour	Heure dans l'intervalle [0, 23]
4	tm_min	Minute dans l'intervalle [0, 59]
5	tm_sec	Seconde dans l'intervalle [0, 61]
6	tm_wday	Jour de semaine dans l'intervalle [0, 6]; 0 est lundi
7	tm_yday	Jour de l'année dans l'intervalle [1, 366]
8	tm_isdst	1 pour temps DST, 0 sinon -1 si indéterminé
	tm_zone	Zone de temps
	tm_gmtoff	Décalage par rapport au temps UTC

DST (*Daylight Saving Time*) représente le temps ajusté pour la zone de temps habituellement d'une heure pour profiter mieux de la lumière du jour une partie de l'année :

```
>>> tgmt = time.gmtime()
>>> tgmt
time.struct_time(tm_year=2018,    tm_mon=11,    tm_mday=5,    tm_hour=15,
tm_min=27, tm_sec=50, tm_wday=0, tm_yday=309, tm_isdst=0)
>>> tgmt[2]
5
>>> tgmt.tm_mday
5
>>> tgmt.tm_zone
'UTC'
>>> tgmt.tm_gmtoff
0
```

La fonction `localtime()` retourne le temps converti selon la zone de temps locale sous forme d'un `struct_time`.

```
>>> tlocal = time.localtime()
>>> tlocal
time.struct_time(tm_year=2018,    tm_mon=11,    tm_mday=5,    tm_hour=10,
tm_min=30, tm_sec=11, tm_wday=0, tm_yday=309, tm_isdst=0)
>>> tlocal.tm_zone
'Eastern Standard Time'
>>> tlocal.tm_gmtoff
-18000
>>>
```

Le module `datetime` permet d'effectuer des calculs de temps en incorporant le type `timedelta` qui représente une durée de temps avec les unités plus usuelles (seconde, heure, jour, mois, …).

5 Graphisme 2D et fonctions

Ce chapitre présente les principes de base des graphiques en deux dimensions (2D) et de la gestion d'évènements avec la bibliothèque `Pygame`. Il introduit aussi la création de fonction en la justifiant dans le contexte d'un programme simple de graphisme 2D.

5.1 Graphisme 2D avec Pygame

Dans un premier temps, cette section examine l'affichage graphique 2D à l'aide d'un exemple de programme qui dessine une figure très simple dans une fenêtre de la bibliothèque `Pygame`. Cette bibliothèque contient un ensemble de modules qui visent à faciliter le développement de jeux en Python. Dans ce premier exemple, le programme ne fait qu'afficher un graphique simple dans une fenêtre et n'exploite pas les mécanismes d'animation et de gestion des évènements qui sont étudiés plus loin. Pour utiliser la bibliothèque, il faut d'abord l'installer dans l'environnement de l'interprète Python.

`Pygame` nécessite Python ; si vous ne l'avez pas encore installé sur votre machine, vous devez maintenant le faire. La meilleure façon d'installer `pygame` est d'utiliser l'outil `pip`. Notez que cet outil est fourni avec les versions récentes de Python. Nous utilisons le drapeau `--user` pour lui demander d'installer dans le répertoire personnel, plutôt que globalement.

```
python -m pip install -U pygame --user
```

Certaines erreurs imprévues peuvent se produire lors de l'installation avec pip, mais elle sont généralement soluble. Par exemple, si vous rencontrez l'erreur « error: externally-managed-environment », modifiez la commande en ajoutant le drapeau --break-system-packages comme ceci :

```
python -m pip install -U pygame –user –break-system-packages
```

Nous verrons une utilisation plus sophistiquée de pip avec les environnements virtuels à la section 13.2.

Rappelez-vous que si vous êtes sur une machine où Python 2 est présent par défaut, il peut être nécessaire de remplacer la commande python par python3. Pour voir si cela fonctionne, exécutez l'un des exemples inclus :

```
python -m pygame.examples.aliens
```

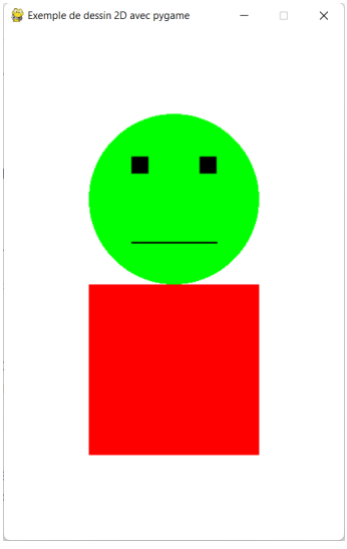
Si un jeu démarre, vous êtes prêt à partir ! Si ce n'est pas le cas, des instructions plus détaillées et spécifiques à la plate-forme vous concernant sont disponibles en ligne : https://www.pygame.org/wiki/GettingStarted

Le programme `ExempleDessin2DPygame` dessine un bonhomme simple (appelons-le `Bot`) dans une fenêtre produite avec `Pygame`.

CodePython/chapitre5/ ExempleDessin2DEtFermetureDeFenetre.py

```
"""
Exemple de dessin 2D avec pygame
"""
# Importer la librairie de pygame et initialiser
import sys, pygame
pygame.init()

LARGEUR_FENETRE = 400
HAUTEUR_FENETRE = 600
fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre

pygame.display.set_caption('Exemple de dessin 2D avec pygame') # Définir le titre dans le haut de la fenêtre

# Définir les couleurs employées dans le dessin
BLANC = (255,255,255)
ROUGE = (255,0,0)
NOIR = (0,0,0)
VERT = (0,255,0)

fenetre.fill(BLANC) # Dessiner le fond de la surface de dessin

pygame.draw.ellipse(fenetre, VERT, ((100,100),(200,200))) # Dessiner la tête
pygame.draw.rect(fenetre, NOIR, ((150,150),(20,20))) # L'oeil gauche
pygame.draw.rect(fenetre, NOIR, ((230,150),(20,20))) # L'oeil droit
pygame.draw.line(fenetre, NOIR, (150,250),(250,250),2) # La bouche
pygame.draw.rect(fenetre, ROUGE, ((100,300),(200,200))) # Le corps

pygame.display.flip() # Mettre à jour la fenêtre graphique

# Traiter la fermeture de la fenêtre
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT: # Vérifier si l'utilisateur a cliqué pour fermer la fenêtre
            pygame.quit() # Terminer pygame
            sys.exit()

```

La fenêtre suivante est affichée :



Pour fermer la fenêtre, il faut cliquer dans la zone de fermeture de fenêtre qui correspond au X dans le coin supérieur droit. Le dessin est effectué dans une fenêtre graphique qui correspond à un espace à deux dimensions (2D) illustré à la figure suivante. L'axe des *x* est l'axe horizontal et l'axe des *y*, le vertical. Par opposition à la convention mathématique usuelle, l'axe des *y* est orienté vers le bas. Les figures graphiques produites avec les appels à la bibliothèque `Pygame` font référence aux coordonnées de cet espace. Lorsque le graphique est représenté à l'écran, chacune des coordonnées correspond à un pixel (*picture element*). Chacun des pixels de l'écran prend une couleur particulière définie selon un système, habituellement le RVB (RGB).

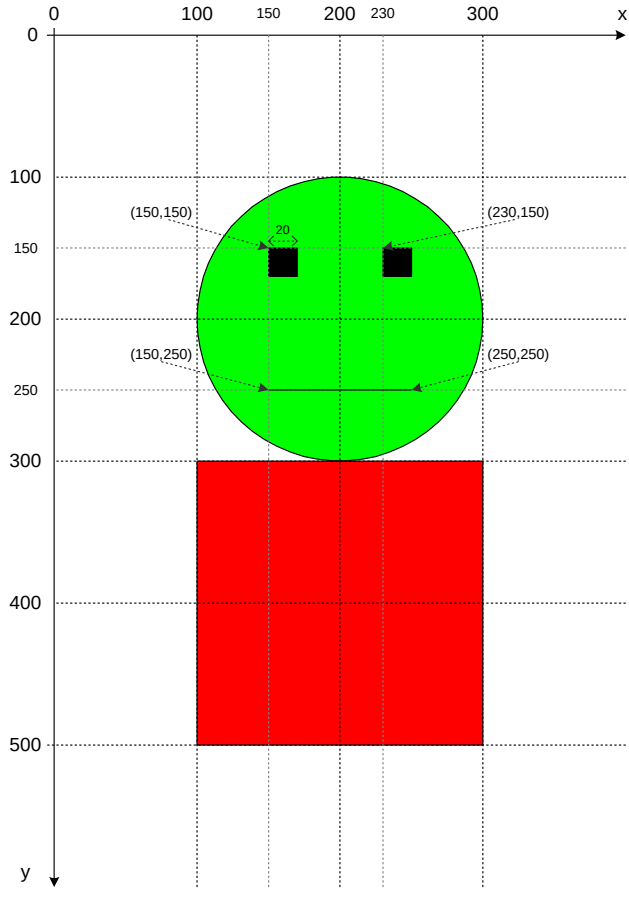


Figure 8. Coordonnées du bonhomme *Bot*.

Dans le code de l'exemple, le `import pygame` importe les modules de la bibliothèque `Pygame` et l'appel à `pygame.init()` initialise les modules de la bibliothèque.

```
import pygame
pygame.init()
```

Les constantes `LARGEUR_FENETRE` et `HAUTEUR_FENETRE` représentent la largeur et la hauteur de la fenêtre en nombre de pixels.

Constantes en Python

Par convention les noms de variables qui représentent des constantes sont en majuscules. Cependant, ce ne sont pas de vrais constantes et la valeur de

ces variables peut être modifiée. La version 3.4 de Python introduit le concept d'énumération (`enum`) qui permet de définir des constantes.

L'appel à `pygame.display.set-mode()` crée et ouvre une fenêtre pour l'affichage des graphiques, selon les dimensions passées en arguments.

```
LARGEUR_FENETRE = 400
HAUTEUR_FENETRE = 600
fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre
```

L'appel à la méthode `pygame.display.set_caption()` définit un titre qui est affiché dans le haut de la fenêtre graphique.

```
pygame.display.set_caption('Exemple de dessin 2D avec pygame')
```

Les lignes suivantes définissent des constantes pour les couleurs employées dans le dessin. Chacune des couleurs est un n-uplet de trois entiers entre 0 et 255 qui correspondent aux trois couleurs, rouge (*Red*), vert (*Green*) et bleu (*Bleu*), du système RVB. Il y a aussi un quatrième argument optionnel, l'opacité, qui n'est pas employé ici. Chacun des entiers définit l'intensité lumineuse de la couleur correspondante. Le 0 est l'absence de couleur et 255 est l'intensité maximale. Le blanc est la combinaison des trois couleurs dans un tel système. Le noir est l'absence de couleur.

```
BLANC = (255,255,255)
ROUGE = (255,0,0)
NOIR = (0,0,0)
VERT = (0,255,0)
```

Pour déterminer les valeurs RVB d'une couleur particulière, il est possible d'employer un outil tel que :

http://www.colorpicker.com/

L'appel suivant remplit la surface de dessin avec la couleur blanche.

```
fenetre.fill(BLANC) # Dessiner le fond de la surface de dessin
```

Ensuite, une série d'appels à des fonctions du module `pygame.draw` dessinent les diverses parties du bonhomme. L'appel à la fonction `pygame.draw.ellipse()` dessine la tête de couleur verte. Cette fonction produit une ellipse sur la surface graphique `fenetre`. Le deuxième argument VERT est la couleur de l'ellipse et le troisième argument `((100, 100), (200, 200))` spécifie la position et la taille de l'ellipse par les coordonnées d'un rectangle qui englobe l'ellipse. Le rectangle englobant est défini par le couple formé de la position, `(x, y)` du coin supérieur gauche, et de la taille `(largeur,hauteur)` de l'ellipse. Dans notre exemple, la

largeur est la même que la hauteur (200), ce qui produit un cercle de diamètre 200.

```
pygame.draw.ellipse(fenetre, VERT, ((100,100),(200,200))) # Dessiner la tête
```

La fonction `pygame.draw.rect()` est appelée à deux reprises pour dessiner les deux yeux. Les arguments ont la même signification que pour `pygame.draw.ellipse()`.

```
pygame.draw.rect(fenetre, NOIR, ((150,150),(20,20))) # L'oeil gauche
pygame.draw.rect(fenetre, NOIR, ((230,150),(20,20))) # L'oeil droit
```

La fonction `pygame.draw.line()` dessine une ligne qui représente la bouche. Les deux premiers arguments ont la même signification que pour `pygame.draw.rect()`. Le troisième argument est la coordonnée `(x,y)` du début de la ligne et le quatrième argument, la fin. Le dernier argument, le cinquième, désigne l'épaisseur de la ligne.

```
pygame.draw.line(fenetre, NOIR, (150,250),(250,250),2) # La bouche
```

Le dernier appel à `pygame.draw.rect()` produit le corps du bonhomme.

```
pygame.draw.rect(fenetre, ROUGE, ((100,300),(200,200))) # Le corps
```

C'est l'appel à la fonction `pygame.display.flip()` qui met effectivement à jour l'écran avec le nouveau contenu. Le mécanisme exact de mise à jour dépend de l'environnement graphique employé.

```
pygame.display.flip() # Mettre à jour la fenêtre graphique
```

Le reste du code permet de fermer la fenêtre lorsque l'utilisateur clique dans la zone de fermeture. Ceci nécessite une boucle de gestion des événements. Cet aspect est détaillé à la section 5.3.

```
# Traiter la fermeture de la fenêtre
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT: # Vérifier si l'utilisateur a cliqué
pour fermer la fenêtre
            pygame.quit() # Terminer pygame
            sys.exit()
```

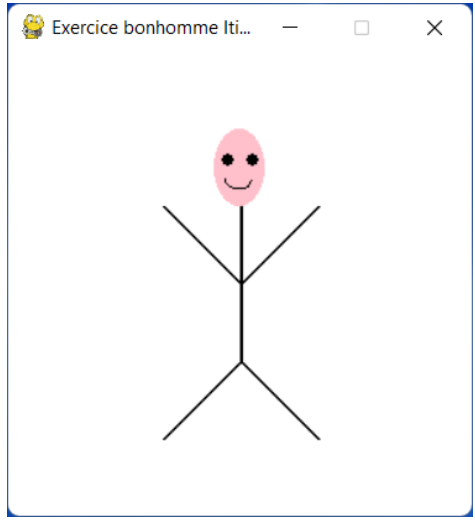
La classe `pygame.Color` facilite la manipulation des couleurs. Plusieurs noms de couleur prédéfinis peuvent être employés. L'exemple suivant reprend l'exemple précédent en employant des couleurs prédéfinies.

CodePython/chapitre5/ExempleDessin2DClasseColor.py

```
"""
Exemple de dessin 2D avec pygame qui emploie la classe pygame.Color
"""
```

```
# Importer la librairie de pygame et initialiser
import sys, pygame
from pygame import Color
pygame.init()

LARGEUR_FENETRE = 400
HAUTEUR_FENETRE = 600
fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre

pygame.display.set_caption('Exemple de dessin 2D avec pygame') # Définir le titre dans le haut de la fenêtre

fenetre.fill(Color('white')) # Dessiner le fond de la surface de dessin

pygame.draw.ellipse(fenetre, Color('green'), ((100,100),(200,200))) # Dessiner la tête
pygame.draw.rect(fenetre, Color('black'), ((150,150),(20,20))) # L'oeil gauche
pygame.draw.rect(fenetre, Color('black'), ((230,150),(20,20))) # L'oeil droit
pygame.draw.line(fenetre, Color('black'), (150,250),(250,250),2) # La bouche
pygame.draw.rect(fenetre, Color('red'), ((100,300),(200,200))) # Le corps

pygame.display.flip() # Mettre à jour la fenêtre graphique

# Traiter la fermeture de la fenêtre
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT: # Vérifier si l'utilisateur a cliqué pour fermer la fenêtre
            pygame.quit() # Terminer pygame
            sys.exit()
```

Exercice. En vous inspirant de l'exemple précédent, écrivez un programme qui dessine un bonhomme de votre cru ou encore le bonhomme **Iti** suivant. La taille de la fenêtre de **Iti** est 300 par 300.



Solution. CodePython/chapitre5/ExerciceDessinIti.py

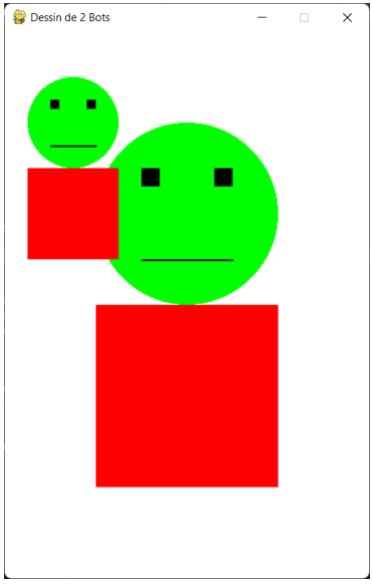
```
"""
Exercice : bonhomme Iti
"""
# Importer la bibliothèque de pygame et initialiser
import sys, pygame
from pygame import Color
pygame.init()

```

```
 9. LARGEUR_FENETRE = 300
10. HAUTEUR_FENETRE = 300
11. fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre
12.
13. pygame.display.set_caption('Exercice bonhomme Iti avec pygame') # Définir le titre dans le
haut de la fenêtre
14.
15. fenetre.fill(Color('white')) # Dessiner le fond de la surface de dessin
16.
17. pygame.draw.ellipse(fenetre, Color('pink'), ((133, 50), (33, 50))) # Dessiner la tête
18. pygame.draw.arc(fenetre, Color('black'),((140,75),(19,15)),3.1416,0,1) # Le sourire
19. pygame.draw.ellipse(fenetre, Color('black'), ((138,66),(8,8))) # L'oeil gauche
20. pygame.draw.ellipse(fenetre, Color('black'), ((154,66),(8,8))) # L'oeil droit
21. pygame.draw.line(fenetre, Color('black'), (150,100), (150,200), 2) # Le corps
22. pygame.draw.line(fenetre, Color('black'), (100,100), (150,150), 2) # Bras gauche
23. pygame.draw.line(fenetre, Color('black'), (200,100), (150,150), 2) # Bras droit
24. pygame.draw.line(fenetre, Color('black'), (100,250), (150,200), 2) # Jambe gauche
25. pygame.draw.line(fenetre, Color('black'), (200,250), (150,200), 2) # Jambe droite
26.
27. pygame.display.flip() # Mettre à jour la fenêtre graphique
28.
29. # Traiter la fermeture de la fenêtre
30. while True:
31.     for event in pygame.event.get():
32.         if event.type == pygame.QUIT: # Vérifier si l'utilisateur a cliqué pour fermer la
fenêtre
33.             pygame.quit() # Terminer pygame
34.             sys.exit()
```

5.2 Simplification du programme par une fonction avec paramètres

Cette section introduit la création d'une fonction en Python en passant par un exemple qui montre comment l'utilisation d'une fonction avec paramètres peut grandement simplifier un programme et ceci dans le contexte du dessin 2D. La notion de passage des arguments est étudiée en même temps. L'exercice suivant permet de motiver l'utilisation d'une fonction avec paramètres.

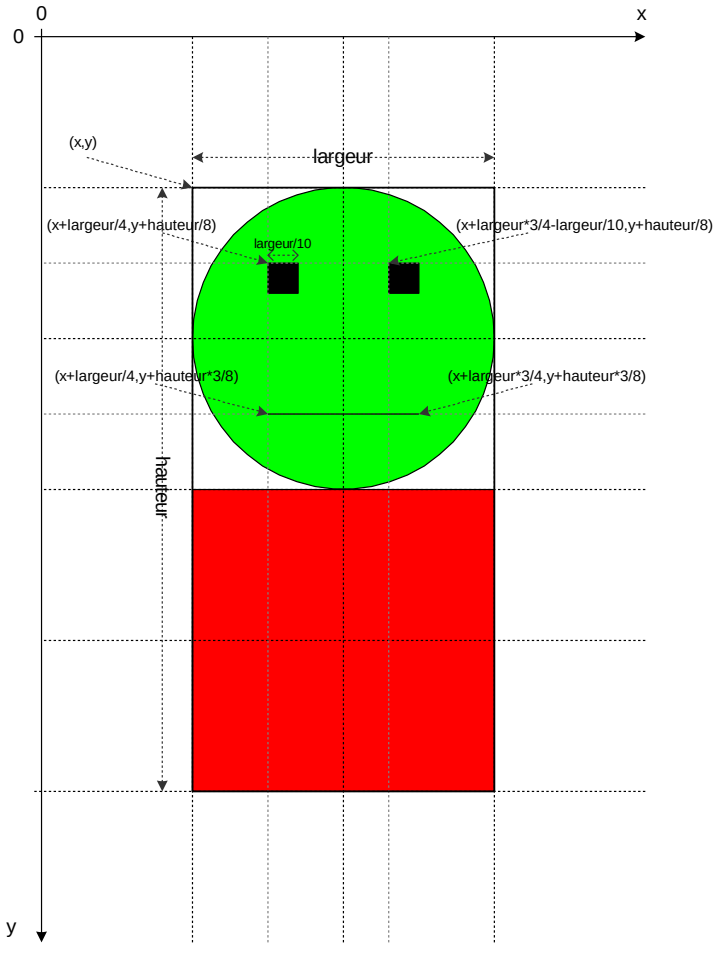
Exercice. Dessiner deux bonhommes de taille et position différentes tel qu'illustré par la figure suivante :



Solution. [CodePython](#)/chapitre5/ExerciceDessin2Bots.py

```
"""
Exercice : dessin de 2 bots
"""
# Importer la bibliothèque de pygame et initialiser
import sys, pygame
from pygame import Color
pygame.init()

LARGEUR_FENETRE = 400
HAUTEUR_FENETRE = 600
fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre

pygame.display.set_caption('Dessin de 2 Bots') # Définir le titre dans le haut de la fenêtre

fenetre.fill(Color('white')) # Dessiner le fond de la surface de dessin
# Le premier Bot
pygame.draw.ellipse(fenetre, Color('green'), ((100, 100),(200, 200))) # Dessiner la tête
pygame.draw.rect(fenetre, Color('black'), ((150, 150),(20, 20))) # L'oeil gauche
pygame.draw.rect(fenetre, Color('black'), ((230, 150),(20, 20))) # L'oeil droit
pygame.draw.line(fenetre, Color('black'), (150,250),(250,250), 2) # La bouche
pygame.draw.rect(fenetre, Color('red'), ((100, 300),(200, 200))) # Le corps

# Le deuxième Bot
pygame.draw.ellipse(fenetre, Color('green'), ((25,50),(100,100))) # Dessiner la tête
pygame.draw.rect(fenetre, Color('black'), ((50, 75),(10, 10))) # L'oeil gauche
pygame.draw.rect(fenetre, Color('black'), ((90,75),(10,10))) # L'oeil droit
pygame.draw.line(fenetre, Color('black'), (50,125),(100,125), 2) # La bouche
pygame.draw.rect(fenetre, Color('red'), ((25,150),(100,100))) # Le corps

pygame.display.flip() # Mettre à jour la fenêtre graphique

# Traiter la fermeture de la fenêtre
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT: # Vérifier si l'utilisateur a cliqué pour fermer la fenêtre
            pygame.quit() # Terminer pygame
            sys.exit()
```

La solution précédente rappelle les mêmes fonctions deux fois pour dessiner les deux `Bot`. Pour la position et la taille du `Bot`, il faut calculer de nouvelles valeurs des arguments à chaque fois. Une manière plus élégante de traiter ce problème consiste à chercher une solution plus générale au dessin d'un `Bot` où sa position et sa taille sont variables. Une technique souvent employée en graphisme 2D consiste à définir un *rectangle englobant* à l'intérieur duquel est dessiné la figure tel que vu précédemment pour le dessin d'une ellipse avec `pygame.draw.ellipse()`. Ainsi, le `Bot` est dessiné à l'échelle à l'intérieur du rectangle englobant. Comme pour `pygame.draw.ellipse()`, quatre paramètres sont définis pour représenter le rectangle englobant : les coordonnées `x` et `y` du coin supérieur gauche, la `largeur` et la `hauteur` du rectangle englobant. La figure suivante montre les dimensions des différentes parties du `Bot` relativement à ces paramètres.



La version suivante dessine le même `Bot` que notre premier exemple mais en utilisant des variables qui représentent le rectangle englobant. Les paramètres des figures correspondent aux valeurs de la figure précédente.

CodePython/chapitre5/ExempleBotRectangleEnglobant.py

```
"""
Exemple de dessin du Bot dans un rectangle englobant
"""
# Importer la bibliothèque de pygame et initialiser
import sys,pygame
from pygame import Color
pygame.init()

LARGEUR_FENETRE = 400
```

```
10. HAUTEUR_FENETRE = 600
11. fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre
12.
13. pygame.display.set_caption('Exemple de dessin du Bot dans un rectangle englobant') #
Définir le titre dans le haut de la fenêtre
14.
15. fenetre.fill(Color('white')) # Dessiner le fond de la surface de dessin
16.
17. r = pygame.Rect((100,100),(200,400)) # le rectangle englobant
18.
19. # Dessiner le Bot relativement au rectangle englobant r
20. pygame.draw.ellipse(fenetre, Color('green'), ((r.x,r.y),(r.width, r.height/2))) # Dessiner
la tête
21. pygame.draw.rect(fenetre, Color('black'),
((r.x+r.width/4,r.y+r.height/8),(r.width/10,r.height/20))) # L'oeil gauche
22. pygame.draw.rect(fenetre, Color('black'), ((r.x+r.width*3/4-
r.width/10,r.y+r.height/8),(r.width/10,r.height/20))) # L'oeil droit
23. pygame.draw.line(fenetre, Color('black'),
(r.x+r.width/4,r.y+r.height*3/8),(r.x+r.width*3/4,r.y+r.height*3/8), 2) # La bouche
24. pygame.draw.rect(fenetre, Color('red'), ((r.x,r.y+r.height/2),(r.width,r.height/2))) # Le
corps
25.
26. pygame.display.flip() # Mettre à jour la fenêtre graphique
27.
28. # Traiter la fermeture de la fenêtre
29. while True:
30.     for event in pygame.event.get():
31.         if event.type == pygame.QUIT: # Vérifier si l'utilisateur a cliqué pour fermer la
fenêtre
32.             pygame.quit() # Terminer pygame
33.             sys.exit()
```

On peut maintenant facilement redessiner le **Bot** deux fois en changeant la position et la taille par la modification des valeurs de **x, y, largeur et hauteur** mais en répétant exactement les mêmes instructions deux fois.

CodePython/chapitre5/Exemple2BotsRectangleEnglobant.py

```
1. """
2. Exemple de dessin de 2 Bots dans un rectangle englobant
3. """
4. # Importer la bibliothèque de pygame et initialiser
5. import sys,pygame
6. from pygame import Color
7. pygame.init()
8.
9. LARGEUR_FENETRE = 400
10. HAUTEUR_FENETRE = 600
11. fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre
12.
13. pygame.display.set_caption('Exemple de dessin du Bot dans un rectangle englobant') #
Définir le titre dans le haut de la fenêtre
14.
15. fenetre.fill(Color('white')) # Dessiner le fond de la surface de dessin
16.
17. # Rectangle englobant du premier Bot
18. r = pygame.Rect((100,100),(200,400)) # le rectangle englobant
19.
20. # Dessiner le Bot relativement au rectangle englobant r
21. pygame.draw.ellipse(fenetre, Color('green'), ((r.x,r.y),(r.width, r.height/2))) # Dessiner
la tête
22. pygame.draw.rect(fenetre, Color('black'),
((r.x+r.width/4,r.y+r.height/8),(r.width/10,r.height/20))) # L'oeil gauche
23. pygame.draw.rect(fenetre, Color('black'), ((r.x+r.width*3/4-
r.width/10,r.y+r.height/8),(r.width/10,r.height/20))) # L'oeil droit
24. pygame.draw.line(fenetre, Color('black'),
(r.x+r.width/4,r.y+r.height*3/8),(r.x+r.width*3/4,r.y+r.height*3/8), 2) # La bouche
25. pygame.draw.rect(fenetre, Color('red'), ((r.x,r.y+r.height/2),(r.width,r.height/2))) # Le
corps
26.
27. # Rectangle englobant pour le deuxième Bot
```

```
28. r = pygame.Rect((25,50),(100,200)) # le rectangle englobant
29.
30. # Dessiner le Bot relativement au rectangle englobant r
31. pygame.draw.ellipse(fenetre, Color('green'), ((r.x,r.y),(r.width, r.height/2))) # Dessiner
la tête
32. pygame.draw.rect(fenetre, Color('black'),
((r.x+r.width/4,r.y+r.height/8),(r.width/10,r.height/20))) # L'oeil gauche
33. pygame.draw.rect(fenetre, Color('black'), ((r.x+r.width*3/4-
r.width/10,r.y+r.height/8),(r.width/10,r.height/20))) # L'oeil droit
34. pygame.draw.line(fenetre, Color('black'),
(r.x+r.width/4,r.y+r.height*3/8),(r.x+r.width*3/4,r.y+r.height*3/8), 2) # La bouche
35. pygame.draw.rect(fenetre, Color('red'), ((r.x,r.y+r.height/2),(r.width,r.height/2))) # Le
corps
36.
37. pygame.display.flip() # Mettre à jour la fenêtre graphique
38.
39. # Traiter la fermeture de la fenêtre
40. while True:
41.     for event in pygame.event.get():
42.         if event.type == pygame.QUIT: # Vérifier si l'utilisateur a cliqué pour fermer la
fenêtre
43.             pygame.quit() # Terminer pygame
44.             sys.exit()
45.
```

5.2.1 Création d'une fonction

La solution de l'exemple précédent oblige de répéter deux fois les mêmes énoncés. Cette répétition peut être évitée en les regroupant dans une fonction et en l'appelant à deux reprises. Jusqu'à présent, les fonctions appelées étaient prédéfinies. Voyons maintenant comment créer une nouvelle fonction.

Dans l'exemple suivant la fonction `dessiner_bot()` est créée. Elle regroupe les énoncés de dessin du `Bot`. La fonction est ensuite appelée à deux reprises pour dessiner les deux `Bots`.

CodePython/chapitre5/ExempleFonctionDessinerBot.py

```
 1. """
 2. Exemple de fonction dessiner_bot
 3. """
 4.
 5. import sys,pygame
 6. from pygame import Color
 7.
 8. def dessiner_bot(fenetre,r):
 9.     """ Dessiner un Bot.
10.
11.     fenetre : la surface de dessin
12.     r : rectangle englobant de type pygame.Rect
13.     """
14.
15.     # Dessiner le Bot relativement au rectangle englobant r
16.     pygame.draw.ellipse(fenetre, Color('green'), ((r.x,r.y),(r.width, r.height/2))) #
Dessiner la tête
17.     pygame.draw.rect(fenetre, Color('black'),
((r.x+r.width/4,r.y+r.height/8),(r.width/10,r.height/20))) # L'oeil gauche
18.     pygame.draw.rect(fenetre, Color('black'), ((r.x+r.width*3/4-
r.width/10,r.y+r.height/8),(r.width/10,r.height/20))) # L'oeil droit
19.     pygame.draw.line(fenetre, Color('black'),
(r.x+r.width/4,r.y+r.height*3/8),(r.x+r.width*3/4,r.y+r.height*3/8), 2) # La bouche
20.     pygame.draw.rect(fenetre, Color('red'), ((r.x,r.y+r.height/2),(r.width,r.height/2))) #
Le corps
21.
22. pygame.init() # Initialiser Pygame
23. LARGEUR_FENETRE = 400
```

```
24. HAUTEUR_FENETRE = 600
25. fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre
26. pygame.display.set_caption('Exemple de dessin du Bot dans un rectangle englobant') #
Définir le titre dans le haut de la fenêtre
27.
28. BLANC = (255,255,255)
29. fenetre.fill(Color('white')) # Dessiner le fond de la surface de dessin
30.
31. # Dessiner deux Bots en appelant la fonction à deux reprises
32. dessiner_bot(fenetre,pygame.Rect((100,100),(200,400)))
33. dessiner_bot(fenetre,pygame.Rect((25,50),(100,200)))
34.
35. pygame.display.flip() # Mettre à jour la fenêtre graphique
36.
37. # Traiter la fermeture de la fenêtre
38. while True:
39.     for event in pygame.event.get():
40.         if event.type == pygame.QUIT: # Vérifier si l'utilisateur a cliqué pour fermer la
fenêtre
41.             pygame.quit() # Terminer pygame
42.             sys.exit()
```

Lorsqu'un script est appelé, le code où l'exécution démarre est le *programme principal*. Dans notre exemple, c'est d'abord l'instruction suivante qui est exécutée :

```
import sys, pygame
```

Ensuite la définition de la fonction est débutée par l'entête suivante :

```
def dessiner_bot(fenetre,r):
```

Lorsque la définition de fonction est rencontrée, le code de la fonction en lui-même n'est pas exécuté à ce moment. Ce n'est que la définition de la fonction qui est mémorisée par l'interprète Python. Par la suite, le code est exécuté lorsque la fonction est appelée explicitement.

L'entête de la fonction définit la *signature* de la fonction `dessiner_bot()` :

Signature d'une fonction

La signature d'une fonction précise :
- le nom de la fonction
- les paramètres

La signature d'une fonction permet de déterminer comment appeler la fonction.

Le programmeur décide des noms et du nombre des paramètres. Lorsqu'on appelle la fonction, il faut spécifier les arguments qui doivent correspondre aux paramètres. Le terme *paramètre formel* est aussi employée pour distinguer les paramètres de la définition de la fonction des *paramètres réels* ou arguments employés lors de l'appel de la fonction.

Le corps de la fonction suit l'entête et doit être indenté par rapport à l'entête. Dans le corps de la fonction, les énoncés font référence aux noms

des paramètres comme s'ils étaient des variables. Par exemple, dans la ligne suivante, les variables `fenetre` et `r` sont les paramètres de la fonction :

```
pygame.draw.ellipse(fenetre, VERT, ((r.x,r.y),(r.width, r.height/2))) # Dessiner la tête
```

Les valeurs de ces paramètres (arguments) sont passées à l'appel de la fonction. Pour dessiner les deux `Bot`, la fonction `dessiner_bot()` est appelée deux fois dans le programme principal en précisant les arguments. Le premier appel est le suivant :

```
dessiner_bot(fenetre,pygame.Rect((100,100),(200,400)))
```

Les valeurs `fenetre` et `pygame.Rect((100,100),(200,400))` sont les arguments. Lors de l'appel de la fonction, on peut imaginer que la valeur de l'argument est affectée au paramètre correspondant tel que le montre le code suivant :

```
fenetre (paramètre) = fenetre (argument)
r = pygame.Rect((100,100),(200,400))
```

Ensuite, la fonction est exécutée avec les paramètres qui apparaissent à l'intérieur de la fonction comme des variables. La première affectation ci-haut signifie que la valeur de la variable `fenetre` qui est l'argument est affectée au paramètre du même nom. Mais il faut bien comprendre qu'il s'agit de deux éléments différents du programme qui portent le même nom. Dans le code de la fonction, ce qui est visible est le paramètre `fenetre` et la variable `fenetre` n'est pas visible. Elle est masquée par le paramètre de même nom. Ainsi, dans le programme, le même nom peut être employé avec des rôles différents selon l'endroit où se trouve l'utilisation de ce nom. Pour éliminer toute ambiguïté, des règles précises sont définies au sujet de la *portée* des noms.

Pour bien illustrer le fait que la variable `fenetre` introduite dans le programme au niveau global est différente du paramètre défini localement à la fonction, l'exemple suivant montre qu'il aurait été possible d'employer un nom différent dans la déclaration du paramètre. Le même effet est obtenu que pour l'exemple précédent.

Le paramètre qui représente la fenêtre de dessin s'appelle *f* dans cet exemple. L'appel est fait avec l'argument `fenetre` dont la valeur sera affectée au paramètre `f`.

CodePython/chapitre5/ExempleFonctionNomParametreDifferent.py

```
"""
Exemple de fonction dessiner_bot
Nom de paramètre et d'argument différents
"""

import sys, pygame
from pygame import Color

def dessiner_bot(f,r):
    """ Dessiner un Bot.

    fenetre : la surface de dessin
    r : rectangle englobant de type pygame.Rect
    """

    # Dessiner le Bot relativement au rectangle englobant r
    pygame.draw.ellipse(f, Color('green'), ((r.x,r.y),(r.width, r.height/2))) # Dessiner la tête
    pygame.draw.rect(f, Color('black'), ((r.x+r.width/4,r.y+r.height/8),(r.width/10,r.height/20))) # L'oeil gauche
    pygame.draw.rect(f, Color('black'), ((r.x+r.width*3/4-r.width/10, r.y+r.height/8), (r.width/10,r.height/20))) # L'oeil droit
    pygame.draw.line(f, Color('black'), (r.x+r.width/4,r.y+r.height*3/8), (r.x+r.width*3/4, r.y+r.height*3/8), 2) # La bouche
    pygame.draw.rect(f, Color('red'), ((r.x,r.y+r.height/2), (r.width,r.height/2))) # Le corps

pygame.init() # Initialiser Pygame
LARGEUR_FENETRE = 400
HAUTEUR_FENETRE = 600
fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre
pygame.display.set_caption('Exemple de dessin du Bot dans un rectangle englobant') # Définir le titre dans le haut de la fenêtre

BLANC = (255,255,255)
fenetre.fill(BLANC) # Dessiner le fond de la surface de dessin

# Dessiner deux Bots en appelant la fonction à deux reprises
dessiner_bot(fenetre, pygame.Rect((100,100),(200,400)))
dessiner_bot(fenetre, pygame.Rect((25,50),(100,200)))

pygame.display.flip() # Mettre à jour la fenêtre graphique

# Traiter la fermeture de la fenêtre
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT: # Vérifier si l'utilisateur a cliqué pour fermer la fenêtre
            pygame.quit() # Terminer pygame
            sys.exit()
```

Portée d'un nom, nom local ou global

Lorsqu'un nom apparaît dans une fonction, l'interprète vérifie si ce nom est local à la fonction. Le nom est local s'il a été défini à l'intérieur de la fonction soit comme variable créée localement, soit comme paramètre. La portée du nom est la fonction où il a été défini. Si le nom n'est pas local, la fonction peut alors accéder au nom qui est dans une portée plus globale. Ainsi le nom devient une variable globale par rapport à la fonction.

Les règles exactes qui régissent la portée des variables peuvent être complexes et varient d'un langage à l'autre. Pour limiter les ambiguïtés, il

est possible de choisir des noms différents pour désigner des variables qui ont des rôles différents !

Dans l'exemple suivant, le nom `fenetre` n'est pas un paramètre de la fonction. Lorsque la fonction fait référence à `fenetre` dans le corps de la fonction, c'est la variable globale `fenetre` qui est effectivement employée.

CodePython/chapitre5/ExempleFonctionAccesVariableGlobale.py

```
"""
Exemple de fonction dessiner_bot
"""

import sys, pygame
from pygame import Color

def dessiner_bot(r):
    """ Dessiner un Bot.

    r : rectangle englobant de type pygame.Rect
    """

    # Dessiner le Bot relativement au rectangle englobant r
    pygame.draw.ellipse(fenetre, Color('green'), ((r.x,r.y),(r.width, r.height/2))) #
Dessiner la tête
    pygame.draw.rect(fenetre, Color('black'),
((r.x+r.width/4,r.y+r.height/8),(r.width/10,r.height/20))) # L'oeil gauche
    pygame.draw.rect(fenetre, Color('black'), ((r.x+r.width*3/4-
r.width/10,r.y+r.height/8),(r.width/10,r.height/20))) # L'oeil droit
    pygame.draw.line(fenetre, Color('black'),
(r.x+r.width/4,r.y+r.height*3/8),(r.x+r.width*3/4,r.y+r.height*3/8), 2) # La bouche
    pygame.draw.rect(fenetre, Color('red'), ((r.x,r.y+r.height/2),(r.width,r.height/2))) #
Le corps

pygame.init() # Initialiser Pygame
LARGEUR_FENETRE = 400
HAUTEUR_FENETRE = 600
fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre
pygame.display.set_caption('Exemple de dessin du Bot dans un rectangle englobant') #
Définir le titre dans le haut de la fenêtre

fenetre.fill(Color('white')) # Dessiner le fond de la surface de dessin

# Dessiner deux Bots en appelant la fonction à deux reprises
dessiner_bot(pygame.Rect((100,100),(200,400)))
dessiner_bot(pygame.Rect((25,50),(100,200)))

pygame.display.flip() # Mettre à jour la fenêtre graphique

# Traiter la fermeture de la fenêtre
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT: # Vérifier si l'utilisateur a cliqué pour fermer la
fenêtre
            pygame.quit() # Terminer pygame
            sys.exit()
```

L'accès à une variable globale à l'intérieur d'une fonction est une pratique qui est souvent déconseillée parce qu'en examinant le code de la fonction, il est difficile de retrouver d'où vient la variable. Ceci crée une dépendance indirecte entre la fonction et le code qui l'appelle. Pour faciliter le développement des programmes, il est important de viser à limiter ce genre de dépendance. Lorsqu'on appelle une fonction, il est préférable que la

fonction ne touche que les paramètres afin de rendre le couplage entre le code appelant et la fonction plus transparent.

Le mot-clé global en Python est utilisé pour déclarer une variable à l'extérieur de la portée actuelle. Il permet à une fonction de modifier une variable qui est définie en dehors de la fonction.

Voici les règles de base pour utiliser le mot-clé global en Python:

- Si une variable est affectée à une valeur n'importe où dans le corps de la fonction, elle est considérée comme locale à moins qu'elle ne soit explicitement déclarée comme globale.

- Les variables qui ne sont référencées que dans une fonction sont implicitement globales.

- Nous utilisons le mot-clé global pour utiliser une variable globale à l'intérieur d'une fonction.

Voici un exemple de code Python qui utilise le mot-clé global:

```
1. x = 15
2.
3. def change():
4.     global x
5.     x = x + 5
6.
7. change()
8. print(x)
```

Dans cet exemple, nous avons défini une variable x en dehors de la fonction `change()`. À l'intérieur de la fonction `change()`, nous avons déclaré x comme étant globale en utilisant le mot-clé global. Nous avons ensuite incrémenté la valeur de `x` de 5. Enfin, nous avons appelé la fonction `change()` et imprimé la valeur de `x`. La sortie de ce programme sera 20.

Espace de nom (*namespace*)

Python réalise la recherche des noms avec le concept d'espace de nom. Un espace de nom est concrètement un ensemble d'associations entre un nom et un élément associé qui est activé dans un contexte particulier du code. Lorsqu'une fonction est appelée, Python lui associe un espace de nom local à la fonction. À la fin de l'exécution de la fonction, l'espace de nom n'est plus accessible.

Chacun des modules a aussi un espace de nom distinct. Ceci permet d'employer le même nom dans différents modules pour désigner des choses différentes.

Exercice. Reprenez l'exemple de fonction avec votre bonhomme (ou le `Iti`). Définissez une méthode `dessiner_bonhomme()` qui dessine votre bonhomme avec les mêmes paramètres que `dessiner_bot()`. Appelez la fonction à quelques reprises avec des valeurs différentes pour les paramètres.

Solution avec `Iti` :

CodePython/chapitre5/ExerciceFonctionDessinerIti.py

```
"""
Exercice: fonction dessiner_iti
"""

import sys, pygame
from pygame import Color

def dessiner_iti(fenetre,r):
    """ Dessiner un Iti.

    fenetre : la surface de dessin
    r : rectangle englobant de type pygame.Rect
    """

    # Coordonnées du milieu du rectangle englobant pour faciliter les calculs
    milieux = r.x + r.width/2;
    milieuy = r.y + r.height/2;

    pygame.draw.ellipse(fenetre, Color('pink'),
((r.x+r.width/3,r.y),(r.width/3,r.height/4))) # Dessiner la tête
    pygame.draw.arc(fenetre, Color('black'),((milieux-
r.width/12,r.y+r.height/8),(r.width/6,r.height/14)),3.1416,0,2) # Le sourire
    pygame.draw.ellipse(fenetre, Color('black'), ((milieux-
r.width/8,r.y+r.height/12),(r.width/12,r.height/24))) # L'oeil gauche
    pygame.draw.ellipse(fenetre, Color('black'), ((milieux+r.width/8-
r.width/12,r.y+r.height/12),(r.width/12,r.height/24))) # L'oeil droit
    pygame.draw.line(fenetre, Color('black'),
(milieux,r.y+r.height/4),(milieux,r.y+r.height*3/4), 2) # Le corps
    pygame.draw.line(fenetre, Color('black'), (r.x,r.y+r.height/4),(milieux,milieuy), 2) #
Bras gauche
    pygame.draw.line(fenetre, Color('black'),
(r.x+r.width,r.y+r.height/4),(milieux,milieuy), 2) # Bras droit
    pygame.draw.line(fenetre, Color('black'),
(r.x,r.y+r.height),(milieux,r.y+r.height*3/4), 2) # Jambe gauche
    pygame.draw.line(fenetre, Color('black'),
(r.x+r.width,r.y+r.height),(milieux,r.y+r.height*3/4), 2) # Jambe droite

pygame.init() # Initialiser Pygame
LARGEUR_FENETRE = 400
HAUTEUR_FENETRE = 600
fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre
pygame.display.set_caption('Exemple de dessin du Iti dans un rectangle englobant') #
Définir le titre dans le haut de la fenêtre

fenetre.fill(Color('white')) # Dessiner le fond de la surface de dessin

# Dessiner deux Iti en appelant la fonction à deux reprises
dessiner_iti(fenetre,pygame.Rect((100,100),(30,60)))
dessiner_iti(fenetre,pygame.Rect((25,50),(100,200)))

pygame.display.flip() # Mettre à jour la fenêtre graphique
```

```
42.
43. # Traiter la fermeture de la fenêtre
44. while True:
45.     for event in pygame.event.get():
46.         if event.type == pygame.QUIT: # Vérifier si l'utilisateur a cliqué pour fermer la
fenêtre
47.             pygame.quit() # Terminer pygame
48.             sys.exit()
```

Exercice. Reprenez l'exemple de code précédent en changeant le nom des paramètres.

5.2.2 Documentation d'une fonction et abstraction

Lorsqu'on appelle une fonction, il est important de bien comprendre les effets de la fonction sans avoir à en examiner le code en entier. A cet effet, il est primordial de documenter le rôle joué par la fonction en spécifiant ce que la fonction utilise (*input*) et ce que la fonction produit comme effet (*output*). La fonction devient ainsi une manière de simplifier un programme par la création d'une abstraction d'un traitement à effectuer. La fonction devrait pouvoir être utilisée sans avoir à connaître les détails de son fonctionnement interne. La fonction peut aussi être réutilisée par la suite dans d'autres contextes. Un aspect important d'un langage de programmation est la bibliothèque de fonctions et autres éléments qui peuvent être réutilisés, simplifiant ainsi grandement le développement de nouveaux programmes.

Des normes ont été proposées pour la documentation d'une fonction en Python. Après l'entête de la fonction, un commentaire (docstring) devrait documenter la fonction. La première ligne décrit ce que la fonction fait d'une manière succincte dans une courte phrase. Ensuite, il faut mettre une ligne vide. Le reste de la documentation devrait préciser ce que la fonction fait, en particulier, son input et son output. Il faut indiquer le rôle des paramètres, les effets sur les unités externes, les variables globales touchées, etc. La fonction `help()` de Python permet d'afficher la signature d'une fonction ainsi que le docstring.

Exemple.

```
help(dessiner_bot)
```

Résultat :

```
Help on function dessiner_bot in module __main__:

dessiner_bot(fenetre, r)
    Dessiner un Bot.

    fenetre : la surface de dessin
    r : rectangle englobant en pixels de type pygame.Rect
```

5.2.3 Passage de paramètre par valeur, par objet, ou par référence

Le fait de copier la valeur de l'argument en l'affectant au paramètre correspond au concept de passage de *paramètre par valeur*. Ainsi, dans la fonction, si la valeur du paramètre est modifiée, ceci n'affecte pas l'argument. En Python, c'est la seule manière de passer un paramètre. D'autres langages offrent une autre option : le passage de paramètre par référence qui permet de modifier directement la valeur de l'argument. Dans le cas d'un paramètre objet, comme c'est toujours le cas en Python, la valeur passée est une **copie de la référence à l'objet**, aussi désigné par *passage de paramètre par objet* (*call by object*). Ceci implique que même si la valeur de l'argument, le pointeur à l'objet, ne peut être modifiée comme tel, le contenu de l'objet auquel l'argument fait référence peut être modifié dans le cas où il est muable.

Dans l'exemple suivant, la fonction `f1` modifie le paramètre. Le résultat montre que la modification n'a pas d'effet sur l'argument `a` après l'appel de `f1(a)`. Comme l'objet est immuable, l'affectation assigne un nouvel objet à la variable dans le corps de la fonction. Ainsi, l'objet de départ de l'argument reste inchangé.

CodePython/chapitre5/ExempleModifierParametre.py

```
 1. """
 2. Exemple modification de paramètre
 3. """
 4.
 5. def f1(x):
 6.     x=1
 7. a=0
 8. print("Valeur de a avant l'appel de f1:",a)
 9. f1(a)
10. print("Valeur de a après l'appel de f1:",a)
11.
```

Résultat :

```
Valeur de a avant l'appel de f1: 0
Valeur de a après l'appel de f1: 0
```

Dans l'exemple suivant, la fonction `f2()` ne modifie pas le paramètre `une_liste` directement mais modifie le premier élément de l'objet de type liste auquel le paramètre formel fait référence. Comme une liste est un objet muable, elle peut être modifiée. Après l'appel de la fonction, la modification est bien effectuée sur l'argument `liste`.

CodePython/chapitre5/ExemplеModifierListe.py

```
"""
Exemple de modification d'une liste par une fonction
"""

def f2(une_liste):
    une_liste[0] = 1
liste = [5,6,7]
print("Valeur de la liste avant l'appel de f2:",liste)
f2(liste)
print("Valeur de la liste avant l'appel de f2:",liste)
```

Résultat :

```
Valeur de la liste avant l'appel de f2: [5, 6, 7]
Valeur de la liste après l'appel de f2: [1, 6, 7]
```

A noter que dans notre exemple de fonction `dessiner_bot()`, le premier argument qui représente la surface de dessin est effectivement modifié par l'appel à `dessiner_bot()` qui effectue les opérations graphiques sur le premier paramètre.

Dans l'exemple suivant, le paramètre `une_liste` est modifié au complet et `une_liste` fait donc référence à un nouvel objet de type liste suite à l'affectation. Comme pour l'exemple du int avec *f1()*, l'argument n'est pas changé après l'appel de la fonction.

CodePython/chapitre5/ ExempleModifierArgumentCompletListe.py

```
"""
Exemple de modification de l'argument complet de type liste par
une fonction
Même effet qu'un passage par valeur
"""

def f3(une_liste):
    une_liste = [1,2,3,4]
liste = [5,6,7]
print("Valeur de la liste avant l'appel de f3:",liste)
f3(liste)
print("Valeur de la liste après l'appel de f3:",liste)
```

Résultat :

```
Valeur de la liste avant l'appel de f3: [5, 6, 7]
Valeur de la liste après l'appel de f3: [5, 6, 7]
```

5.2.4 Fonction avec une valeur de retour

Une fonction peut retourner une valeur qui peut être récupérée après l'appel de la fonction. Ceci a déjà été vu à plusieurs occasions dans le chapitre précédent pour des exemples d'expression qui contiennent des appels de fonction.

Dans l'exemple suivant, la fonction `surface()` retourne la surface (aire) d'un rectangle représenté par les deux paramètres `largeur` et `hauteur`.

CodePython/chapitre5/ExempleFonctionSurface.py

```
1. """
2. Exemple de fonction surface qui retourne une valeur
3. """
4. def surface(largeur,
hauteur):
5.     return largeur*hauteur
6. s = surface(3,5)
7. print("La surface est :",s)
8.
```

Résultat :

```
La surface est : 15
```

Pour retourner une valeur, la fonction doit contenir un énoncé *return* suivi de la valeur à retourner. La valeur à retourner peut être une expression :

```
    return largeur*hauteur
```

La valeur retournée par la fonction peut être récupérée après l'exécution de la fonction :

```
s = surface(3,5)
```

Exercice. Définissez une fonction `surface_cercle(r)` qui retourne la surface d'un cercle de rayon `r`. La surface est définie par la formule : surface = πr^2.

Appelez la fonction à plusieurs reprises avec des valeurs différentes de rayon et affichez les résultats.

En Python, une fonction qui ne retourne pas de valeur, retourne en fait une valeur spéciale `None` de type `NoneType`. Cette valeur spéciale désigne en quelque sorte l'absence de valeur retournée par un `return`. Dans l'exemple suivant, comme la fonction `f1()` n'a pas d'énoncé `return`, c'est la valeur `None` qui est retournée.

CodePython/chapitre5/ExempleRetourNone.py

```
1. def f1(x):
2.     print("Valeur de x :",x)
3. r = f1(5)
4. print("Valeur retournée par f1 :",r)
```

Résultat :

```
Valeur de x : 5
Valeur retournée par f1 : None
```

5.2.5 Valeur de défaut et paramètres nommés

Python offre plusieurs options pour le passage des paramètres. En plus des paramètres spécifiés par leur position tel que vu jusqu'à présent, il est possible de déclarer des paramètres nommés avec valeurs de défaut qui sont donc optionnels lors de l'appel de la fonction.

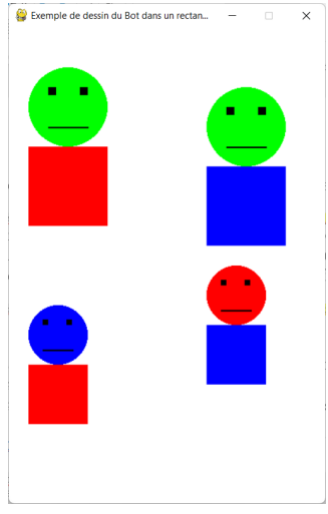
Supposons que l'on veuille permettre de modifier les couleurs de la tête et du corps du `Bot` mais en conservant les valeurs de défaut VERT et ROUGE. L'exemple suivant montre comment procéder.

CodePython/chapitre5/ExempleParametreOptionnel.py

```
1. """
2. Exemple de fonction dessiner_bot avec paramètre optionnel et valeur de défaut
3. """
4.
5. import sys, pygame
6. from pygame import Color
7.
8. def dessiner_bot(fenetre,r,couleur_tete = Color('green'),couleur_corps = Color('red')):
9.     """ Dessiner un Bot.
10.
11.     fenetre : la surface de dessin
12.     r : rectangle englobant de type pygame.Rect
13.     """
14.
15.     # Dessiner le Bot relativement au rectangle englobant r
16.     pygame.draw.ellipse(fenetre, couleur_tete, ((r.x,r.y),(r.width, r.height/2))) #
Dessiner la tête
17.     pygame.draw.rect(fenetre, Color('black'),
((r.x+r.width/4,r.y+r.height/8),(r.width/10,r.height/20))) # L'oeil gauche
18.     pygame.draw.rect(fenetre, Color('black'), ((r.x+r.width*3/4-
r.width/10,r.y+r.height/8),(r.width/10,r.height/20))) # L'oeil droit
19.     pygame.draw.line(fenetre, Color('black'),
(r.x+r.width/4,r.y+r.height*3/8),(r.x+r.width*3/4,r.y+r.height*3/8), 2) # La bouche
20.     pygame.draw.rect(fenetre, couleur_corps, ((r.x,r.y+r.height/2),(r.width,r.height/2))) #
Le corps
21.
22. pygame.init() # Initialiser Pygame
23. LARGEUR_FENETRE = 400
24. HAUTEUR_FENETRE = 600
25. fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre
26. pygame.display.set_caption('Exemple de dessin du Bot dans un rectangle englobant') #
Définir le titre dans le haut de la fenêtre
27.
28. fenetre.fill(Color('white')) # Dessiner le fond de la surface de dessin
29.
30. # Variantes d'appel de fonction avec paramètres optionnels
31. dessiner_bot(fenetre,pygame.Rect((25,50),(100,200))) # Employer valeurs de défaut
```

```
32. BLEU = (0,0,255)
33. dessiner_bot(fenetre,pygame.Rect((25,350),(75,150)),couleur_tete = Color('blue')) #
Spécifier une couleur pour la tete
34. dessiner_bot(fenetre, pygame.Rect((250,75),(100,200)),couleur_corps = Color('blue')) #
Spécifier une couleur pour le corps
35. dessiner_bot(fenetre, pygame.Rect((250,300),(75,150)),couleur_corps =
Color('blue'),couleur_tete = Color('red')) # Spécifier les deux
36.
37. pygame.display.flip() # Mettre à jour la fenêtre graphique
38.
39. # Traiter la fermeture de la fenêtre
40. while True:
41.     for event in pygame.event.get():
42.         if event.type == pygame.QUIT: # Vérifier si l'utilisateur a cliqué pour fermer la
fenêtre
43.             pygame.quit() # Terminer pygame
44.             sys.exit()
45.
```

Résultat :



La syntaxe `nom_parametre = valeur_defaut` permet de spécifier un paramètre optionnel avec une valeur de défaut :

```
def dessiner_bot(fenetre,r,couleur_tete = Color('green'),couleur_corps =
Color('red')):
```

Les paramètres optionnels doivent suivre les paramètres positionnels obligatoires dans la liste. Lorsque la fonction est appelée, il est possible de spécifier ou non un argument pour les paramètres optionnels. Contrairement aux paramètres obligatoires, l'ordre des arguments optionnels peut être différent de l'ordre des paramètres comme dans le cas suivant :

```
dessiner_bot(fenetre,     pygame.Rect((250,300),(75,150)),couleur_corps     =
Color('blue'),couleur_tete = Color('red'))
```

Enfin, les arguments des paramètres positionnels peuvent aussi être spécifiés par leur nom et l'ordre peut changer dans ce cas. Dans l'exemple suivant, l'argument pour *r* est désigné par nom et il apparaît en premier dans l'appel de la fonction alors qu'il est en deuxième dans la définition de la fonction.

CodePython/chapitre5/ ExempleArgumentParNomOrdreChange.py

```
 1. def dessiner_bot(fenetre,r,couleur_tete =
Color('green'),couleur_corps = Color('red')):
 2.     """ Dessiner un Bot.
 3.
 4.     fenetre : la surface de dessin
 5.     r : rectangle englobant de type pygame.Rect
 6.     """
 7.
 8. …
 9.
10. # Variantes d'appel de fonction avec paramètres optionnels
11. dessiner_bot(r=pygame.Rect((25,50),(100,200)),fenetre=fenetre)
# Employer valeurs de défaut
12.
```

Exercice. Ajouter des paramètres optionnels pour les couleurs de votre bonhomme (ou le `Iti`) dans la fonction de dessin et appelez la fonction à plusieurs reprises en variant les arguments.

Exercice. Dessiner plusieurs bonhommes (`Bot` et autre) dans la même fenêtre.

5.2.6 Nombre variable d'arguments (*parametres, **parametresnommes) et argument de type dict

Python permet de définir qu'une fonction prend un nombre variable d'arguments avec la notation * qui précède le nom du paramètre. Le * regroupe les éléments passés en arguments pour en former un objet itérable. Les paramètres obligatoires doivent précéder ces paramètres dans la liste des paramètres.

Dans l'exemple suivant, le paramètre `a` est obligatoire. Il est suivi de `*parametres`. Dans le corps de la fonction, `parametres` apparaît comme un tuple qui contient les arguments correspondants qui ont été regroupés. Un `for` est utilisé pour récupérer les valeurs des arguments dans l'exemple.

```
>>> def f(a, *parametres):
...     print(a)
...     for p in parametres:
...         print(p)
```

```
...
>>> f("La ","vie","est","belle")
La
vie
est
belle
```

Exercice. Coder une fonction `somme()` qui prend un nombre variable d'arguments et en retourne la somme.

Il est aussi possible de spécifier un nombre variable de paramètres nommés avec la notation ** qui précède le nom du paramètre, ce qui a pour effet de regrouper les paramètres nommés sous forme d'un objet `dict`. Les paramètres nommés doivent suivre les paramètres obligatoires et ceux de taille variable le cas échéant.

Dans l'exemple suivant, le paramètre `**parametresnommes` représente un nombre variable de paramètres nommés regroupés en `dict`. Dans le corps de la fonction, `parametresnommes` apparaît comme un `dict`. Un `for` parcoure les clés du `dict` afin d'extraire les couples `(nom, valeur)` pour chacun des arguments nommés passés à l'appel de la fonction.

```
>>> def f(**parametresnommes):
...     for pn in parametresnommes:
...         print(pn, ":", parametresnommes [pn])
...
>>> f(a=5,b=10,c=15)
a : 5
b : 10
c : 15
```

Il est aussi possible de passer un dict en argument pour spécifier les valeurs des paramètres en identifiant leurs noms. L'argument doit être précédé par ** ce qui a pour effet de dégrouper les éléments du `dict` :

```
>>> def f(a,b,c):
...     print("a :",a)
...     print("b :",b)
...     print("c :",c)
...
>>> f(**{'a' : 5, 'b' : 10, 'c' : 15})
a : 5
b : 10
c : 15
```

L'ordre des paramètres dans le `dict` n'est pas nécessairement l'ordre des paramètres dans la définition de la fonction :

```
>>> f(**{'b' : 10,'a' : 5, 'c' : 15})
a : 5
b : 10
```

```
c : 15
```

5.3 Traitement des événements de souris avec Pygame.event

Dans les applications interactives, il faut pouvoir détecter les actions de l'utilisateur (déplacement de la souris, clic de la souris, touche de clavier, etc.) et y réagir. Le module `pygame.event` inclut les mécanismes à cet effet.

Le programme suivant illustre les mécanismes de base de détection des actions de la souris. Le programme répond à un *clic de la souris* (bouton de gauche enfoncé avec Windows) en déplaçant le centre du *Bot* à la position du clic. La position du clic est déterminée par la position du curseur de souris au moment où le bouton est enfoncé.

CodePython/chapitre5/ExempleEventMouse.py

```
"""
Exemple de détection d'évènement de souris
"""
# Importer la librairie de pygame et initialiser
import pygame
from pygame import Color

def dessiner_bot(fenetre,r):
    """ Dessiner un Bot.

    fenetre : la surface de dessin
    r : rectangle englobant de type pygame.Rect
    """
…

pygame.init() # Initialiser les modules de Pygame

LARGEUR_FENETRE = 400
HAUTEUR_FENETRE = 600
fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre
pygame.display.set_caption('Exemple de gestion de la souris') # Définir le titre dans le haut de la fenêtre

fin = False

# Position initiale du Bot
x=100
y=100
# Itérer jusqu'à ce qu'un évènement provoque la fermeture de la fenêtre
while not fin:
    event = pygame.event.wait() # Chercher le prochain évènement à traiter
    if event.type == pygame.QUIT:  # Utilisateur a cliqué sur la fermeture de fenêtre ?
        fin = True  # Fin de la boucle du jeu
    elif event.type == pygame.MOUSEBUTTONUP: # Utilisateur a cliqué dans la fenêtre ?
        x=event.pos[0] # Position x de la souris
        y=event.pos[1] # Position y de la souris
        fenetre.fill(Color('white')) # Dessiner le fond de la surface de dessin
        dessiner_bot(fenetre,pygame.Rect((x-30/2,y-60/2),(30,60))) # Dessiner le Bot à la position de la souris
        pygame.display.flip() # Mettre à jour la fenêtre graphique

pygame.quit() # Terminer pygame
```

La boucle `while` traite les évènements de souris l'un après l'autre jusqu'à ce que la variable Booléenne `fin` devienne `True` :

```
while not fin:
```

Chaque fois que l'utilisateur emploie la souris, ceci produit un *événement* (*event*) *d'interface à l'utilisateur*. Un tel événement est le résultat d'une action de l'utilisateur sur un périphérique d'entrée tel que la souris ou le clavier. Le module `pygame.event` permet de traiter ces évènements. Chacun des évènements est placé dans une file d'attente et la fonction `pygame.event.wait()` retourne le prochain évènement à traiter sous forme d'un objet de type `EventType`. Si la file d'attente est vide, la fonction attend jusqu'à ce qu'un évènement soit produit.

```
    event = pygame.event.wait() # Chercher le prochain évènement à traiter
```

La ligne suivante vérifie si le type de l'évènement est la fermeture de la fenêtre qui correspond à un clic de souris dans le carré du coin supérieur droit de la fenêtre avec un x à l'intérieur :

```
    if event.type == pygame.QUIT:  # Utilisateur a cliqué sur la fermeture de
fenêtre ?
```

Si c'est le cas, la variable `fin` devient `True` ce qui termine la boucle.

```
        fin = True  # Fin de la boucle du jeu
```

Sinon, la ligne suivante vérifie si l'utilisateur a cliqué avec la souris dans la fenêtre de dessin :

```
    elif event.type == pygame.MOUSEBUTTONUP: # Utilisateur a cliqué dans la
fenêtre ?
```

Si c'est le cas, le `Bot` sera affiché à la position du clic de la souris. Le rectangle englobant est centré aux coordonnées du clic. Les coordonnées du clic sont récupérées dans `event.pos` :

```
        x=event.pos[0] # Position x de la souris
        y=event.pos[1] # Position y de la souris
```

Exercice. Reprenez l'exemple précédent avec un bonhomme de votre cru.

6 Introduction à l'animation 2D

Ce chapitre présente les concepts de base de l'animation graphique 2D. L'exemple simple d'animation qui est développé sert à introduire les concepts de programmation objet au chapitre suivant.

6.1 Une première tentative d'animation

Le principe de base de l'animation par ordinateur est semblable à la production vidéo. Il consiste à afficher une série d'images de manière suffisamment rapide pour donner l'impression d'un mouvement continu. Chacune des images produites est appelée une *scène* de l'animation. L'exemple suivant illustre le principe de base d'affichage d'une séquence de scènes par une animation élémentaire. Le programme suivant est une première tentative d'animation simple du `Bot`. L'objectif est de faire bouger le `Bot` de gauche à droite dans la fenêtre.

CodePython/chapitre6/ExempleAnimationSimple.py

```
"""
Exemple d'animation simple
"""
# Importer la librairie de pygame et initialiser
import pygame
from pygame import Color

def dessiner_bot(fenetre,r):
    """ Dessiner un Bot.

    fenetre : la surface de dessin
    r : rectangle englobant de type pygame.Rect
    """

    # Dessiner le Bot relativement au rectangle englobant r
    pygame.draw.ellipse(fenetre, Color('green'), ((r.x,r.y),(r.width, r.height/2))) #
Dessiner la tête
    pygame.draw.rect(fenetre, Color('black'),
((r.x+r.width/4,r.y+r.height/8),(r.width/10,r.height/20))) # L'oeil gauche
    pygame.draw.rect(fenetre, Color('black'), ((r.x+r.width*3/4-
r.width/10,r.y+r.height/8),(r.width/10,r.height/20))) # L'oeil droit
    pygame.draw.line(fenetre, Color('black'),
(r.x+r.width/4,r.y+r.height*3/8),(r.x+r.width*3/4,r.y+r.height*3/8), 2) # La bouche
    pygame.draw.rect(fenetre, Color('red'), ((r.x,r.y+r.height/2),(r.width,r.height/2))) #
Le corps

pygame.init() # Initialiser les modules de Pygame
LARGEUR_FENETRE = 400
HAUTEUR_FENETRE = 600
fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre
pygame.display.set_caption("Exemple d'animation simple") # Définir le titre dans le haut de
la fenêtre

horloge = pygame.time.Clock() # Pour contrôler la fréquence des scènes
position_verticale = 100 # Position verticale du Bot
VITESSE_DEPLACEMENT = 5 # En pixels par scène
TAILLE_BOT = (50,100)
# Boucle d'animation par une suite de scènes
# Le Bot avance de la bordure gauche jusqu'à la droite
for position_horizontale in range(0,LARGEUR_FENETRE-TAILLE_BOT[0],VITESSE_DEPLACEMENT) :
    fenetre.fill(Color('white')) # Dessiner le fond de la surface de dessin
    dessiner_bot(fenetre,pygame.Rect((position_horizontale,position_verticale),TAILLE_BOT))
```

```
37.     pygame.display.flip() # Mettre à jour la fenêtre graphique
38.     horloge.tick(60) # Pour animer avec 60 images par seconde
39.
40. pygame.quit() # Terminer pygame
```

La ligne suivante crée un objet de type `pygame.time.Clock()` qui sert à contrôler la fréquence de l'animation afin que le mouvement soit perceptible.

```
horloge = pygame.time.Clock() # Pour contrôler la fréquence des scènes
```

La boucle d'animation produit successivement le dessin du *Bot* de la position `x=0` jusqu'à la bordure droite de la fenêtre par pas de `VITESSE_DEPLACEMENT`. Chacune des scènes est produite par un nouveau dessin à la position *x* déterminée par le `for`.

```
    fenetre.fill(BLANC) # Dessiner le fond de la surface de dessin
dessiner_bot(fenetre,pygame.Rect((position_horizontale,position_verticale),
TAILLE_BOT))
```

La ligne suivante met à jour la fenêtre de dessin.

```
    pygame.display.flip() # Mettre à jour la fenêtre graphique
```

La manière exacte d'opérer le changement du contenu de l'écran dépend de plusieurs paramètres qui peuvent être manipulés au besoin en Python. La prochaine section explique certains détails de cette opération.

La méthode suivante introduit un délai qui produit une fréquence de 60 scènes par seconde (*frame per second*, *fps*). Ceci est important pour que l'animation soit compatible avec la vision humaine.

```
    horloge.tick(60) # Pour animer avec 60 images par seconde
```

Dans les jeux complexes, la capacité à produire la prochaine scène suffisamment rapidement devient un enjeu important.

Exercice. Modifiez le programme précédent pour que le `Bot` se déplace de haut en bas et inversement. Le `Bot` inverse sa direction de déplacement lorsqu'il atteint la bordure de la fenêtre. Le programme doit se terminer si l'utilisateur clique dans l'icône de fermeture de fenêtre.

Solution. CodePython/chapitre6/ExerciceBotRebondissant.py

```
 1. """
 2. Exercice du Bot qui rebondit
 3. """
 4. # Importer la librairie de pygame et initialiser
 5. import pygame
 6. from pygame import Color
 7.
 8. def dessiner_bot(fenetre,r):
 9.     """ Dessiner un Bot.
10.
11.     fenetre : la surface de dessin
```

```
    r : rectangle englobant de type pygame.Rect
    """

    # Dessiner le Bot relativement au rectangle englobant r
    pygame.draw.ellipse(fenetre, Color('green'), ((r.x,r.y),(r.width, r.height/2))) #
Dessiner la tête
    pygame.draw.rect(fenetre, Color('black'),
((r.x+r.width/4,r.y+r.height/8),(r.width/10,r.height/20))) # L'oeil gauche
    pygame.draw.rect(fenetre, Color('black'), ((r.x+r.width*3/4-
r.width/10,r.y+r.height/8),(r.width/10,r.height/20))) # L'oeil droit
    pygame.draw.line(fenetre, Color('black'),
(r.x+r.width/4,r.y+r.height*3/8),(r.x+r.width*3/4,r.y+r.height*3/8), 2) # La bouche
    pygame.draw.rect(fenetre, Color('red'), ((r.x,r.y+r.height/2),(r.width,r.height/2))) #
Le corps

pygame.init() # Initialiser les modules de Pygame
LARGEUR_FENETRE = 400
HAUTEUR_FENETRE = 600
fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre
pygame.display.set_caption("Exercice du Bot qui rebondit") # Définir le titre dans le haut
de la fenêtre

horloge = pygame.time.Clock() # Pour contrôler la fréquence des scènes
(x_bot,y_bot) = (175,0) # Position du Bot
vitesse_deplacement = 5 # En pixels par scène
TAILLE_BOT = (50,100)

# Boucle d'animation : Le Bot rebondit verticalement
fin = False
while not fin :
    event = pygame.event.poll() # Chercher le prochain évènement à traiter
    if event.type == pygame.QUIT:  # Utilisateur a cliqué sur la fermeture de fenêtre ?
        fin = True  # Fin de la boucle du jeu
    else :
        # Déplacer le Bot : Inverser la direction sur le bord est atteint
        if y_bot+vitesse_deplacement > HAUTEUR_FENETRE-TAILLE_BOT[1] or
y_bot+vitesse_deplacement < 0 :
            vitesse_deplacement = -vitesse_deplacement # Inverser la direction
        y_bot = y_bot+vitesse_deplacement

        fenetre.fill(Color('white')) # Dessiner le fond de la surface de dessin
        dessiner_bot(fenetre,pygame.Rect((x_bot,y_bot),TAILLE_BOT)) # Dessiner le Bot
        pygame.display.flip() # Mettre à jour la fenêtre graphique

        horloge.tick(60) # Pour animer avec 60 images par seconde

pygame.quit() # Terminer pygame
```

Exercice. Reprenez l'exercice précédent avec le bonhomme de votre choix.

Solution. CodePython/chapitre6/ExerciceItiRebondissant.py

```
"""
Exercice du Iti qui rebondit
"""
# Importer la librairie de pygame et initialiser
import pygame
from pygame import Color

def dessiner_iti(fenetre,r):
    """ Dessiner un Iti.

    fenetre : la surface de dessin
    r : rectangle englobant de type pygame.Rect
    """
…

```

```
pygame.init() # Initialiser les modules de Pygame
LARGEUR_FENETRE = 400
HAUTEUR_FENETRE = 600
fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre
pygame.display.set_caption("Exercice du Iti qui rebondit") # Définir le titre dans le haut de la fenêtre

horloge = pygame.time.Clock() # Pour contrôler la fréquence des scènes
x_iti = 175 # Position du Iti sur l'axe x
y_iti = 0 # Position initiale du Iti sur l'axe y
vitesse_deplacement = 5 # En pixels par scène
TAILLE_ITI = (50,100)

# Boucle d'animation : Le Iti rebondit verticalement
fin = False
while not fin :
    event = pygame.event.poll() # Chercher le prochain évènement à traiter
    if event.type == pygame.QUIT:  # Utilisateur a cliqué sur la fermeture de fenêtre ?
        fin = True  # Fin de la boucle du jeu
    else :
        # Déplacer le Iti : Inverser la direction sur le bord est atteint
        if y_iti+vitesse_deplacement > HAUTEUR_FENETRE-TAILLE_ITI[1] or y_iti+vitesse_deplacement < 0 :
            vitesse_deplacement = -vitesse_deplacement # Inverser la direction
        y_iti = y_iti+vitesse_deplacement

        fenetre.fill(Color('white')) # Dessiner le fond de la surface de dessin
        dessiner_iti(fenetre,pygame.Rect((x_iti,y_iti),TAILLE_ITI)) # Dessiner le Bot
        pygame.display.flip() # Mettre à jour la fenêtre graphique

        horloge.tick(60) # Pour animer avec 60 images pas seconde

pygame.quit() # Terminer pygame
```

Exercice. Le `Bot` peut se déplacer en diagonale avec une vitesse de déplacement selon l'axe `x` et une autre selon l'axe `y`. Lorsqu'un bord de fenêtre gauche ou droit est atteint, ceci inverse la direction dans l'axe `x`. Lorsqu'un bord supérieur ou inférieur est atteint, ceci inverse la direction dans l'axe `y`.

Solution. CodePython/chapitre6/ExerciceBotDiagonale.py

```
"""
Exercice du Bot qui rebondit en diagonale
"""
# Importer la librairie de pygame et initialiser
import pygame
from pygame import Color

def dessiner_bot(fenetre,r):
    """ Dessiner un Bot.

    fenetre : la surface de dessin
    r : rectangle englobant de type pygame.Rect
    """

…

pygame.init() # Initialiser les modules de Pygame
LARGEUR_FENETRE = 400
HAUTEUR_FENETRE = 600
fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre
pygame.display.set_caption("Exercice du Bot qui rebondit en diagonale") # Définir le titre dans le haut de la fenêtre

horloge = pygame.time.Clock() # Pour contrôler la fréquence des scènes
(x_bot,y_bot) = (0,0) # Position initiale du Bot
vitesse_deplacement_x = 5 # En pixels par scène
vitesse_deplacement_y = 10
TAILLE_BOT = (50,100)
```

```
28.
29. # Boucle d'animation : Le Bot se déplace en diagonale
30. fin = False
31. while not fin :
32.     event = pygame.event.poll() # Chercher le prochain évènement à traiter
33.     if event.type == pygame.QUIT:  # Utilisateur a cliqué sur la fermeture de fenêtre ?
34.         fin = True  # Fin de la boucle du jeu
35.     else :
36.         # Déplacer le Bot : Inverser la direction si le bord est atteint
37.         if x_bot+vitesse_deplacement_x > LARGEUR_FENETRE-TAILLE_BOT[0] or
x_bot+vitesse_deplacement_x < 0 :
38.             vitesse_deplacement_x = -vitesse_deplacement_x # Inverser la direction en x
39.         x_bot = x_bot+vitesse_deplacement_x
40.         if y_bot+vitesse_deplacement_y > HAUTEUR_FENETRE-TAILLE_BOT[1] or
y_bot+vitesse_deplacement_y < 0 :
41.             vitesse_deplacement_y = -vitesse_deplacement_y # Inverser la direction en y
42.         y_bot = y_bot+vitesse_deplacement_y
43.
44.         fenetre.fill(Color('white')) # Dessiner le fond de la surface de dessin
45.         dessiner_bot(fenetre,pygame.Rect((x_bot,y_bot),TAILLE_BOT)) # Dessiner le Bot
46.         pygame.display.flip() # Mettre à jour la fenêtre graphique
47.
48.
49.         horloge.tick(60) # Pour animer avec 60 images par seconde
50.
51. pygame.quit() # Terminer pygame
```

Exercice. Maintenant combinez quelques `Bot` et `Iti` dans la même animation avec des positions initiales, vitesses et tailles différentes.

Solution. CodePython/chapitre6/ExerciceBotEtItiDiagonale.py

```
  1. """
  2. Exercice : 2 Bot et 2 Iti qui rebondissent en diagonale
  3. """
  4. # Importer la librairie de pygame et initialiser
  5. import pygame
  6. from pygame import Color
  7.
  8. def dessiner_bot(fenetre,r):
  9.     """ Dessiner un Bot.
 10.
 11.     fenetre : la surface de dessin
 12.     r : rectangle englobant de type pygame.Rect
 13.     """
 14.
 15.     # Dessiner le Bot relativement au rectangle englobant r
 16.     pygame.draw.ellipse(fenetre, Color('green'), ((r.x,r.y),(r.width, r.height/2))) #
Dessiner la tête
 17.     pygame.draw.rect(fenetre, Color('black'),
((r.x+r.width/4,r.y+r.height/8),(r.width/10,r.height/20))) # L'oeil gauche
 18.     pygame.draw.rect(fenetre, Color('black'), ((r.x+r.width*3/4-
r.width/10,r.y+r.height/8),(r.width/10,r.height/20))) # L'oeil droit
 19.     pygame.draw.line(fenetre, Color('black'),
(r.x+r.width/4,r.y+r.height*3/8),(r.x+r.width*3/4,r.y+r.height*3/8), 2) # La bouche
 20.     pygame.draw.rect(fenetre, Color('red'), ((r.x,r.y+r.height/2),(r.width,r.height/2))) #
Le corps
 21.
 22. def dessiner_iti(fenetre,r):
 23.     """ Dessiner un Iti.
 24.
 25.     fenetre : la surface de dessin
 26.     r : rectangle englobant de type pygame.Rect
 27.     """
 28.
 29.     # Coordonnées du milieu du rectangle englobant pour faciliter les calculs
 30.     milieux = r.x + r.width/2;
 31.     milieuy = r.y + r.height/2;
 32.
 33.     pygame.draw.ellipse(fenetre, Color('pink'),
((r.x+r.width/3,r.y),(r.width/3,r.height/4))) # Dessiner la tête
```

```
 34.     pygame.draw.arc(fenetre, Color('black'),((milieux-
r.width/12,r.y+r.height/8),(r.width/6,r.height/14)),3.1416,0,2) # Le sourire
 35.     pygame.draw.ellipse(fenetre, Color('black'), ((milieux-
r.width/8,r.y+r.height/12),(r.width/12,r.height/24))) # L'oeil gauche
 36.     pygame.draw.ellipse(fenetre, Color('black'), ((milieux+r.width/8-
r.width/12,r.y+r.height/12),(r.width/12,r.height/24))) # L'oeil droit
 37.     pygame.draw.line(fenetre, Color('black'),
(milieux,r.y+r.height/4),(milieux,r.y+r.height*3/4), 2) # Le corps
 38.     pygame.draw.line(fenetre, Color('black'), (r.x,r.y+r.height/4),(milieux,milieuy), 2) #
Bras gauche
 39.     pygame.draw.line(fenetre, Color('black'),
(r.x+r.width,r.y+r.height/4),(milieux,milieuy), 2) # Bras droit
 40.     pygame.draw.line(fenetre, Color('black'),
(r.x,r.y+r.height),(milieux,r.y+r.height*3/4), 2) # Jambe gauche
 41.     pygame.draw.line(fenetre, Color('black'),
(r.x+r.width,r.y+r.height),(milieux,r.y+r.height*3/4), 2) # Jambe droite
 42.
 43. pygame.init() # Initialiser les modules de Pygame
 44. LARGEUR_FENETRE = 400
 45. HAUTEUR_FENETRE = 600
 46. fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre
 47. pygame.display.set_caption("Exercice des Bots et Itis en diagonale") # Définir le titre
dans le haut de la fenêtre
 48.
 49. horloge = pygame.time.Clock() # Pour contrôler la fréquence des scènes
 50.
 51. # Données du Bot1
 52. (x_bot1,y_bot1) = (0,0) # Position initiale du Bot1 sur l'axe x
 53. (vitesse_x_bot1,vitesse_y_bot1) = (5,10) # En pixels par scène
 54. TAILLE_BOT1 = (20,40)
 55.
 56. # Données du Bot2
 57. (x_bot2,y_bot2) = (100,100)
 58. (vitesse_x_bot2,vitesse_y_bot2) = (10,2) # En pixels par scène
 59. TAILLE_BOT2 = (30,60)
 60.
 61. # Données du Iti1
 62. (x_iti1,y_iti1) = (200,150)
 63. (vitesse_x_iti1,vitesse_y_iti1) = (10,2) # En pixels par scène
 64. TAILLE_ITI1 = (40,80)
 65.
 66. # Données du Iti2
 67. (x_iti2,y_iti2) = (300,300)
 68. (vitesse_x_iti2,vitesse_y_iti2) = (5,10) # En pixels par scène
 69. TAILLE_ITI2 = (50,100)
 70.
 71. # Boucle d'animation
 72. fin = False
 73. while not fin :
 74.     event = pygame.event.poll() # Chercher le prochain évènement à traiter
 75.     if event.type == pygame.QUIT:  # Utilisateur a cliqué sur la fermeture de fenêtre ?
 76.         fin = True  # Fin de la boucle du jeu
 77.     else :
 78.         # Déplacer le Bot1
 79.         if x_bot1+vitesse_x_bot1 > LARGEUR_FENETRE-TAILLE_BOT1[0] or x_bot1+vitesse_x_bot1
< 0 :
 80.             vitesse_x_bot1 = -vitesse_x_bot1 # Inverser la direction en x
 81.         x_bot1 = x_bot1+vitesse_x_bot1
 82.         if y_bot1+vitesse_y_bot1 > HAUTEUR_FENETRE-TAILLE_BOT1[1] or y_bot1+vitesse_y_bot1
< 0 :
 83.             vitesse_y_bot1 = -vitesse_y_bot1 # Inverser la direction en y
 84.         y_bot1 = y_bot1+vitesse_y_bot1
 85.
 86.         # Déplacer le Bot2
 87.         if x_bot2+vitesse_x_bot2 > LARGEUR_FENETRE-TAILLE_BOT2[0] or x_bot2+vitesse_x_bot2
< 0 :
 88.             vitesse_x_bot2 = -vitesse_x_bot2 # Inverser la direction en x
 89.         x_bot2 = x_bot2+vitesse_x_bot2
 90.         if y_bot2+vitesse_y_bot2 > HAUTEUR_FENETRE-TAILLE_BOT2[1] or y_bot2+vitesse_y_bot2
< 0 :
 91.             vitesse_y_bot2 = -vitesse_y_bot2 # Inverser la direction en y
 92.         y_bot2 = y_bot2+vitesse_y_bot2
 93.
 94.         # Déplacer le Iti1
```

```
 95.         if x_iti1+vitesse_x_iti1 > LARGEUR_FENETRE-TAILLE_ITI1[0] or x_iti1+vitesse_x_iti1
< 0 :
 96.             vitesse_x_iti1 = -vitesse_x_iti1 # Inverser la direction en x
 97.         x_iti1 = x_iti1+vitesse_x_iti1
 98.         if y_iti1+vitesse_y_iti1 > HAUTEUR_FENETRE-TAILLE_ITI1[1] or y_iti1+vitesse_y_iti1
< 0 :
 99.             vitesse_y_iti1 = -vitesse_y_iti1 # Inverser la direction en y
100.         y_iti1 = y_iti1+vitesse_y_iti1
101.
102.         # Déplacer le Iti2
103.         if x_iti2+vitesse_x_iti2 > LARGEUR_FENETRE-TAILLE_ITI2[0] or x_iti2+vitesse_x_iti2
< 0 :
104.             vitesse_x_iti2 = -vitesse_x_iti2 # Inverser la direction en x
105.         x_iti2 = x_iti2+vitesse_x_iti2
106.         if y_iti2+vitesse_y_iti2 > HAUTEUR_FENETRE-TAILLE_ITI2[1] or y_iti2+vitesse_y_iti2
< 0 :
107.             vitesse_y_iti2 = -vitesse_y_iti2 # Inverser la direction en y
108.         y_iti2 = y_iti2+vitesse_y_iti2
109.
110.
111.         fenetre.fill(Color('white')) # Dessiner le fond de la surface de dessin
112.         dessiner_bot(fenetre,pygame.Rect((x_bot1,y_bot1),TAILLE_BOT1)) # Dessiner le Bot1
113.         dessiner_bot(fenetre,pygame.Rect((x_bot2,y_bot2),TAILLE_BOT2)) # Dessiner le Bot2
114.         dessiner_iti(fenetre,pygame.Rect((x_iti1,y_iti1),TAILLE_ITI1)) # Dessiner le Iti1
115.         dessiner_iti(fenetre,pygame.Rect((x_iti2,y_iti2),TAILLE_ITI2)) # Dessiner le Iti2
116.         pygame.display.flip() # Mettre à jour la fenêtre graphique
117.
118.         horloge.tick(60) # Pour animer avec 60 images par seconde
119.
120. pygame.quit() # Terminer pygame
121.
```

Le programme de l'exercice précédent devient compliqué et les possibilités d'erreurs de codage se multiplient ! Plusieurs parties du code sont répétitives. Le prochain chapitre montre comment simplifier et mieux organiser le programme en exploitant de manière judicieuse la notion d'objet et de classe.

6.2 Animation par double tampon

Cette section explique certains aspects de l'implémentation de l'animation graphique par ordinateur qui sont effectués de manière transparente par `Pygame`. Cette section peut être omise sans affecter la compréhension des concepts présentés par la suite.

Une technique fondamentale de l'animation par ordinateur est l'emploi du *double tampon* (*double buffering*) et de *l'alternance de tampon* (*page flipping*). Plutôt que de dessiner une scène directement dans la zone mémoire de la fenêtre affichée à l'écran, le dessin est effectué dans une autre zone mémoire aussi appelée *tampon*. La figure suivante montre les deux zones mémoire au début d'une itération d'animation pour notre exemple de déplacement du `Bot` de gauche à droite.

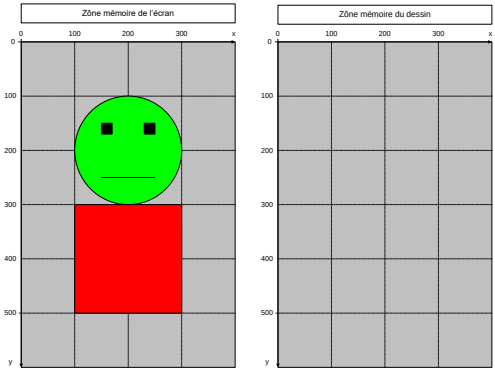


Figure 9. Double tampon.

La zone mémoire du dessin est initialement vide. Les opérations de dessin d'une scène du `Bot` sont effectuées dans cette zone. La position du `Bot` sera légèrement décalée vers la droite dans le dessin de la prochaine scène tel qu'illustré par la séquence suivante.

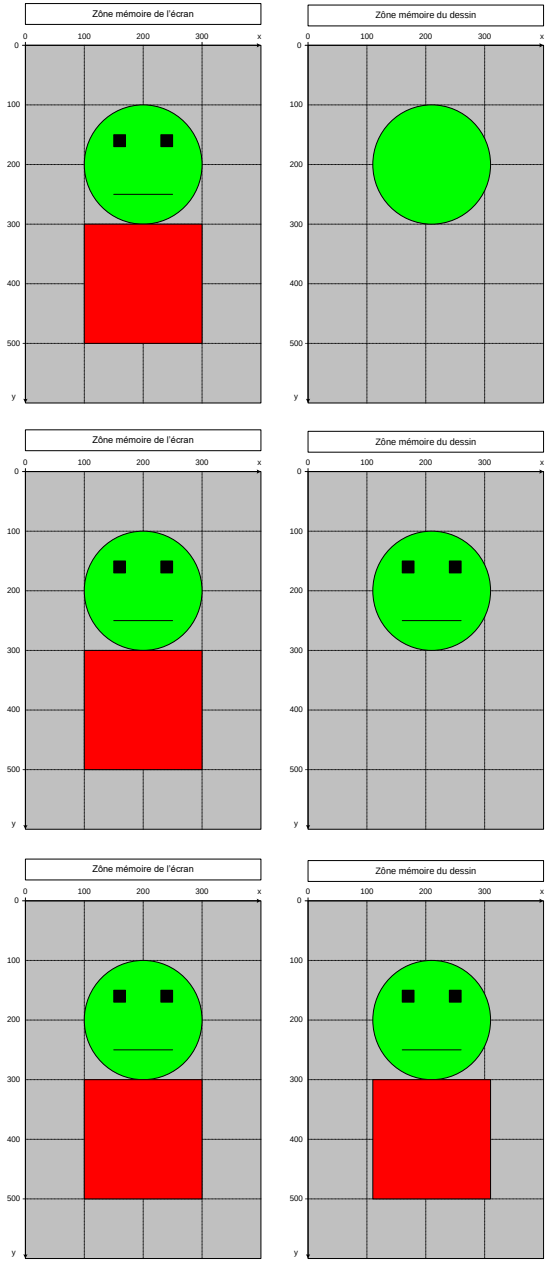


Figure 10. Dessin effectué dans la zone de dessin.

Lorsque le dessin est terminé, la zone mémoire du dessin est copiée dans la zone de l'écran le plus rapidement possible, ce qui provoque l'affichage de

la nouvelle scène à l'écran. Si les opérations graphiques étaient directement effectuées dans la zone de mémoire de l'écran, il pourrait y avoir un effet indésirable de *scintillement* (*flickering*) à l'écran, surtout si le dessin est long à produire.

Alternance de tampon (*page flipping*)

Dans les applications graphiques à haute performance, en particulier pour les jeux d'ordinateur, il est avantageux d'éviter la copie de la zone de dessin à la zone de l'écran en alternant la zone de mémoire associée à l'écran. Le mécanisme d'alternance peut être effectué en matériel par la carte graphique afin de le rendre très rapide. Ceci évite l'étape de copie. Ainsi, chacune des scènes est dessinée en alternance sur deux tampons de mémoire qui sont alternativement associés à l'écran. Les deux zones de mémoire ont ainsi des rôles symétriques.

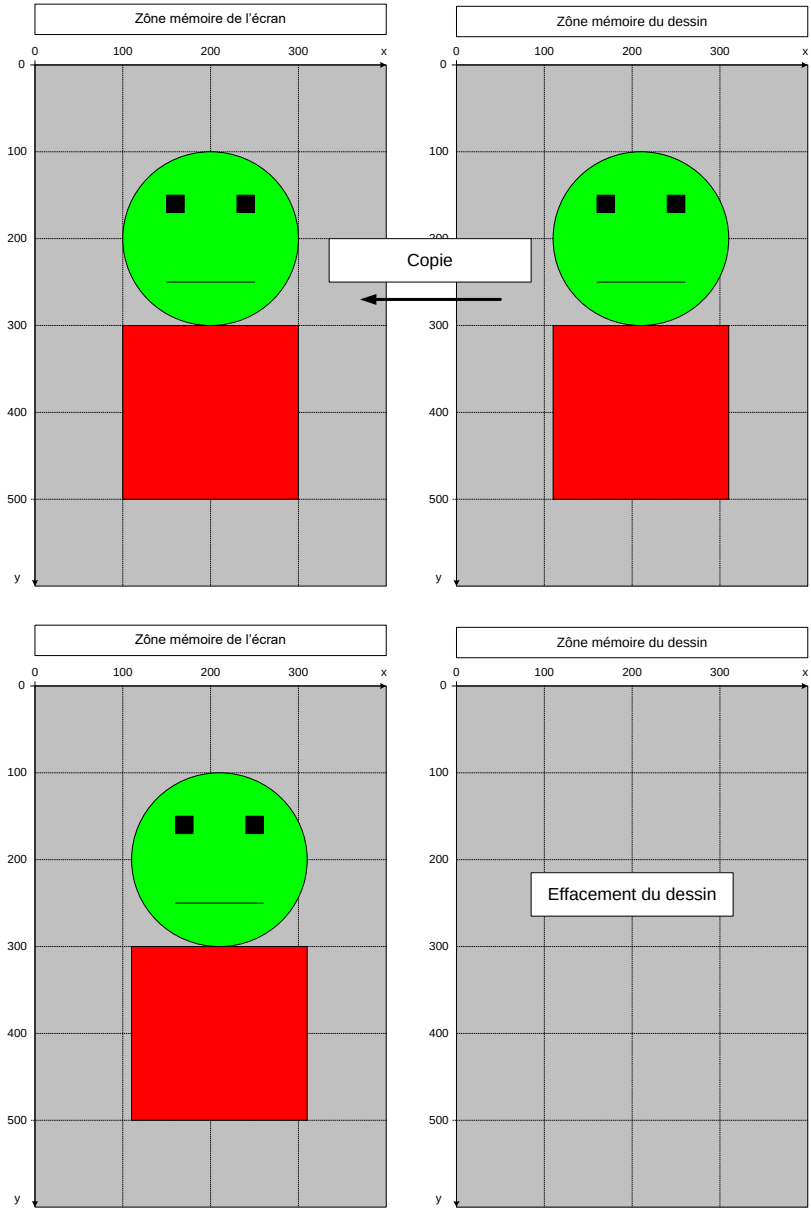


Figure 11. Copie de l'image de la zone de dessin dans la zone de l'écran.

7 Développement de classes : conception objet

La *conception objet* désigne le problème de conception d'un programme objet comportant des classes et des objets. Un aspect particulièrement important à considérer est le découpage du programme en classes. Ce chapitre approfondit quelques notions de base de la conception objet : diviser pour régner, minimiser la répétition, encapsulation, interface, cohésion, couplage, et relation d'héritage. Les principes de Python touchant à l'organisation, la compilation et l'exécution d'un programme composé de plusieurs modules sont aussi abordés.

7.1 Découpage d'un programme en classes

Les classes prédéfinies de Python ont été introduites dans les chapitres précédents. Ce chapitre montre comment développer de nouvelles classes.

Génie logiciel : diviser pour régner

Le principe de diviser pour régner est appliqué dans plusieurs contextes en informatique. Pour solutionner un gros problème, il est souvent avantageux de le décomposer en plusieurs sous-problèmes plus simples, et de tenter de solutionner les sous-problèmes indépendamment les uns des autres. La solution au problème plus large est obtenue en combinant les solutions des sous-problèmes.

Nous avons déjà abordé les fonctions qui permettent d'appliquer ce principe en représentant la solution à un sous-problème sous forme d'une fonction. La notion de classe en programmation objet permet de tirer profit de ce principe d'une manière plus sophistiquée en regroupant ensemble les données et les fonctions qui sont fortement interreliées pour résoudre un sous-problème.

Génie logiciel : éviter de répéter

Comme pour les fonctions, le mécanisme de classe permet aussi d'éviter la répétition de code mais d'une manière plus élaborée en exploitant les relations entre les variables et les énoncés de traitement.

L'exemple suivant opère une réorganisation du programme du dernier exercice du chapitre précédent avec plusieurs `Bot` et `Iti`. Plutôt que de tout mettre dans le programme principal, deux nouvelles classes sont créées, une

pour les `Bot`, appelée `BotAnime`, et une autre pour les `Iti`, `ItiAnime`. Chacun des `Bot` à animer est représenté dans le programme par un objet de la classe `BotAnime`, et de même pour les `Iti`. Ceci permet d'éviter de répéter le même code à plusieurs reprises pour chacun des objets de la même classe.

CodePython/chapitre7/ExempleAvecClasse.py

```
"""
Exemple d'animation d'entités : création des classes BotAnime et ItiAnime
"""
# Importer la librairie de pygame et initialiser
import pygame
from pygame import Color
class BotAnime :
    """ Un objet représente un Bot qui est animé dans une fenêtre Pygame

    Le Bot est inscrit dans le rectangle englobant défini par r. Il se déplace en diagonale selon vitesse.
        r : pygame.Rect       Le rectangle englobant
        v : [int,int]   Vitesse de déplacement selon les deux axes x et y
    """

    def __init__(self,rectangle,vitesse):
        self.r = rectangle
        self.v = vitesse

    def dessiner(self,fenetre):
        """ Dessiner un Bot.

        Le Bot est inscrit dans le rectangle englobant défini par la variable d'objet r dans une fenetre de Pygame
        """

        pygame.draw.ellipse(fenetre, Color('green'), ((self.r.x,self.r.y),(self.r.width, self.r.height/2))) # Dessiner la tête
        pygame.draw.rect(fenetre, Color('black'), ((self.r.x+self.r.width/4,self.r.y+self.r.height/8),(self.r.width/10,self.r.height/20))) # L'oeil gauche
        pygame.draw.rect(fenetre, Color('black'), ((self.r.x+self.r.width*3/4-self.r.width/10,self.r.y+self.r.height/8),(self.r.width/10,self.r.height/20))) # L'oeil droit
        pygame.draw.line(fenetre, Color('black'), (self.r.x+self.r.width/4,self.r.y+self.r.height*3/8),(self.r.x+self.r.width*3/4,self.r.y+self.r.height*3/8), 2) # La bouche
        pygame.draw.rect(fenetre, Color('red'), ((self.r.x,self.r.y+self.r.height/2),(self.r.width,self.r.height/2))) # Le corps


    def deplacer(self,largeur_fenetre,hauteur_fenetre):
        """ Déplacer le Bot en diagonale en rebondissant sur les bords de la fenetre"""
        if self.r.x+self.v[0] > largeur_fenetre-self.r.width or self.r.x+self.v[0] < 0 :
            self.v[0] = -self.v[0] # Inverser la direction en x
        self.r.x = self.r.x+self.v[0]
        if self.r.y+self.v[1] > hauteur_fenetre-self.r.height or self.r.y+self.v[1] < 0 :
            self.v[1] = -self.v[1] # Inverser la direction en y
        self.r.y = self.r.y+self.v[1]

class ItiAnime :
    """ Un objet représente un Iti qui est animé dans une fenêtre Pygame

    Le Iti est inscrit dans le rectangle englobant défini par rectangle. Il se déplace en diagonale selon vitesse.
        r : pygame.Rect       Le rectangle englobant
        v : [int,int]   Vitesse de déplacement selon les deux axes x et y
    """

    def __init__(self,rectangle,vitesse):
        self.r = rectangle
        self.v = vitesse
```

```
52.
53.     def dessiner(self,fenetre):
54.         """ Dessiner un Iti.
55.
56.         Le Iti est inscrit dans le rectangle englobant défini par la variable d'objet r
dans une fenetre de Pygame
57.         """
58.         self.milieux = self.r.x + self.r.width/2;
59.         self.milieuy = self.r.y + self.r.height/2;
60.
61.         pygame.draw.ellipse(fenetre, Color('pink'),
((self.r.x+self.r.width/3,self.r.y),(self.r.width/3,self.r.height/4))) # Dessiner la tête
62.         pygame.draw.arc(fenetre, Color('black'),((self.milieux-
self.r.width/12,self.r.y+self.r.height/8),(self.r.width/6,self.r.height/14)),3.1416,0,2) # Le
sourire
63.         pygame.draw.ellipse(fenetre, Color('black'), ((self.milieux-
self.r.width/8,self.r.y+self.r.height/12),(self.r.width/12,self.r.height/24))) # L'oeil gauche
64.         pygame.draw.ellipse(fenetre, Color('black'), ((self.milieux+self.r.width/8-
self.r.width/12,self.r.y+self.r.height/12),(self.r.width/12,self.r.height/24))) # L'oeil droit
65.         pygame.draw.line(fenetre, Color('black'),
(self.milieux,self.r.y+self.r.height/4),(self.milieux,self.r.y+self.r.height*3/4), 2) # Le
corps
66.         pygame.draw.line(fenetre, Color('black'),
(self.r.x,self.r.y+self.r.height/4),(self.milieux,self.milieuy), 2) # Bras gauche
67.         pygame.draw.line(fenetre, Color('black'),
(self.r.x+self.r.width,self.r.y+self.r.height/4),(self.milieux,self.milieuy), 2) # Bras droit
68.         pygame.draw.line(fenetre, Color('black'),
(self.r.x,self.r.y+self.r.height),(self.milieux,self.r.y+self.r.height*3/4), 2) # Jambe gauche
69.         pygame.draw.line(fenetre, Color('black'),
(self.r.x+self.r.width,self.r.y+self.r.height),(self.milieux,self.r.y+self.r.height*3/4), 2) #
Jambe droite
70.
71.
72.     def deplacer(self,largeur_fenetre,hauteur_fenetre):
73.         """ Déplacer le Iti en diagonale en rebondissant sur les bords de la fenetre"""
74.         if self.r.x+self.v[0] > largeur_fenetre-self.r.width or self.r.x+self.v[0] < 0 :
75.             self.v[0] = -self.v[0] # Inverser la direction en x
76.         self.r.x = self.r.x+self.v[0]
77.         if self.r.y+self.v[1] > hauteur_fenetre-self.r.height or self.r.y+self.v[1] < 0 :
78.             self.v[1] = -self.v[1] # Inverser la direction en y
79.         self.r.y = self.r.y+self.v[1]
80.
81. pygame.init() # Initialiser les modules de Pygame
82. LARGEUR_FENETRE = 400
83. HAUTEUR_FENETRE = 600
84. fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre
85. pygame.display.set_caption("Bot et Iti en diagonale avec classe") # Définir le titre dans
le haut de la fenêtre
86.
87. horloge = pygame.time.Clock() # Pour contrôler la fréquence des scènes
88.
89. # Création de deux BotAnime et deux ItiAnime
90. bot1 = BotAnime(pygame.Rect((0,0),(20,40)),[5,10])
91. bot2 = BotAnime(pygame.Rect((100,200),(30,60)),[0,2])
92. iti1 = ItiAnime(pygame.Rect((200,150),(40,80)),[3,3])
93. iti2 = ItiAnime(pygame.Rect((300,300),(50,100)),[5,10])
94.
95. # Boucle d'animation
96. fin = False
97. while not fin :
98.     event = pygame.event.poll() # Chercher le prochain évènement à traiter
99.     if event.type == pygame.QUIT:  # Utilisateur a cliqué sur la fermeture de fenêtre ?
100.         fin = True  # Fin de la boucle du jeu
101.     else :
102.         bot1.deplacer(LARGEUR_FENETRE,HAUTEUR_FENETRE)
103.         bot2.deplacer(LARGEUR_FENETRE,HAUTEUR_FENETRE)
104.         iti1.deplacer(LARGEUR_FENETRE,HAUTEUR_FENETRE)
105.         iti2.deplacer(LARGEUR_FENETRE,HAUTEUR_FENETRE)
106.
107.         fenetre.fill(Color('white')) # Dessiner le fond de la surface de dessin
108.         bot1.dessiner(fenetre)
109.         bot2.dessiner(fenetre)
110.         iti1.dessiner(fenetre)
111.         iti2.dessiner(fenetre)
112.
```

```
113.         pygame.display.flip() # Mettre à jour la fenêtre graphique
114.
115.         horloge.tick(60) # Pour animer avec 60 images pas seconde
116.
117. pygame.quit() # Terminer pygame
```

Comme pour la définition d'une fonction, lorsque la définition d'une classe est rencontrée dans le programme principal, elle est mémorisée par l'interprète Python mais le code des méthodes n'est pas exécuté comme tel. La classe `BotAnime` contient les données et opérations d'un objet `Bot`. La définition de la classe débute par un entête :

```
class BotAnime :
```

L'entête est suivi du corps de la classe qui définit les données et les opérations de la classe. Une méthode est une fonction spéciale qui se distingue par le fait qu'elle est appelée sur un objet particulier d'une classe. La fonction possède un paramètre supplémentaire qui représente l'objet sur lequel la fonction est appelée[14]. Ce paramètre supplémentaire, nommé `self` par convention, est le premier paramètre dans la définition de la méthode. Les variables qui décrivent un objet `Bot` (`rectangle` et `vitesse`) peuvent être initialisées dans une méthode spéciale dont le nom est `__init__()`:

```
    def __init__(self,rectangle,vitesse):
        self.r = rectangle
        self.v = vitesse
```

Comme pour une fonction, le code qui fait partie de la méthode doit être indenté par rapport à l'entête. Par convention, la méthode `__init()__` est appelée automatiquement lorsqu'un objet est créé. Un objet est créé par un appel au constructeur d'objet de la forme `NomClasse(paramètres)`, où les paramètres correspondent aux paramètres de la méthode `__init()__`, sauf pour le premier paramètre *self* qui représente une référence à l'objet en construction. Le constructeur d'objet retourne un nouvel objet de type `NomClasse`. Dans notre exemple, chacun des appels ci-bas crée un objet du type `BotAnime`. Les paramètres `rectangle` et `vitesse` sont utilisés pour initialiser les variables d'objet, `r` et `v`.

```
bot1 = BotAnime(pygame.Rect((0,0),(20,40)),[5,10])
bot2 = BotAnime(pygame.Rect((100,200),(30,60)),[0,2])
```

La méthode `__init()__` est appelée automatiquement lors de la création d'un nouvel objet. Elle ne devrait pas être appelée directement.

[14] Habituellement, dans les langages objet, ce paramètre n'est pas déclaré car il est implicite.

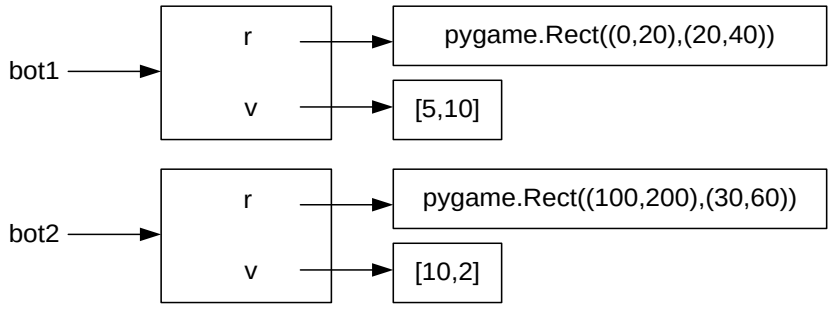
Méthode publique/privée, méthodes spéciales (magiques), *dunder methods*

Une méthode en Python est toujours publique, c'est-à-dire qu'elle peut être appelée de l'extérieur de la classe. Une méthode dite privée ne peut être appelée de l'extérieur de la classe. Même si Python n'offre pas de mécanisme de méthode privée, des conventions sont suggérées pour désigner une méthode qui devrait être privée. Un nom de méthode qui débute par un `_` est privé par convention par opposition à un nom public. L'encadrement d'un nom de méthode par deux caractères de soulignement `__`[15] (tiret bas) au début et à la fin du nom est une convention de Python pour désigner des *méthodes spéciales* au-delà du fait qu'elles devraient être privées. On utilise parfois l'expression « *dunder methods* » pour *double underscore* afin de désigner ces méthodes, ou encore *méthodes magiques* (*magic methods*). Plusieurs méthodes spéciales sont pré-définies par convention telles que `__str__()` qui retourne une représentation en chaîne de caractère de l'objet. Ces méthodes ne sont pas faites pour être appelées directement mais indirectement via des opérations standards. Par exemple, la fonction `__str__()` est appelée par les fonctions prédéfinies `str()`, `format()` et `print()` pour produire une représentation sous forme de chaîne de caractère d'un objet quelconque. Par défaut, `__str__()` appelle la méthode `__repr__()` qui retourne une représentation sous forme de chaîne plus formelle que `__str__()` et moins lisible dans plusieurs cas. Il est possible de redéfinir ces méthodes au besoin pour personnaliser le comportement par défaut.

Le module `operator` contient les méthodes spéciales qui définissent le comportement des opérateurs Python (`+, -, >, in,` …). Ces méthodes spéciales permettent de personnaliser le comportement des opérateurs.

Il est possible de créer plusieurs objets de la même classe. Pour chacun des objets créés, des *variables d'objets* distinctes sont créées même si les noms des variables sont identiques d'un objet à l'autre. Chacun des objets a un espace de nom privé qui permet de distinguer les différentes variables d'objet. Ainsi pour notre exemple, les deux appels au constructeur d'objet créent deux objets de type `BotAnime` qui ont chacun leurs propres variables d'objets tel qu'illustré à la figure suivante :

[15] Attention ! La suite des deux caractères « _ _ » peut porter à confusion avec certaines fontes comme celle de notre texte parce que les deux caractères sont collés lorsqu'ils sont juxtaposés (sans espace).



Chacun des objets a sa propre identité. La variable `bot1` réfère au premier objet au moyen de son identifiant d'objet. La variable `bot2` réfère au deuxième objet. Chacune des variables créées dans la méthode `__init__()` est appelée une *variable d'objet* parce que chacun des objets possède une telle variable. A l'intérieur de la définition d'une classe, la syntaxe `self.nom_de_variable` sert à accéder à une variable d'objet. Le nom `self` désigne l'objet lui-même.

État (*state*) d'un objet

L'ensemble des variables d'objet (aussi appelées *variables d'instances* ou *attributs d'objets*) d'un objet représentent l'*état* de l'objet. Un objet évolue en passant d'un état à un autre suite aux modifications apportées aux variables de l'objet.

Dans notre exemple, en plus du constructeur, deux autres méthodes sont définies : `deplacer()` et `dessiner()`. Ce sont des *méthodes d'objet* car elles accèdent aux variables d'objet.

Comportement (*behavior*) d'un objet

Les méthodes d'objet représentent le comportement d'un objet. Elles décrivent les opérations applicables à un objet de la classe.

La méthode `deplacer()` décrit le comportement de déplacement d'un `BotAnime` pour passer à la scène suivante.

```
    def deplacer(self,largeur_fenetre,hauteur_fenetre):
      """ Déplacer le Bot en diagonale en rebondissant sur les bords de la fenetre"""
      if self.r.x+self.vitesse[0] > largeur_fenetre-self.r.width or self.r.x+self.vitesse[0] < 0 :
          self.vitesse[0] = -self.vitesse[0] # Inverser la direction en x
      self.r.x = self.r.x+self.vitesse[0]
      if self.r.y+self.vitesse[1] > hauteur_fenetre-self.r.height or self.r.y+self.vitesse[1] < 0 :
          self.vitesse[1] = -self.vitesse[1] # Inverser la direction en y
      self.r.y = self.r.y+self.vitesse[1]
```

Comme pour le cas de `__init__()`, le premier paramètre `self` sert à représenter l'objet sur lequel la méthode est appliquée mais il n'est pas passé par les arguments. Il est désigné par la syntaxe `nom_objet.nom_methode()`.

Les paramètres `largeur_fenetre` et `hauteur_fenetre` sont employés pour déterminer si la bordure de la fenêtre est atteinte. La méthode utilise aussi les variables d'objet (`r` et `v`). A noter que les variables d'objet sont accédées par la syntaxe `self.nom_variable`, ce qui n'est pas le cas des paramètres.

La méthode `dessiner()` utilise la variable d'objet `r` qui n'est plus passée en paramètre par opposé à la version du programme sans classes.

```
    def dessiner(self,fenetre):
        """ Dessiner un Bot.

        Le Bot est inscrit dans le rectangle englobant défini par la variable d'objet rectangle dans une
fenetre de Pygame
        """

        pygame.draw.ellipse(fenetre,        Color('green'),        ((self.r.x,self.r.y),(self.r.width,
self.r.height/2))) # Dessiner la tête
        pygame.draw.rect(fenetre,                                                    Color('black'),
((self.r.x+self.r.width/4,self.r.y+self.r.height/8),(self.r.width/10,self.r.height/20))) # L'oeil gauche
        pygame.draw.rect(fenetre,Color('black'),((self.r.x+self.r.width*3/4-
self.r.width/10,self.r.y+self.r.height/8), (self.r.width/10,self.r.height/20))) # L'oeil droit
        pygame.draw.line(fenetre,                                                    Color('black'),
(self.r.x+self.r.width/4,self.r.y+self.r.height*3/8),(self.r.x+self.r.width*3/4,self.r.y+self.r.height*
3/8), 2) # La bouche
        pygame.draw.rect(fenetre,                                                     Color('red'),
((self.r.x,self.r.y+self.r.height/2),(self.r.width,self.r.height/2))) # Le corps
```

Lorsqu'un script est invoqué, ce sont les instructions du programme principal qui sont exécutées. Le programme principal utilise les deux classes `BotAnime` et `ItiAnime`. Les quatre bonhommes à animer sont créés par les appels au constructeur de la classe visée qui est désigné par le nom de la classe :

```
bot1 = BotAnime(pygame.Rect((0,0),(20,40)),[5,10])
bot2 = BotAnime(pygame.Rect((100,200),(30,60)),[0,2])
iti1 = ItiAnime(pygame.Rect((200,150),(40,80)),[3,3])
iti2 = ItiAnime(pygame.Rect((300,300),(50,100)),[5,10])
```

Par la suite, dans la boucle d'animation les objets sont déplacés pour la prochaine scène par appel à la méthode d'objet `deplacer()` :

```
        bot1.deplacer(LARGEUR_FENETRE,HAUTEUR_FENETRE)
        bot2.deplacer(LARGEUR_FENETRE,HAUTEUR_FENETRE)
        iti1.deplacer(LARGEUR_FENETRE,HAUTEUR_FENETRE)
        iti2.deplacer(LARGEUR_FENETRE,HAUTEUR_FENETRE)
```

La syntaxe `bot1.deplacer()` est employée pour appeler la méthode d'objet `deplacer()` sur l'objet représenté par la variable `bot1`. Le premier

paramètre `self` d'une méthode d'objet correspond alors à l'objet sur lequel la méthode est appelée. Et ainsi de suite pour les trois autres objets de la scène. Ensuite les objets sont dessinés à l'écran par appel à la méthode `dessiner()` sur chacun des quatre objets de la scène.

```
        bot1.dessiner(fenetre)
        bot2.dessiner(fenetre)
        iti1.dessiner(fenetre)
        iti2.dessiner(fenetre)
```

Documentation d'une classe

Comme pour une fonction, un commentaire (docstring) devrait documenter la classe, en particulier, ses variables d'objet ainsi que ses méthodes.

Le programme `ExempleAvecClasse` utilise les classes `BotAnime` et `ItiAnime`. On dit que `ExempleAvecClasse` fait appel aux *services* fournis par les classes `BotAnime` et `ItiAnime`. Tous les détails du fonctionnement des `Bot` et `Iti` sont cachés dans les classes `BotAnime` et `ItiAnime` du point de vue de `ExempleAvecClasse`.

Encapsulation, abstraction, interface programmatique (*Application Programming Interface* - API), service, client

Cette manière d'isoler les détails du fonctionnement d'une classe est une caractéristique de la programmation objet appelée l'*encapsulation*. Tout ce que l'utilisateur doit savoir, c'est comment appeler les méthodes appropriées de la classe `BotAnime` (le constructeur `BotAnime()`, `deplacer()`, `dessiner()`). Il n'a pas besoin de comprendre comment cela se passe à l'intérieur des méthodes appelées. Ainsi la classe `BotAnime` fournit une *abstraction* sous forme d'un ensemble de méthodes simples à appeler. Dans le langage objet, cet ensemble de méthodes est appelé une *interface programmatique*. On dit aussi que la classe `BotAnime` fournit un *service* à `ExempleAvecClasse` qui est le *client* de ce service. Le client voit l'interface programmatique mais pas l'implémentation.

Principes de génie logiciel : cohésion forte et couplage faible entre classes

Une question fondamentale au cœur de la conception d'un programme objet est la manière de répartir les variables et méthodes entre les classes. Deux principes fondamentaux sont à considérer : chercher à maximiser la cohésion des classes et à minimiser le couplage entre les classes. La *cohésion*

à l'intérieur d'une classe est forte lorsque les variables et méthodes de la classe sont fortement interreliées. Le couplage entre deux classes est *faible* lorsqu'il y a peu de dépendances entre les classes. Concrètement la dépendance entre classes est déterminée par la manière dont une classe utilise une autre classe, en passant par des déclarations, l'utilisation des variables, les appels de méthodes et le passage de paramètres.

Dans notre exemple, les méthodes `deplacer()` et `dessiner()` utilisent toutes une grande proportion des variables de la classe `BotAnime`, ce qui est un signe de grande cohésion.

D'autre part, le fait de passer le paramètre `fenetre` à `dessiner()` et, `LARGEUR_FENETRE`, `HAUTEUR_FENETRE` à `deplacer()` est un indice de couplage entre le programme principal de `ExempleAvecClasse` et `BotAnime`/`ItiAnime`. Minimiser le couplage signifie, entre autres, de chercher à réduire le nombre de paramètres lorsque cela est approprié. Par exemple, une possibilité serait de faire de `fenetre` et des variables `LARGEUR_FENETRE` et `HAUTEUR_FENETRE`, des variables d'objet des classes `BotAnime`/`ItiAnime`. Ceci évite de passer ces paramètres à chacun des appels à `deplacer()` et `dessiner()` tel qu'illustré par l'exemple suivant.

Dans cet exemple, la `fenetre` de dessin est initialisée par le constructeur de classe. La largeur et la hauteur de la `fenetre` sont extraits de l'objet `fenetre` par les méthodes `get_width()` et `get_height()`.

CodePython/chapitre7/ExempleFenetreVariableDObjet.py

```
"""
Exemple d'animation d'entités : création des classes BotAnime et ItiAnime
avec variable d'objet pour la fenetre
"""
# Importer la librairie de pygame et initialiser
import pygame
from pygame import Color

class BotAnime :
    """ Un objet représente un Bot qui est animé dans une fenêtre Pygame

    Le Bot est inscrit dans le rectangle englobant défini par r. Il se déplace en
diagonale selon la vitesse v.
        r : pygame.Rect       Le rectangle englobant
        v : [int,int]   Vitesse de déplacement selon les deux axes x et y
        f : pygame.Surface
    """

    def __init__(self,rectangle,vitesse,fenetre):
        self.r = rectangle
        self.v = vitesse
        self.f = fenetre

    def dessiner(self):
        """ Dessiner un Bot.

```

```
        Le Bot est inscrit dans le rectangle englobant défini par la variable d'objet r dans une fenetre de Pygame
        """

        pygame.draw.ellipse(self.f, Color('green'), ((self.r.x,self.r.y),(self.r.width, self.r.height/2))) # Dessiner la tête
        pygame.draw.rect(self.f, Color('black'), ((self.r.x+self.r.width/4,self.r.y+self.r.height/8),(self.r.width/10,self.r.height/20))) # L'oeil gauche
        pygame.draw.rect(self.f, Color('black'), ((self.r.x+self.r.width*3/4-self.r.width/10,self.r.y+self.r.height/8),(self.r.width/10,self.r.height/20))) # L'oeil droit
        pygame.draw.line(self.f, Color('black'), (self.r.x+self.r.width/4,self.r.y+self.r.height*3/8),(self.r.x+self.r.width*3/4,self.r.y+self.r.height*3/8), 2) # La bouche
        pygame.draw.rect(self.f, Color('red'), ((self.r.x,self.r.y+self.r.height/2),(self.r.width,self.r.height/2))) # Le corps


    def deplacer(self):
        """ Déplacer selon self.v en diagonale en rebondissant sur les bords de la fenetre"""
        if self.r.x+self.v[0] > self.f.get_width()-self.r.width or self.r.x+self.v[0] < 0 :
            self.v[0] = -self.v[0] # Inverser la direction en x
        self.r.x = self.r.x+self.v[0]
        if self.r.y+self.v[1] > self.f.get_height()-self.r.height or self.r.y+self.v[1] < 0 :
            self.v[1] = -self.v[1] # Inverser la direction en y
        self.r.y = self.r.y+self.v[1]

class ItiAnime :
    """ Un objet représente un Bot qui est animé dans une fenêtre Pygame

    Le Iti est inscrit dans le rectangle englobant défini par r. Il se déplace en diagonale selon la vitesse v.
        r : pygame.Rect       Le rectangle englobant
        v : [int,int]   Vitesse de déplacement selon les deux axes x et y
        f : pygame.Surface
    """

    def __init__(self,rectangle,vitesse,fenetre):
        self.r = rectangle
        self.v = vitesse
        self.f = fenetre

    def dessiner(self):
        """ Dessiner un Iti.

        Le Iti est inscrit dans le rectangle englobant défini par la variable d'objet r dans une fenetre de Pygame
        """
        self.milieux = self.r.x + self.r.width/2;
        self.milieuy = self.r.y + self.r.height/2;

        pygame.draw.ellipse(self.f, Color('pink'), ((self.r.x+self.r.width/3,self.r.y),(self.r.width/3,self.r.height/4))) # Dessiner la tête
        pygame.draw.arc(self.f,Color('black'),((self.milieux-self.r.width/12,self.r.y+self.r.height/8),(self.r.width/6,self.r.height/14)),3.1416,0,2) # Le sourire
        pygame.draw.ellipse(self.f, Color('black'), ((self.milieux-self.r.width/8,self.r.y+self.r.height/12),(self.r.width/12,self.r.height/24))) # L'oeil gauche
        pygame.draw.ellipse(self.f, Color('black'), ((self.milieux+self.r.width/8-self.r.width/12,self.r.y+self.r.height/12),(self.r.width/12,self.r.height/24))) # L'oeil droit
        pygame.draw.line(self.f, Color('black'), (self.milieux,self.r.y+self.r.height/4),(self.milieux,self.r.y+self.r.height*3/4), 2) # Le corps
        pygame.draw.line(self.f, Color('black'), (self.r.x,self.r.y+self.r.height/4),(self.milieux,self.milieuy), 2) # Bras gauche
        pygame.draw.line(self.f, Color('black'), (self.r.x+self.r.width,self.r.y+self.r.height/4),(self.milieux,self.milieuy), 2) # Bras droit
        pygame.draw.line(self.f, Color('black'), (self.r.x,self.r.y+self.r.height),(self.milieux,self.r.y+self.r.height*3/4), 2) # Jambe gauche
        pygame.draw.line(self.f, Color('black'), (self.r.x+self.r.width,self.r.y+self.r.height),(self.milieux,self.r.y+self.r.height*3/4), 2) # Jambe droite
```

```
76.
77.     def deplacer(self):
78.         """ Déplacer selon self.v en diagonale en rebondissant sur les bords de la
fenetre"""
79.         if self.r.x+self.v[0] > self.f.get_width()-self.r.width or self.r.x+self.v[0] < 0
:
80.             self.v[0] = -self.v[0] # Inverser la direction en x
81.         self.r.x = self.r.x+self.v[0]
82.         if self.r.y+self.v[1] > self.f.get_height()-self.r.height or self.r.y+self.v[1] <
0 :
83.             self.v[1] = -self.v[1] # Inverser la direction en y
84.         self.r.y = self.r.y+self.v[1]
85.
86. pygame.init() # Initialiser les modules de Pygame
87. LARGEUR_FENETRE = 400
88. HAUTEUR_FENETRE = 600
89. fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre
90. pygame.display.set_caption("Exemple des Bots et Itis en diagonale : fenetre variable
d'objet") # Définir le titre dans le haut de la fenêtre
91.
92. horloge = pygame.time.Clock() # Pour contrôler la fréquence des scènes
93.
94. # Création de deux BotAnime et deux ItiAnime
95. bot1 = BotAnime(pygame.Rect((0,0),(20,40)),[5,10],fenetre)
96. bot2 = BotAnime(pygame.Rect((100,200),(30,60)),[0,2],fenetre)
97. iti1 = ItiAnime(pygame.Rect((200,150),(40,80)),[3,3],fenetre)
98. iti2 = ItiAnime(pygame.Rect((300,300),(50,100)),[5,10],fenetre)
99.
100. # Boucle d'animation
101. fin = False
102. while not fin :
103.     event = pygame.event.poll() # Chercher le prochain évènement à traiter
104.     if event.type == pygame.QUIT:  # Utilisateur a cliqué sur la fermeture de fenêtre ?
105.         fin = True  # Fin de la boucle du jeu
106.     else :
107.         bot1.deplacer()
108.         bot2.deplacer()
109.         iti1.deplacer()
110.         iti2.deplacer()
111.
112.         fenetre.fill(Color('white')) # Dessiner le fond de la surface de dessin
113.         bot1.dessiner()
114.         bot2.dessiner()
115.         iti1.dessiner()
116.         iti2.dessiner()
117.
118.         pygame.display.flip() # Mettre à jour la fenêtre graphique
119.
120.         horloge.tick(60) # Pour animer avec 60 images pas seconde
121.
122. pygame.quit() # Terminer pygame
```

Exercice. Créez une classe `Contact` pour gérer vos contacts téléphoniques. Un contact a trois variables d'objet : `nom`, `prenom` et `numero_telephone`. Le constructeur crée un nouveau contact en spécifiant ces trois données. Définissez la méthode spéciale `__str__()` qui produit une représentation sous forme de chaîne de caractère pour l'affichage. Créez trois objets `Contact` en les ajoutant à une liste de contacts. Affichez tous les contacts de la liste un par un avec `print()`.

Solution : CodePython/chapitre7/ExerciceClasseContact.py

```
1. """
2. Exercice classe Contact
3. """
4.
```

```
 5. class Contact :
 6.     """ Un objet représente un contact
 7.
 8.         nom : str
 9.         prenom : str
10.         numero_telephone : str
11.     """
12.
13.     def __init__(self,nom,prenom,numero_telephone):
14.         self.nom = nom
15.         self.prenom = prenom
16.         self.numero_telephone = numero_telephone
17.
18.     def __str__(self):
19.         return 'Le numero de téléphone de '+self.prenom+'
'+self.nom+' est :'+self.numero_telephone
20.
21. liste_contacts = []
22. liste_contacts.append(Contact('Binette','Bob','333-333-3333'))
23. liste_contacts.append(Contact('Emerson','Keith','111-111-
1111'))
24. liste_contacts.append(Contact('Anderson','Ian','222-222-2222'))
25.
26. for un_contact in liste_contacts:
27.     print(un_contact)
```

Résultat :

```
Le numéro de téléphone de Bob Binette est :333-333-3333
Le numéro de téléphone de Keith Emerson est :111-111-1111
Le numéro de téléphone de Ian Anderson est :222-222-2222
```

Une particularité de Python est la possibilité de contourner l'encapsulation d'un objet en accédant directement aux variables d'objet. Dans l'exemple de code suivant, le numéro de téléphone du contact est modifié directement de l'extérieur de la classe en accédant la variable d'objet `numero_telephone` :

CodePython/chapitre7/ExempleVariableDobjetPublique.py

```
 1. """
 2. Exemple classe Contact
 3. Variable d'objet publique
 4. """
 5.
 6. class Contact :
 7.     """ Un objet représente un contact
 8.
 9.         nom : str
10.         prenom : str
11.         numero_telephone : str
12.     """
```

```
13.
14.     def __init__(self,nom,prenom,numero_telephone):
15.         self.nom = nom
16.         self.prenom = prenom
17.         self.numero_telephone = numero_telephone
18.
19.     def __str__(self):
20.         return 'Le numéro de téléphone de '+self.prenom+'
'+self.nom+' est :'+self.numero_telephone
21.
22. contact1 = Contact('Binette','Bob','333-333-3333')
23. print(contact1)
24. contact1.numero_telephone = '444-444-4444'
25. print(contact1)
26.
```

Résultat :

```
Le numéro de téléphone de Bob Binette est :333-333-3333
Le numéro de téléphone de Bob Binette est :444-444-4444
```

Convention pour variable privée

Pour désigner une variable d'objet qui devrait être privée, la convention vue pour les méthodes qui consiste à précéder son nom d'un soulignement _, peut aussi être employée.

La possibilité d'accéder aux variables d'objet directement doit être exploitée avec prudence parce qu'elle permet d'effectuer des manipulations qui n'ont pas été prévues au moment de définir la classe. Ceci peut conduire à une plus grande complexité du logiciel et à des bogues difficiles à détecter. Python permet même d'ajouter de nouvelles variables d'objet à des objets existants. Le résultat est que différents objets de la même classe peuvent avoir des variables d'objets différentes. La cacophonie résultante peut devenir déconcertante !

Dans l'exemple suivant, une nouvelle variable d'objet `age` est ajoutée à un contact. La documentation de la classe ne reflète pas cette possibilité. Lorsque l'objet est imprimé, cette nouvelle variable n'apparaît pas. Ceci peut entraîner une certaine confusion.

CodePython/chapitre7/ExempleAjoutVariableDobjet.py

```
1. """
2. Exemple classe Contact
3. Ajout d'une variable d'objet à un objet existant
4. """
5. class Contact :
```

```
 6.      """ Un objet représente un contact
 7.
 8.          nom : str
 9.          prenom : str
10.          numero_telephone : str
11.      """
12.
13.      def __init__(self,nom,prenom,numero_telephone):
14.          self.nom = nom
15.          self.prenom = prenom
16.          self.numero_telephone = numero_telephone
17.
18.      def __str__(self):
19.          return 'Le numéro de téléphone de '+self.prenom+'
'+self.nom+' est :'+self.numero_telephone
20. print(contact1)
21. contact1.age= 52
22. print(contact1)
23.
```

Résultat :

```
Le numéro de téléphone de Bob Binette est :333-333-3333
Le numéro de téléphone de Bob Binette est :333-333-3333
```

7.2 Variable et méthode de classe

Un aspect important à noter dans la solution de la section précédente pour l'animation est le fait que les variables d'objet, `fenetre`, `largeur_fenetre` et `hauteur_fenetre`, sont répétées dans chacun des objets `BotAnime` et `ItiAnime`, même si les valeurs sont identiques pour tous les objets de la même classe. Généralement, le fait de répéter la même information à plusieurs endroits est à éviter. Pour limiter la répétition des données entre les objets, il est possible d'en faire des *variables de classe* (*variable statique*). Par opposition à une variable d'objet, il n'y a qu'une variable de classe commune pour tous les objets d'une classe. La variable de classe est ainsi partagée par tous les objets de la classe.

L'exemple suivant crée des variables de classes pour la fenêtre, sa hauteur et sa largeur, dans les deux classes `BotAnime` et `ItiAnime`.

CodePython/chapitre7/ExempleVariableDeClasse.py

```
 1. """
 2. Exemple d'animation d'entités : création des classes BotAnime et ItiAnime
 3. avec variable de classe pour la fenetre
 4. """
 5. # Importer la librairie de pygame et initialiser
 6. import pygame
 7. from pygame import Color
 8.
 9. class BotAnime :
10.     """ Un objet représente un Bot qui est animé dans une fenêtre Pygame
```

```
 11.
 12.     Le Bot est inscrit dans le rectangle englobant défini par r. Il se déplace en
diagonale selon la vitesse v.
 13.         r : pygame.Rect       Le rectangle englobant
 14.         v : [int,int]         Vitesse de déplacement selon les deux axes x et y
 15.     """
 16.
 17.     @staticmethod
 18.     def set_fenetre(fenetre):
 19.         """ Fixer la variable de classe f qui représente la fenetre graphique
 20.
 21.             fenetre : pygame.Surface
 22.         """
 23.         BotAnime.f = fenetre
 24.
 25.     def __init__(self,rectangle,vitesse):
 26.         self.r = rectangle
 27.         self.v = vitesse
 28.
 29.     def dessiner(self):
 30.         """ Dessiner un Bot.
 31.
 32.         Le Bot est inscrit dans le rectangle englobant défini par la variable d'objet r
dans une fenetre de Pygame
 33.         """
 34.
 35.         pygame.draw.ellipse(BotAnime.f, Color('green'),
((self.r.x,self.r.y),(self.r.width, self.r.height/2))) # Dessiner la tête
 36.         pygame.draw.rect(BotAnime.f, Color('black'),
((self.r.x+self.r.width/4,self.r.y+self.r.height/8),(self.r.width/10,self.r.height/20))) #
L'oeil gauche
 37.         pygame.draw.rect(BotAnime.f, Color('black'), ((self.r.x+self.r.width*3/4-
self.r.width/10,self.r.y+self.r.height/8),(self.r.width/10,self.r.height/20))) # L'oeil droit
 38.         pygame.draw.line(BotAnime.f, Color('black'),
(self.r.x+self.r.width/4,self.r.y+self.r.height*3/8),(self.r.x+self.r.width*3/4,self.r.y+self.r
.height*3/8), 2) # La bouche
 39.         pygame.draw.rect(BotAnime.f, Color('red'),
((self.r.x,self.r.y+self.r.height/2),(self.r.width,self.r.height/2))) # Le corps
 40.
 41.
 42.     def deplacer(self):
 43.         """ Déplacer selon self.v en diagonale en rebondissant sur les bords de la
fenetre"""
 44.         if self.r.x+self.v[0] > BotAnime.f.get_width()-self.r.width or self.r.x+self.v[0]
< 0 :
 45.             self.v[0] = -self.v[0] # Inverser la direction en x
 46.         self.r.x = self.r.x+self.v[0]
 47.         if self.r.y+self.v[1] > BotAnime.f.get_height()-self.r.height or
self.r.y+self.v[1] < 0 :
 48.             self.v[1] = -self.v[1] # Inverser la direction en y
 49.         self.r.y = self.r.y+self.v[1]
 50.
 51. class ItiAnime :
 52.     """ Un objet représente un Bot qui est animé dans une fenêtre Pygame
 53.
 54.     Le Bot est inscrit dans le rectangle englobant défini par r. Il se déplace en
diagonale selon vitesse.
 55.         r : pygame.Rect       Le rectangle englobant
 56.         vitesse : [int,int]   Vitesse de déplacement selon les deux axes x et y
 57.         fenetre : pygame.Surface
 58.         taille_fenetre : (int,int)
 59.     """
 60.     @staticmethod
 61.     def set_fenetre(fenetre):
 62.         """ Fixer la variable de classe f qui représente la fenetre graphique
 63.
 64.             fenetre : pygame.Surface
 65.         """
 66.         ItiAnime.f = fenetre
 67.
 68.     def __init__(self,rectangle,vitesse):
 69.         self.r = rectangle
 70.         self.v = vitesse
 71.
 72.     def dessiner(self):
```

```
 73.         """ Dessiner un Iti.
 74.
 75.         Le Iti est inscrit dans le rectangle englobant défini par la variable d'objet r
dans une fenetre de Pygame
 76.         """
 77.         self.milieux = self.r.x + self.r.width/2;
 78.         self.milieuy = self.r.y + self.r.height/2;
 79.
 80.         pygame.draw.ellipse(ItiAnime.f, Color('pink'),
((self.r.x+self.r.width/3,self.r.y),(self.r.width/3,self.r.height/4))) # Dessiner la tête
 81.         pygame.draw.arc(ItiAnime.f,Color('black'),((self.milieux-
self.r.width/12,self.r.y+self.r.height/8),(self.r.width/6,self.r.height/14)),3.1416,0,2) # Le
sourire
 82.         pygame.draw.ellipse(ItiAnime.f, Color('black'), ((self.milieux-
self.r.width/8,self.r.y+self.r.height/12),(self.r.width/12,self.r.height/24))) # L'oeil gauche
 83.         pygame.draw.ellipse(ItiAnime.f, Color('black'), ((self.milieux+self.r.width/8-
self.r.width/12,self.r.y+self.r.height/12),(self.r.width/12,self.r.height/24))) # L'oeil droit
 84.         pygame.draw.line(ItiAnime.f, Color('black'),
(self.milieux,self.r.y+self.r.height/4),(self.milieux,self.r.y+self.r.height*3/4), 2) # Le
corps
 85.         pygame.draw.line(ItiAnime.f, Color('black'),
(self.r.x,self.r.y+self.r.height/4),(self.milieux,self.milieuy), 2) # Bras gauche
 86.         pygame.draw.line(ItiAnime.f, Color('black'),
(self.r.x+self.r.width,self.r.y+self.r.height/4),(self.milieux,self.milieuy), 2) # Bras droit
 87.         pygame.draw.line(ItiAnime.f, Color('black'),
(self.r.x,self.r.y+self.r.height),(self.milieux,self.r.y+self.r.height*3/4), 2) # Jambe gauche
 88.         pygame.draw.line(ItiAnime.f, Color('black'),
(self.r.x+self.r.width,self.r.y+self.r.height),(self.milieux,self.r.y+self.r.height*3/4), 2) #
Jambe droite
 89.
 90.     def deplacer(self):
 91.         """ Déplacer selon self.v en diagonale en rebondissant sur les bords de la
fenetre"""
 92.         if self.r.x+self.v[0] > ItiAnime.f.get_width()-self.r.width or self.r.x+self.v[0]
< 0 :
 93.             self.v[0] = -self.v[0] # Inverser la direction en x
 94.         self.r.x = self.r.x+self.v[0]
 95.         if self.r.y+self.v[1] > ItiAnime.f.get_height()-self.r.height or
self.r.y+self.v[1] < 0 :
 96.             self.v[1] = -self.v[1] # Inverser la direction en y
 97.         self.r.y = self.r.y+self.v[1]
 98.
 99. pygame.init() # Initialiser les modules de Pygame
100. LARGEUR_FENETRE = 400
101. HAUTEUR_FENETRE = 600
102. fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre
103. BotAnime.set_fenetre(fenetre)
104. ItiAnime.set_fenetre(fenetre)
105. pygame.display.set_caption("Exemple des Bots et Itis en diagonale : fenetre variable
d'objet") # Définir le titre dans le haut de la fenêtre
106.
107. horloge = pygame.time.Clock() # Pour contrôler la fréquence des scènes
108.
109. # Création de deux BotAnime et deux ItiAnime
110. bot1 = BotAnime(pygame.Rect((0,0),(20,40)),[5,10])
111. bot2 = BotAnime(pygame.Rect((100,200),(30,60)),[0,2])
112. iti1 = ItiAnime(pygame.Rect((200,150),(40,80)),[3,3])
113. iti2 = ItiAnime(pygame.Rect((300,300),(50,100)),[5,10])
114.
115. # Boucle d'animation
116. fin = False
117. while not fin :
118.     event = pygame.event.poll() # Chercher le prochain évènement à traiter
119.     if event.type == pygame.QUIT:  # Utilisateur a cliqué sur la fermeture de fenêtre ?
120.         fin = True  # Fin de la boucle du jeu
121.     else :
122.         bot1.deplacer()
123.         bot2.deplacer()
124.         iti1.deplacer()
125.         iti2.deplacer()
126.
127.         fenetre.fill(Color('white')) # Dessiner le fond de la surface de dessin
128.         bot1.dessiner()
129.         bot2.dessiner()
130.         iti1.dessiner()
```

```
131.         iti2.dessiner()
132.
133.         pygame.display.flip() # Mettre à jour la fenêtre graphique
134.
135.         horloge.tick(60) # Pour animer avec 60 images pas seconde
136.
137. pygame.quit() # Terminer pygame
```

Pour fixer les valeurs des variables de classe, la méthode de classe `set_fenetre()` est définie :

```
    @staticmethod
    def set_fenetre(fenetre):
        """ Fixer la variable de classe f qui représente la fenetre graphique

            fenetre : pygame.Surface
        """
        BotAnime.f = fenetre
```

Le décorateur `@staticmethod` est employé en Python pour désigner une méthode de classe (*méthode statique*). Par opposition à une méthode d'objet, une méthode de classe est appelée sur la classe et non pas sur un objet. Une méthode de classe n'a pas accès aux variables d'objet mais uniquement aux variables de classe. La notation `NomClasse.variable_de_classe` permet d'accéder à une variable de classe :

```
        BotAnime.f = fenetre
```

Après avoir créé l'objet `fenetre`, il est passé à la méthode de classe `set_fenetre()` pour initialiser les variables de classe :

```
BotAnime.set_fenetre(fenetre)
```

Un dernier cas de répétition demeure malgré l'emploi de variables de classes, parce qu'elles sont répétées dans les deux classes `BotAnime` et `ItiAnime`. La section suivante montre comment l'utilisation d'une super classe permet d'éliminer cette redondance ainsi que d'autres formes de répétitions de code.

7.3 Hiérarchie de classes, héritage, sous-classe, super-classe

La création d'une super-classe permet d'éviter de répéter les éléments communs à un ensemble de classes en créant une classe plus abstraite. À l'inverse, la définition d'une sous-classe permet de définir des objets plus spécialisés qui ajoutent des comportements plus spécifiques à ceux de la super-classe.

Dans l'exemple précédent, il y a beaucoup de répétition de code entre les deux classes `BotAnime` et `ItiAnime`. Les variables sont les mêmes ainsi que les méthodes `set_fenetre()`, `__init__()` et `deplacer()`. Une

manière d'éviter cette redondance de code est de créer une super-classe qui met en facteur les aspects communs des deux classes dans la super-classe commune. La super-classe `EntiteAnimee` déclare les variables et méthodes communes aux deux classes `BotAnime` et `ItiAnime` de l'exemple précédent.

CodePython/chapitre7/ExempleSuperClasse.py

```
"""
Exemple d'animation d'entités : exemple de super-classe EntiteAnimee
"""
import pygame
from pygame import Color

class EntiteAnimee :
    """ Un objet représente une entité qui est animée dans une fenêtre Pygame

    L'entité est inscrite dans le rectangle englobant défini par r. Il se déplace en
diagonale selon la vitesse v.
        r : pygame.Rect       Le rectangle englobant
        v : [int,int]         Vitesse de déplacement selon les deux axes x et y
    """

    @staticmethod
    def set_fenetre(fenetre):
        """ Fixer la variable de classe f qui représente la fenetre graphique

            fenetre : pygame.Surface
        """
        EntiteAnimee.f = fenetre

    def __init__(self,rectangle,vitesse):
        self.r = rectangle
        self.v = vitesse

    def deplacer(self):
        """ Déplacer selon self.v en diagonale en rebondissant sur les bords de la
fenetre"""
        if self.r.x+self.v[0] > EntiteAnimee.f.get_width()-self.r.width or
self.r.x+self.v[0] < 0 :
            self.v[0] = -self.v[0] # Inverser la direction en x
        self.r.x = self.r.x+self.v[0]
        if self.r.y+self.v[1] > EntiteAnimee.f.get_height()-self.r.height or
self.r.y+self.v[1] < 0 :
            self.v[1] = -self.v[1] # Inverser la direction en y
        self.r.y = self.r.y+self.v[1]

class BotAnime(EntiteAnimee) :
    """ Un objet représente un Bot qui est animé dans une fenêtre Pygame
        Sous-classe de EntiteAnimee
    """

    def dessiner(self):
        """ Dessiner un Bot.

        Le Bot est inscrit dans le rectangle englobant défini par la variable d'objet r
dans une fenetre de Pygame
        """

        pygame.draw.ellipse(BotAnime.f, Color('green'),
((self.r.x,self.r.y),(self.r.width, self.r.height/2))) # Dessiner la tête
        pygame.draw.rect(BotAnime.f, Color('black'),
((self.r.x+self.r.width/4,self.r.y+self.r.height/8),(self.r.width/10,self.r.height/20))) #
L'oeil gauche
        pygame.draw.rect(BotAnime.f, Color('black'), ((self.r.x+self.r.width*3/4-
self.r.width/10,self.r.y+self.r.height/8),(self.r.width/10,self.r.height/20))) # L'oeil droit
        pygame.draw.line(BotAnime.f, Color('black'),
(self.r.x+self.r.width/4,self.r.y+self.r.height*3/8),(self.r.x+self.r.width*3/4,self.r.y+self.r
.height*3/8), 2) # La bouche
```

```
51.         pygame.draw.rect(BotAnime.f, Color('red'),
((self.r.x,self.r.y+self.r.height/2),(self.r.width,self.r.height/2))) # Le corps
52.
53.
54. class ItiAnime(EntiteAnimee) :
55.     """ Un objet représente un Bot qui est animé dans une fenêtre Pygame
56.         Sous-classe de EntiteAnimee
57.     """
58.
59.     def dessiner(self):
60.         """ Dessiner un Iti.
61.
62.         Le Iti est inscrit dans le rectangle englobant défini par la variable d'objet r
dans une fenetre de Pygame
63.         """
64.         self.milieux = self.r.x + self.r.width/2;
65.         self.milieuy = self.r.y + self.r.height/2;
66.
67.         pygame.draw.ellipse(ItiAnime.f, Color('pink'),
((self.r.x+self.r.width/3,self.r.y),(self.r.width/3,self.r.height/4))) # Dessiner la tête
68.         pygame.draw.arc(ItiAnime.f,Color('black'),((self.milieux-
self.r.width/12,self.r.y+self.r.height/8),(self.r.width/6,self.r.height/14)),3.1416,0,2) # Le
sourire
69.         pygame.draw.ellipse(ItiAnime.f, Color('black'), ((self.milieux-
self.r.width/8,self.r.y+self.r.height/12),(self.r.width/12,self.r.height/24))) # L'oeil gauche
70.         pygame.draw.ellipse(ItiAnime.f, Color('black'), ((self.milieux+self.r.width/8-
self.r.width/12,self.r.y+self.r.height/12),(self.r.width/12,self.r.height/24))) # L'oeil droit
71.         pygame.draw.line(ItiAnime.f, Color('black'),
(self.milieux,self.r.y+self.r.height/4),(self.milieux,self.r.y+self.r.height*3/4), 2) # Le
corps
72.         pygame.draw.line(ItiAnime.f, Color('black'),
(self.r.x,self.r.y+self.r.height/4),(self.milieux,self.milieuy), 2) # Bras gauche
73.         pygame.draw.line(ItiAnime.f, Color('black'),
(self.r.x+self.r.width,self.r.y+self.r.height/4),(self.milieux,self.milieuy), 2) # Bras droit
74.         pygame.draw.line(ItiAnime.f, Color('black'),
(self.r.x,self.r.y+self.r.height),(self.milieux,self.r.y+self.r.height*3/4), 2) # Jambe gauche
75.         pygame.draw.line(ItiAnime.f, Color('black'),
(self.r.x+self.r.width,self.r.y+self.r.height),(self.milieux,self.r.y+self.r.height*3/4), 2) #
Jambe droite
76.
77. pygame.init() # Initialiser les modules de Pygame
78. LARGEUR_FENETRE = 400
79. HAUTEUR_FENETRE = 600
80. fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre
81. EntiteAnimee.set_fenetre(fenetre)
82. pygame.display.set_caption("Exemple des Bots et Itis animés en diagonale avec super-classe
EntiteAnimee") # Définir le titre dans le haut de la fenêtre
83.
84. horloge = pygame.time.Clock() # Pour contrôler la fréquence des scènes
85.
86. # Création de deux BotAnime et deux ItiAnime
87. bot1 = BotAnime(pygame.Rect((0,0),(20,40)),[5,10])
88. bot2 = BotAnime(pygame.Rect((100,200),(30,60)),[0,2])
89. iti1 = ItiAnime(pygame.Rect((200,150),(40,80)),[3,3])
90. iti2 = ItiAnime(pygame.Rect((300,300),(50,100)),[5,10])
91.
92. # Boucle d'animation
93. fin = False
94. while not fin :
95.     event = pygame.event.poll() # Chercher le prochain évènement à traiter
96.     if event.type == pygame.QUIT:  # Utilisateur a cliqué sur la fermeture de fenêtre ?
97.         fin = True  # Fin de la boucle du jeu
98.     else :
99.         bot1.deplacer()
100.         bot2.deplacer()
101.         iti1.deplacer()
102.         iti2.deplacer()
103.
104.         fenetre.fill(Color('white')) # Dessiner le fond de la surface de dessin
105.         bot1.dessiner()
106.         bot2.dessiner()
107.         iti1.dessiner()
108.         iti2.dessiner()
109.
110.         pygame.display.flip() # Mettre à jour la fenêtre graphique
```

```
111.
112.         horloge.tick(60) # Pour animer avec 60 images pas seconde
113.
114. pygame.quit() # Terminer pygame
115.
```

Dans la définition d'une nouvelle classe, le nom entre parenthèses spécifie la super-classe. Il est permis d'en avoir plusieurs qui seront alors séparées par des virgules. Dans la ligne suivante, la classe `BotAnime` est définie comme une *sous-classe* de la super-classe `EntiteAnimee`.

```
class BotAnime(EntiteAnimee) :
```

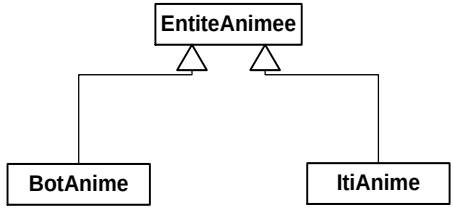


Figure 12. Représentation des relations de sous-classes en UML.

Sous-classe, super-classe, héritage, généralisation/spécialisation

La figure précédente montre un diagramme de classe UML qui représente la hiérarchie des classes. Un tel diagramme est utile pour comprendre l'organisation des classes d'un programme. Chacune des classes est représentée par un rectangle. La flèche représente une relation de sous-classe (aussi appelée relation d'*héritage* ou encore de *généralisation/spécialisation*). En définissant une classe X comme une sous-classe d'une autre classe Y, la classe X hérite des méthodes et variables définies dans la classe Y. La sous-classe X devient une spécialisation de Y et inversement Y est une généralisation par rapport à X.

Ainsi, en définissant la classe `BotAnime` comme une sous-classe de `EntiteAnimee`, la classe `BotAnime` hérite des méthodes et variables définies dans `EntiteAnimee`. Ceci entraîne qu'un objet de la classe `BotAnime` est aussi considéré comme un objet de la super-classe `EntiteAnimee`. Les méthodes d'objet héritées de la classe `EntiteAnimee` peuvent être appelées sur un objet de la classe `BotAnime` comme si elles y avaient été définies. De la même façon, les variables d'objet de la classe `EntiteAnimee` peuvent être accédées dans la classe `BotAnime` comme si elles y avaient été définies. Les méthodes de classe de la super-classe peuvent être appelées sur une sous-classe et les variables de classe de la super-classe peuvent être utilisées dans la sous-classe. Le mécanisme d'héritage permet ainsi de réutiliser les variables et méthodes de la classe

`EntiteAnimee` dans la classe `BotAnime` sans avoir à les répéter dans le code de `BotAnime`.

Classe *Object*, racine de la hiérarchie des classes

Lorsqu'une nouvelle classe est définie sans spécifier de superclasse, elle est automatiquement une sous-classe de la classe prédéfinie `Object`. Cette classe définit plusieurs opérations standards pour toutes les classes Python dont plusieurs méthodes spéciales telles que `__str__()`, `__repr__()`, `__hash__()`, etc. Ces méthodes définissent certains comportements de base pour tous les objets.

Syntaxiquement, les deux définitions suivantes sont équivalentes :

```
class EntiteAnimee :
class EntiteAnimee(Object) :
```

Lorsque le constructeur est appelé pour créer un objet `BotAnime` dans la ligne suivante, c'est la méthode `__init__()` définie dans la super-classe `EntiteAnimee` qui est employée :

```
bot1 = BotAnime(pygame.Rect((0,0),(20,40)),[5,10])
```

L'appel de la méthode `deplacer()` sur l'objet `bot1` de la classe `BotAnime` utilise la méthode définie dans la super-classe `EntiteAnimee` :

```
        bot1.deplacer()
```

En revanche, l'appel de la méthode `dessiner()` sur l'objet `bot1` de la classe `BotAnime` utilise la méthode définie dans la classe `BotAnime`. Il est ainsi possible d'utiliser des méthodes héritées ainsi que des méthodes locales définies dans la classe de l'objet.

Dans la méthode `dessiner()` définie dans la sous-classe `BotAnime`, la variable de classe `BotAnime.f` correspond à celle qui a été définie dans la super-classe `EntiteAnimee`. La variable d'objet `self.r` correspond à la variable d'objet `r` définie dans la super-classe :

```
        pygame.draw.ellipse(BotAnime.f,Color('green'),
((self.r.x,self.r.y),(self.r.width, self.r.height/2))) # Dessiner la tête
```

Conceptuellement, la super-classe définie une abstraction qui peut être raffinée dans les sous-classes en ajoutant des éléments ou encore en redéfinissant des éléments. La redéfinition est expliquée plus loin. Cette manière de programmer en créant une sous-classe d'une classe existante en réutilisant son code tout en ajoutant un comportement plus spécialisé est

une approche typique et très puissante de la programmation objet. Dans notre exemple, ceci nous permet de créer des entités animées spécialisées qui héritent des caractéristiques plus générales de la classe `EntiteAnimee`.

Classe abstraite, classe concrète

Une classe dite *abstraite* ne peut être utilisée directement pour la création d'objet. Elle n'est utile que comme super-classe d'autres *classes* non abstraites dites *concrètes* qui, elles, sont utilisées pour créer des objets. Il peut être utile de définir des méthodes de classes abstraites qui n'ont pas de corps pour obliger une sous-classe à implémenter la méthode dite *abstraite* de la super-classe. Il est possible de spécifier qu'une classe est abstraite en Python avec le module ABC (voir PEP 3119 -- Introducing Abstract Base Classes | Python.org).

La classe `EntiteAnimee` de notre exemple est abstraite car elle ne peut servir à créer un objet. Ce sont les sous-classes, `BotAnime` et `ItiAnime` qui doivent être employées à cet effet.

Refactorisation de code (*refactoring*)

Dans l'exemple *précédent*, le code de `ExempleVariableDeClasse` a été transformé en déplaçant des méthodes et variables dans une super-classe mais sans en changer le comportement. Ce genre de transformation qui préserve le même comportement en améliorant la qualité du code est appelé une *refactorisation* du code (*refactoring*).

Dans le code précédent, il faut appeler à répétition la méthode `deplacer()` et ensuite `dessiner()` sur chacune des entités à animer. Il est possible de simplifier encore le code en plaçant les entités dans une liste et en appelant la méthode dans une boucle `for` tel qu'illustré par le code surligné dans l'exemple suivant qui refactorise le code de `ExempleSuperClasse.py`.

CodePython/chapitre7/ExempleSurchargeDynamique.py

```
"""
Exemple d'animation d'entités : création d'une super classe EntiteAnime et
appel des méthodes d'animation des entités par itération et surcharge dynamique
"""
import pygame
from pygame import Color

class EntiteAnimee :
    """ Un objet représente une entité qui est animée dans une fenêtre Pygame

    L'entité est inscrite dans le rectangle englobant défini par r. Il se déplace en
diagonale selon la vitesse v.
        r : pygame.Rect       Le rectangle englobant
```

```
13.         v : [int,int]         Vitesse de déplacement selon les deux axes x et y
14.     """
15.
16.     @staticmethod
17.     def set_fenetre(fenetre):
18.         """ Fixer la variable de classe f qui représente la fenetre graphique
19.
20.             fenetre : pygame.Surface
21.         """
22.         EntiteAnimee.f = fenetre
23.
24.     def __init__(self,rectangle,vitesse):
25.         self.r = rectangle
26.         self.v = vitesse
27.
28.     def deplacer(self):
29.         """ Déplacer selon self.v en diagonale en rebondissant sur les bords de la
fenetre"""
30.         if self.r.x+self.v[0] > EntiteAnimee.f.get_width()-self.r.width or
self.r.x+self.v[0] < 0 :
31.             self.v[0] = -self.v[0] # Inverser la direction en x
32.         self.r.x = self.r.x+self.v[0]
33.         if self.r.y+self.v[1] > EntiteAnimee.f.get_height()-self.r.height or
self.r.y+self.v[1] < 0 :
34.             self.v[1] = -self.v[1] # Inverser la direction en y
35.         self.r.y = self.r.y+self.v[1]
36.
37. class BotAnime(EntiteAnimee) :
38.     """ Un objet représente un Bot qui est animé dans une fenêtre Pygame
39.         Sous-classe de EntiteAnimee
40.     """
41.
42.     def dessiner(self):
43.         """ Dessiner un Bot.
44.
45.         Le Bot est inscrit dans le rectangle englobant défini par la variable d'objet r
dans une fenetre de Pygame
46.         """
47.
48.         pygame.draw.ellipse(BotAnime.f, Color('green'),
((self.r.x,self.r.y),(self.r.width, self.r.height/2))) # Dessiner la tête
49.         pygame.draw.rect(BotAnime.f, Color('black'),
((self.r.x+self.r.width/4,self.r.y+self.r.height/8),(self.r.width/10,self.r.height/20))) #
L'oeil gauche
50.         pygame.draw.rect(BotAnime.f, Color('black'), ((self.r.x+self.r.width*3/4-
self.r.width/10,self.r.y+self.r.height/8),(self.r.width/10,self.r.height/20))) # L'oeil droit
51.         pygame.draw.line(BotAnime.f, Color('black'),
(self.r.x+self.r.width/4,self.r.y+self.r.height*3/8),(self.r.x+self.r.width*3/4,self.r.y+self.r
.height*3/8), 2) # La bouche
52.         pygame.draw.rect(BotAnime.f, Color('red'),
((self.r.x,self.r.y+self.r.height/2),(self.r.width,self.r.height/2))) # Le corps
53.
54.
55. class ItiAnime(EntiteAnimee) :
56.     """ Un objet représente un Bot qui est animé dans une fenêtre Pygame
57.         Sous-classe de EntiteAnimee
58.     """
59.
60.     def dessiner(self):
61.         """ Dessiner un Iti.
62.
63.         Le Iti est inscrit dans le rectangle englobant défini par la variable d'objet r
dans une fenetre de Pygame
64.         """
65.         self.milieux = self.r.x + self.r.width/2;
66.         self.milieuy = self.r.y + self.r.height/2;
67.
68.         pygame.draw.ellipse(ItiAnime.f, Color('pink'),
((self.r.x+self.r.width/3,self.r.y),(self.r.width/3,self.r.height/4))) # Dessiner la tête
69.         pygame.draw.arc(ItiAnime.f,Color('black'),((self.milieux-
self.r.width/12,self.r.y+self.r.height/8),(self.r.width/6,self.r.height/14)),3.1416,0,2) # Le
sourire
70.         pygame.draw.ellipse(ItiAnime.f, Color('black'), ((self.milieux-
self.r.width/8,self.r.y+self.r.height/12),(self.r.width/12,self.r.height/24))) # L'oeil gauche
```

```
 71.         pygame.draw.ellipse(ItiAnime.f, Color('black'), ((self.milieux+self.r.width/8-
self.r.width/12,self.r.y+self.r.height/12),(self.r.width/12,self.r.height/24))) # L'oeil droit
 72.         pygame.draw.line(ItiAnime.f, Color('black'),
(self.milieux,self.r.y+self.r.height/4),(self.milieux,self.r.y+self.r.height*3/4), 2) # Le
corps
 73.         pygame.draw.line(ItiAnime.f, Color('black'),
(self.r.x,self.r.y+self.r.height/4),(self.milieux,self.milieuy), 2) # Bras gauche
 74.         pygame.draw.line(ItiAnime.f, Color('black'),
(self.r.x+self.r.width,self.r.y+self.r.height/4),(self.milieux,self.milieuy), 2) # Bras droit
 75.         pygame.draw.line(ItiAnime.f, Color('black'),
(self.r.x,self.r.y+self.r.height),(self.milieux,self.r.y+self.r.height*3/4), 2) # Jambe gauche
 76.         pygame.draw.line(ItiAnime.f, Color('black'),
(self.r.x+self.r.width,self.r.y+self.r.height),(self.milieux,self.r.y+self.r.height*3/4), 2) #
Jambe droite
 77.
 78. pygame.init() # Initialiser les modules de Pygame
 79. LARGEUR_FENETRE = 400
 80. HAUTEUR_FENETRE = 600
 81. fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre
 82. EntiteAnimee.set_fenetre(fenetre)
 83. pygame.display.set_caption("Exemple des Bots et Itis animés en diagonale avec super-classe
EntiteAnimee") # Définir le titre dans le haut de la fenêtre
 84.
 85. horloge = pygame.time.Clock() # Pour contrôler la fréquence des scènes
 86.
 87. # Placer deux BotAnime et deux ItiAnime dans la liste des entités
 88. liste_entite = []
 89. liste_entite.append(BotAnime(pygame.Rect((0,0),(20,40)),[5,10]))
 90. liste_entite.append(BotAnime(pygame.Rect((100,200),(30,60)),[0,2]))
 91. liste_entite.append(ItiAnime(pygame.Rect((200,150),(40,80)),[3,3]))
 92. liste_entite.append(ItiAnime(pygame.Rect((300,300),(50,100)),[5,10]))
 93.
 94. # Boucle d'animation
 95. fin = False
 96. while not fin :
 97.     event = pygame.event.poll() # Chercher le prochain évènement à traiter
 98.     if event.type == pygame.QUIT:  # Utilisateur a cliqué sur la fermeture de fenêtre ?
 99.         fin = True  # Fin de la boucle du jeu
100.     else :
101.         for une_entite in liste_entite :
102.             une_entite.deplacer()
103.
104.         fenetre.fill(Color('white')) # Dessiner le fond de la surface de dessin
105.         for une_entite in liste_entite :
106.             une_entite.dessiner()
107.
108.         pygame.display.flip() # Mettre à jour la fenêtre graphique
109.
110.         horloge.tick(60) # Pour animer avec 60 images pas seconde
111.
112. pygame.quit() # Terminer pygame
```

A noter que les appels aux méthodes `deplacer()` et `dessiner()` sont des exemples de la *surcharge dynamique*. La bonne version de la méthode à appeler est déterminée automatiquement à l'exécution du programme en fonction du type de l'objet.

Exercice. Ajoutez à l'exemple précédent une nouvelle sous-classe pour votre bonhomme préféré. Créez quelques objets de cette classe dans l'animation.

Exercice. Supposons que le `Iti` se déplace dans l'axe `z`. Coder une méthode `deplacer()` spécifique au `Iti` afin de donner cette impression. Ceci peut être effectué en modifiant la taille du `Iti` plutôt que sa position !

7.4 Création d'un module

Au-delà des fonctions et des classes, les modules offrent un moyen supplémentaire de gérer la complexité d'un logiciel en les découpant en plusieurs modules. Il est possible de regrouper des fonctions, classes, variables et du code exécutable dans un module. Par la suite, le module peut être importé et utilisé dans un script ou un autre module. Concrètement un module nommé `nomModule` correspond à un fichier appelé `nomModule.py` qui contient le code des éléments du module.

Dans l'exemple suivant, le code de l'exemple de la section précédente est réparti en deux fichiers. Les classes de la hiérarchie des entités animées, `EntiteAnimee`, `BotAnime` et `ItiAnime`, sont regroupées dans le module `Entite` qui correspond au fichier `Entite.py`. Le reste du code qui correspond au programme principal est placé dans le fichier `ExempleModuleImportEntite.py`.

CodePython/chapitre7/Entite.py

```
"""
Module qui contient la hiérarchie des classes EntiteAnimee
"""

import pygame
from pygame import Color

class EntiteAnimee :
    """ Un objet représente une entité qui est animée dans une fenêtre Pygame

    L'entité est inscrite dans le rectangle englobant défini par r. Il se déplace en
diagonale selon la vitesse v.
        r : pygame.Rect       Le rectangle englobant
        v : [int,int]         Vitesse de déplacement selon les deux axes x et y
    """

    @staticmethod
    def set_fenetre(fenetre):
        """ Fixer la variable de classe f qui représente la fenetre graphique

            fenetre : pygame.Surface
        """
        EntiteAnimee.f = fenetre

    def __init__(self,rectangle,vitesse):
        self.r = rectangle
        self.v = vitesse

    def deplacer(self):
        """ Déplacer selon self.v en diagonale en rebondissant sur les bords de la
fenetre"""
        if self.r.x+self.v[0] > EntiteAnimee.f.get_width()-self.r.width or
self.r.x+self.v[0] < 0 :
            self.v[0] = -self.v[0] # Inverser la direction en x
        self.r.x = self.r.x+self.v[0]
        if self.r.y+self.v[1] > EntiteAnimee.f.get_height()-self.r.height or
self.r.y+self.v[1] < 0 :
            self.v[1] = -self.v[1] # Inverser la direction en y
        self.r.y = self.r.y+self.v[1]

class BotAnime(EntiteAnimee) :
    """ Un objet représente un Bot qui est animé dans une fenêtre Pygame
        Sous-classe de EntiteAnimee
```

```
    """

    def dessiner(self):
        """ Dessiner un Bot.

        Le Bot est inscrit dans le rectangle englobant défini par la variable d'objet r
dans une fenetre de Pygame
        """

        pygame.draw.ellipse(BotAnime.f, Color('green'), ((self.r.x,self.r.y),(self.r.width,
self.r.height/2))) # Dessiner la tête
        pygame.draw.rect(BotAnime.f, Color('black'),
((self.r.x+self.r.width/4,self.r.y+self.r.height/8),(self.r.width/10,self.r.height/20))) #
L'oeil gauche
        pygame.draw.rect(BotAnime.f, Color('black'), ((self.r.x+self.r.width*3/4-
self.r.width/10,self.r.y+self.r.height/8),(self.r.width/10,self.r.height/20))) # L'oeil droit
        pygame.draw.line(BotAnime.f, Color('black'),
(self.r.x+self.r.width/4,self.r.y+self.r.height*3/8),(self.r.x+self.r.width*3/4,self.r.y+self.r
.height*3/8), 2) # La bouche
        pygame.draw.rect(BotAnime.f, Color('red'),
((self.r.x,self.r.y+self.r.height/2),(self.r.width,self.r.height/2))) # Le corps


class ItiAnime(EntiteAnimee) :
    """ Un objet représente un Bot qui est animé dans une fenêtre Pygame
        Sous-classe de EntiteAnimee
    """

    def dessiner(self):
        """ Dessiner un Iti.

        Le Iti est inscrit dans le rectangle englobant défini par la variable d'objet r
dans une fenetre de Pygame
        """
        self.milieux = self.r.x + self.r.width/2;
        self.milieuy = self.r.y + self.r.height/2;

        pygame.draw.ellipse(ItiAnime.f, Color('pink'),
((self.r.x+self.r.width/3,self.r.y),(self.r.width/3,self.r.height/4))) # Dessiner la tête
        pygame.draw.arc(ItiAnime.f, Color('black'),((self.milieux-
self.r.width/12,self.r.y+self.r.height/8),(self.r.width/6,self.r.height/14)),3.1416,0,2) # Le
sourire
        pygame.draw.ellipse(ItiAnime.f, Color('black'), ((self.milieux-
self.r.width/8,self.r.y+self.r.height/12),(self.r.width/12,self.r.height/24))) # L'oeil gauche
        pygame.draw.ellipse(ItiAnime.f, Color('black'), ((self.milieux+self.r.width/8-
self.r.width/12,self.r.y+self.r.height/12),(self.r.width/12,self.r.height/24))) # L'oeil droit
        pygame.draw.line(ItiAnime.f, Color('black'),
(self.milieux,self.r.y+self.r.height/4),(self.milieux,self.r.y+self.r.height*3/4), 2) # Le
corps
        pygame.draw.line(ItiAnime.f, Color('black'),
(self.r.x,self.r.y+self.r.height/4),(self.milieux,self.milieuy), 2) # Bras gauche
        pygame.draw.line(ItiAnime.f, Color('black'),
(self.r.x+self.r.width,self.r.y+self.r.height/4),(self.milieux,self.milieuy), 2) # Bras droit
        pygame.draw.line(ItiAnime.f, Color('black'),
(self.r.x,self.r.y+self.r.height),(self.milieux,self.r.y+self.r.height*3/4), 2) # Jambe gauche
        pygame.draw.line(ItiAnime.f, Color('black'),
(self.r.x+self.r.width,self.r.y+self.r.height),(self.milieux,self.r.y+self.r.height*3/4), 2) #
Jambe droite
```

CodePython/chapitre7/ExempleModuleImportEntite.py

```
"""
Exemple d'animation d'entités avec module Entite
"""
import Entite
import pygame
from pygame import Color

pygame.init() # Initialiser les modules de Pygame
LARGEUR_FENETRE = 400
HAUTEUR_FENETRE = 600
fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre
Entite.EntiteAnimee.set_fenetre(fenetre)
```

```
13. pygame.display.set_caption("Exemple des Bots et Itis animés en diagonale avec super-classe
EntiteAnimee") # Définir le titre dans le haut de la fenêtre
14.
15. horloge = pygame.time.Clock() # Pour contrôler la fréquence des scènes
16.
17. # Placer deux BotAnime et deux ItiAnime dans la liste des entités
18. liste_entite = []
19. liste_entite.append(Entite.BotAnime(pygame.Rect((0,0),(20,40)),[5,10]))
20. liste_entite.append(Entite.BotAnime(pygame.Rect((100,200),(30,60)),[0,2]))
21. liste_entite.append(Entite.ItiAnime(pygame.Rect((200,150),(40,80)),[3,3]))
22. liste_entite.append(Entite.ItiAnime(pygame.Rect((300,300),(50,100)),[5,10]))
23.
24. # Boucle d'animation
25. fin = False
26. while not fin :
27.     event = pygame.event.poll() # Chercher le prochain évènement à traiter
28.     if event.type == pygame.QUIT:  # Utilisateur a cliqué sur la fermeture de fenêtre ?
29.         fin = True  # Fin de la boucle du jeu
30.     else :
31.         for une_entite in liste_entite :
32.             une_entite.deplacer()
33.
34.         fenetre.fill(Color('white')) # Dessiner le fond de la surface de dessin
35.         for une_entite in liste_entite :
36.             une_entite.dessiner()
37.         pygame.display.flip() # Mettre à jour la fenêtre graphique
38.
39.         horloge.tick(60) # Pour animer avec 60 images pas seconde
40.
41. pygame.quit() # Terminer pygame
```

Pour utiliser le module, il faut d'abord l'importer :

```
import Entite
```

Quand Python voit un `import`, il recherche un fichier nommé `Entite.py` qui contient le code du module et qui se trouve dans un des répertoires de la variable `sys.path`. Cette variable est initialisée avec les chemins de répertoire suivant :

1. le dossier du script courant
2. la variable d'environnement PYTHONPATH
3. une valeur par défaut dépendante de l'installation.

Pour faire référence à un élément du module, on emploie la notation préfixée `NomModule.NomElement`. Dans le code suivant, `BotAnime` provient du module `Entite`.

```
liste_entite.append(Entite.BotAnime(pygame.Rect((0,0),(20,40)),[5,10]))
```

Tel qu'expliqué précédemment, un module possède son propre espace de nom. Ceci permet d'éviter les conflits de noms entre modules différents. La notation avec préfixe permet d'éliminer toute ambiguïté au sujet de la provenance de l'élément désigné.

Il y a d'autres possibilités pour importer un module. Il est possible d'assigner un nom différent au module avec la clause *as*. La ligne suivante renomme le module `Entite` avec l'alias `En` :

```
import Entite as En
```

Par la suite, le nouveau nom `En` sert à désigner le module `Entite` comme dans :

```
liste_entite.append(En.BotAnime(pygame.Rect((0,0),(20,40)),[5,10]))
```

Pour plus de détails au sujet des modules, voir : https://docs.python.org/fr/3.6/tutorial/modules.html.

Exercice. Reprenez l'exercice de la section précédente pour votre propre créature animée en employant un module pour les entités à la manière de l'exemple précédent.

7.5 Objet itérable, objet itérateur, fonction génératrice et expression génératrice

Un aspect élégant de Python est la manière dont le `for` exploite le patron d'itérateur pour parcourir des collections d'éléments. Le patron d'itérateur suppose qu'il est possible d'itérer sur un ensemble d'éléments, un à la fois, en appelant une méthode qui retourne l'élément suivant.

Les séquences Python telles que `list`, `tuple` et `str` sont des *itérables*. Un objet itérable doit implémenter la méthode spéciale `__iter__()` qui retourne un objet itérateur permettant d'itérer sur les éléments de l'itérable. Pour obtenir un objet itérateur, il faut appeler la fonction `iter()` qui appelle la méthode spéciale `__iter__()`. Un *objet itérateur* doit implémenter une méthode spéciale `__next__()`. La fonction `next()` retourne le prochain objet à traiter en appelant la méthode `__next__()` sur l'objet itérateur. S'il n'y en a plus, la méthode retourne l'exception `StopIteration`.

L'exemple de code suivant exploite les fonctions `iter()` et `next()` pour parcourir les éléments d'une liste.

```
1. une_liste = [3,1,2]
2. un_iterateur = iter(une_liste)
3. print(un_iterateur)
4.
5. print(next(un_iterateur))
6. print(next(un_iterateur))
7. print(next(un_iterateur))
```

```
8. print(next(un_iterateur))
```

Résultat :

```
<list_iterator object at 0x00000206826D69A0>
3
1
2
Traceback (most recent call last):

  File
"C:\Users\vango\OneDrive\Documents\GitHub\CodePython\ExempleIterNext.py",
line 10, in <module>
    print(next(un_iterateur))

StopIteration
```

Il est possible d'appeler les méthodes spéciales directement mais ceci n'est pas recommandé :

```
1. une_liste = [3,1,2]
2. un_iterateur = une_liste.__iter__()
3. print(un_iterateur)
4.
5. print(un_iterateur.__next__())
6. print(un_iterateur.__next__())
7. print(un_iterateur.__next__())
8. print(un_iterateur.__next__())
```

Résultat :

```
<list_iterator object at 0x0000020682AD9130>
3
1
2
Traceback (most recent call last):

  File
"C:\Users\vango\OneDrive\Documents\GitHub\CodePython\ExempleMethodesSpeciale
sIterNext.py", line 10, in <module>
    print(un_iterateur.__next__())

StopIteration
```

L'exemple suivant utilise une boucle qui appelle explicitement les fonctions `iter()` et `next()` pour parcourir la liste :

```
 1. une_liste = [3,1,2]
 2. un_iterateur = iter(une_liste)
 3. while True:
 4.     try:
 5.         suivant = next(un_iterateur)
 6.         print(suivant)
 7.     except StopIteration:
```

```
 8.          print("Fin de l'itérable")
 9.          break
```

Résultat :

```
3
1
2
Fin de l'itérable
```

Le même effet est produit par le `for` d'une manière beaucoup plus succincte :

```
1. for suivant in une_liste :
2.     print(suivant)
```

Le `for` exploite les fonctions `iter()` et `next()` pour itérer sur les collections. Il est possible de définir une classe itérable en implémentant les méthodes `__iter__()` et `__next__()`.

Dans l'exemple suivant, la classe `SequenceSup0` implémente une classe d'objets itérables. Un objet de cette classe permet d'itérer sur les éléments d'une séquence qui sont supérieurs à 0.

CodePython/chapitre7/ExempleIterator.py

```
 1. class SequenceSup0:
 2.     """Représente la séquence des éléments supérieurs à 0"""
 3.     def __init__(self, seq):
 4.         self.seq = seq
 5.         self.taille = len(seq)
 6.         self.indice = 0
 7.
 8.     def __iter__(self):
 9.         return self
10.
11.     def __next__(self):
12.         while (self.indice < self.taille):
13.             if (self.seq[self.indice] > 0) :
14.                 self.indice = self.indice + 1
15.                 return self.seq[self.indice-1]
16.             self.indice = self.indice + 1
17.         raise StopIteration
18.
19. for element in SequenceSup0([3,-1,4,6,0,8]):
20.       print(element)
21.
```

Résultat :

```
3
4
```

```
6
8
```

Exercice. Codez une classe itérable qui retourne les éléments pairs d'une liste d'entiers.

Fonction génératrice (*generator*)

Une fonction génératrice (generator) permet de produire un itérateur d'une manière plus succincte que la définition d'une classe à cet effet. L'énoncé *yield* est employé dans la fonction pour retourner le prochain élément à l'opposé d'une fonction normale qui emploie le *return*. Pour produire un effet équivalent à un itérateur, la fonction suspend l'exécution à l'appel du *yield* et préserve sont état entre les appels afin de poursuivre le traitement à partir du *yield* précédent contrairement à une fonction normale qui part de zéro à chacun des appels. Lorsque la fonction génératrice est appelée, elle retourne implicitement un *itérateur de générateur* produit par Python à partir de la fonction génératrice.

La fonction `filtreSup0()` produit le même effet que l'itérateur de l'exemple précédent avec un code beaucoup plus compact.

```
def filtreSup0(seq) :
    for element in seq :
        if element > 0 :
            yield element

for element in filtreSup0([3,-1,4,6,0,8]):
        print(element)
```

La notion de liste en compréhension a été présentée précédemment. Elle permet de produire une séquence à partir d'une forme simplifiée du `for`.

Expression génératrice

Une *expression génératrice* (*generator expression*) est analogue à une liste en compréhension mais elle produit un itérateur, ce qui élimine la copie en mémoire. Ceci peut conduire à une économie de mémoire. D'autre part, les éléments ne sont générés qu'au besoin. Syntaxiquement, l'utilisation de parenthèses plutôt que de crochets désigne une expression génératrice.

L'exemple suivant illustre la différence entre une liste en compréhension et une expression génératrice. Le premier cas montre une liste en compréhension qui produit le même effet que l'exemple précédent d'extraction des éléments supérieurs à 0.

```
>>> liste_sup0 = [element for element in [3,-1,4,6,0,8] if element > 0]
>>> liste_sup0
[3, 4, 6, 8]
```

Dans le deuxième cas, l'expression génératrice produit un générateur qui ne matérialise pas le résultat. Pour matérialiser le résultat, une liste est construite à partir du générateur.

```
>>> liste_sup0 = (element for element in [3,-1,4,6,0,8] if element > 0)
>>> liste_sup0
<generator object <genexpr> at 0x00000191336C6888>
>>> list(liste_sup0)
[3, 4, 6, 8]
```

7.6 Création d'un itérateur avec la fonction zip()

Supposons qu'il faille produire une liste de couples formés des éléments de deux listes, en combinant les éléments qui ont le même indice. La fonction `zip()` produit un objet itérateur sur les couples ainsi générés.

```
>>> iterateur_couples = zip([1,2,3],[5,3,4])
>>> for i,j in iterateur_couples:
...     print(i,j)
...
1 5
2 3
3 4
```

Exercice. Produire une liste formée de la somme des éléments deux à deux des deux listes [1,2,3] et [5,3,4].

Solution.

```
>>> [i+j for i,j in zip([1,2,3],[5,3,4])]
[6, 5, 7]
```

À l'inverse, pour produire deux listes à partir des éléments d'une liste de couples, il est possible de faire appel à l'opérateur * de décompaction d'argument en combinaison avec la fonction `zip()`. Dans l'exemple suivant, en passant `*liste_triplets` à la fonction `zip()`, la liste de couples est décompactée en trois arguments, chacun des arguments étant un couple. L'itérateur produit par la fonction `zip()` construit un premier triplet formé des premiers éléments des trois couples, et un second triplet formé des seconds éléments des trois couples.

```
>>> liste_couples = [(1,5),(2,3),(3,4)]
>>> iterateur_triplets = zip(*liste_couples)
>>> for triplet in   iterateur_triplets :
...  print(list(triplet))
...
[1, 2, 3]
```

```
[5, 3, 4]
```

Exercice. Former deux listes, la liste des noms et la liste des numéros de téléphones, à partir de la liste des couples suivante :

```
[('Pierre','514-333-3333'),  ('Jean','514-222-2222'),  ('Jacques','514-555-
5555'), ('Paul','514-777-7777')]
```

7.7 Programmation fonctionnelle et fonction lambda (anonyme)

Python inclut des fonctions qui permettent de produire un style de programmation dit fonctionnel. Le traitement est produit par la *composition de fonctions* plutôt que par un enchaînement procédural. Le mécanisme d'itérateur Python est exploité implicitement à cet effet.

Dans sa version de base, la fonction `map(fonction, iterable)` applique la fonction à chacun des éléments de `iterable` et retourne un objet itérable de la classe `map` avec les résultats. A noter que Python permet d'employer une fonction comme paramètre ! Dans l'exemple suivant, la fonction `map()` applique la fonction `int()` à chacune des chaînes de caractères de la liste et retourne un objet itérable de la classe `map` qui permet d'itérer sur les entiers résultants. Dans le code suivant, l'itérable est ensuite converti en liste pour afficher le résultat.

```
>>> map(int, ["3","10","5","-2"])
<map object at 0x000001CCACA7BF60>
>>> list(map(int, ["3","10","5","-2"]))
[3, 10, 5, -2]
```

A noter qu'il est possible de produire le même effet avec une liste en compréhension :

```
>>> [int(x) for x in ["3","10","5","-2"]]
[3, 10, 5, -2]
```

Ou encore une expression génératrice :

```
>>> list((int(x) for x in ["3","10","5","-2"]))
[3, 10, 5, -2]
```

L'emploi d'une liste en compréhension ou d'une expression génératrice est recommandé pour rendre le code plus lisible.

L'exemple suivant calcule les carrés d'une liste d'entiers. À cet effet, la fonction `carres()` est définie et employée comme argument du `map()`.

```
>>> def carres(x):
...     return x**2
```

```
...
>>> list(map(carres, [5,2,10]))
[25, 4, 100]
```

Exercice. Calculez la liste des carrés avec une liste en compréhension. Encore ici, une liste en compréhension produit le même résultat d'une manière plus lisible :

```
>>> def carres(x):
...     return x**2
...
>>> [carres(x) for x in [5,2,10]]
[25, 4, 100]
```

Un avantage d'une liste en compréhension ou d'une expression génératrice est la possibilité d'appliquer une expression à la liste plutôt qu'une simple fonction :

```
>>> [x**2 for x in [5,2,10]]
[25, 4, 100]
>>> list(x**2 for x in [5,2,10])
[25, 4, 100]
```

Dans le cas d'un `map()`, l'expression doit être définie par une fonction. Néanmoins, il est possible d'employer une expression lambda pour définir la fonction directement dans le `map()`.

Fonction lambda (anonyme)

Une fonction lambda est une fonction anonyme.

Une fonction lambda est utile lorsqu'il faut définir une fonction mais qu'il n'est pas nécessaire de lui donner un nom. Par exemple, elle peut être définie directement comme argument d'une fonction qui prend une autre fonction en paramètre. Ici, l'exemple des carrés est repris avec une fonction lambda passée en argument au `map()`.

```
>>> list(map (lambda x:x**2, [5,2,10]))
[25, 4, 100]
```

La programmation fonctionnelle permet de produire un traitement complexe en composant les fonctions. Il est par exemple possible d'appliquer la fonction `map()` au résultat produit par une autre fonction `map()`. Dans l'exemple suivant qui combine les deux précédents, une première fonction `map()` produit une liste d'entiers à partir d'une liste de chaînes de caractères. Le résultat est ensuite passé à une autre fonction `map()` qui en calcule les carrés.

```
>>> list(map(lambda x: x**2, map(int, ["3","10","5","-2"])))
```

```
[9, 100, 25, 4]
```

La fonction `reduce()` du module `functools` permet de prendre une liste d'éléments et de produire un élément qui cumule l'application d'une fonction binaire à chacun des éléments de la liste. L'exemple suivant produit la somme des éléments de la liste.

```
>>> from functools import reduce
>>> reduce(lambda x,y: x+y,[10,5,20,2])
37
```

Le `reduce()` calcule l'équivalent de l'expression suivante : `(((10+5)+20)+2)`

Ainsi, le `reduce()` peut être combiné à l'exemple des carrés des entiers pour calculer la somme des carrés :

```
>>> reduce(lambda x,y: x+y, map (lambda x:x**2, [5,2,10]))
129
```

La fonction `filter(fonction, iterable)` permet de sélectionner les éléments d'un itérable qui respectent une condition exprimée par une fonction booléenne. Le résultat est un itérable. L'exemple suivant retourne la liste des éléments supérieur à 0 :

```
>>> list(filter(lambda x : x>0, [4,-2,5,0]))
[4, 5]
```

A noter que le même effet peut être produit avec un générateur ou une liste en compréhension incluant une clause `if`:

```
 >>> (list(element for element in [4,-2,5,0] if element > 0))
[4, 5]
```

La combinaison des fonctions `map`, `filter` et `reduce`, permet de produire un grand nombre de traitements fréquemment rencontrés en analyse de données[16]. Voici un exemple qui combine les trois fonctions en produisant la somme des carrés des entiers supérieurs à 0.

```
>>> reduce(lambda x,y: x+y,map(lambda x: x**2,filter(lambda x : x>0, [4,-
2,5,0])))
41
```

Encore ici, il est possible d'exprimer ce calcul par un fonction de groupe combinée à une expression génératrice :

[16] L'approche de programmation MapReduce est employée systématiquement dans les environnements de programmation à haut parallélisme à cause de la possibilité de réaliser les calculs en parallèle de manière naturelle.

```
>>> sum(x**2 for x in [4,-2,5,0] if x > 0)
41
```

Exercice. Convertir une liste de chaînes de caractères en majuscules.

Exercice. Convertir une liste de températures de Celsius à fahrenheit et retourner la moyenne des températures qui sont supérieures à 0.

Il est possible d'appliquer une fonction à plusieurs arguments avec le `map()`. Il faut inclure autant d'itérables que d'arguments à cet effet. L'exemple suivant montre une fonction à deux paramètres qui retourne un itérable formé des produits élément par élément de deux itérables.

```
>>> list(map (lambda x,y: x*y, [5,2,10],[4,10,5]))
[20, 20, 50]
```

Exercice. Calculer le produit scalaire de deux vecteurs représentés par deux listes.

Solution.

```
>>> from functools import reduce
>>> reduce(lambda x,y: x+y, map (lambda x,y: x*y, [5,2,10],[4,10,5]))
90
>>> sum(x * y for x, y in zip([5, 2, 10], [4, 10, 5]))
90
```

7.8 Namedtuple : type tuple avec noms d'attributs

Le type `collections.namedtuple` permet d'employer des noms pour accéder aux attributs d'un n-uplet en plus des indices. La fonction `namedtuple()` est une usine à classe qui crée une nouvelle classe, sous-classe de *tuple,* en spécifiant le nom de la classe et les noms des attributs. L'exemple suivant illustre la création d'un `namedtuple`.

```
>>> Contact = collections.namedtuple('Contact', ['nom', 'prenom',
'telephone'])
>>> c1=Contact('Einstein', 'Albert', '123-123-1234')
>>> c1
Contact(nom='Einstein', prenom='Albert', telephone='123-123-1234')
>>> c1.nom
'Einstein'
>>> c1[0]
'Einstein'
```

La version 3.8 de Python introduit la possibilité d'annotations de types avec la classe `typing.NamedTuple`.

8 Animation 2D et développement d'un jeu simple

Ce chapitre présente le développement d'un jeu simple qui combine l'animation 2D et une interactivité de base au moyen de la souris. Les acteurs sont raffinés par rapport au chapitre précédent en ajoutant des sons et des mouvements. Le jeu démarre en animant une série d'entités. Le but de l'utilisateur est de détruire ces entités en cliquant dessus avec la souris. Lorsque l'entité est touchée, elle disparaît en poussant un cri de désarroi.

La classe `EntiteAnimeeAvecSon` ajoute les aspects suivants à la classe `EntiteAnimee` développée au chapitre 7 :

- `son` : cette variable d'objet est un clip sonore qui est employé pour que l'entité émette un son lorsqu'elle est éliminée

- méthode `touche()` : cette méthode retourne vrai si la coordonnée (x, y) est à l'intérieur du rectangle englobant de l'entité. Elle sert à vérifier si l'utilisateur a touché l'entité avec la souris dans le contexte du jeu.

Les classes des entités du jeu sont placées dans le module `EntiteDuJeu`.

CodePython/chapitre8/EntiteDuJeu.py

```
"""
Module qui contient la hiérarchie des classes EntiteAnimee
"""
# Importer la librairie de pygame et initialiser
import pygame
from pygame import Color

class EntiteAnimeeAvecSon :
    """ Un objet représente une entité qui est animée dans une fenêtre Pygame.

    L'entité est inscrite dans le rectangle englobant r. Il se déplace en diagonale selon la vitesse v.
    Elle émet un son lorsque supprimée.
        r : pygame.Rect       Le rectangle englobant
        v : [int,int]         Vitesse de déplacement selon les deux axes x et y
        son : pygame.mixer.Sound
    """

    @staticmethod
    def set_fenetre(fenetre):
        """ Fixer la variable de classe f qui représente la fenetre graphique

            fenetre : pygame.Surface
        """
        EntiteAnimeeAvecSon.f = fenetre

    def __init__(self,rectangle,vitesse,fichier_son):
        self.r = rectangle
        self.v = vitesse
```

```
29.         self.son = pygame.mixer.Sound(fichier_son)
30.
31.     def prochaine_scene(self):
32.         """ Déplacer selon self.v en diagonale en rebondissant sur les bords de la
fenetre"""
33.         if self.r.x+self.v[0] > EntiteAnimeeAvecSon.f.get_width()-self.r.width or
self.r.x+self.v[0] < 0 :
34.             self.v[0] = -self.v[0] # Inverser la direction en x
35.         self.r.x = self.r.x+self.v[0]
36.         if self.r.y+self.v[1] > EntiteAnimeeAvecSon.f.get_height()-self.r.height or
self.r.y+self.v[1] < 0 :
37.             self.v[1] = -self.v[1] # Inverser la direction en y
38.         self.r.y = self.r.y+self.v[1]
39.
40.     def touche(self,x,y):
41.         return ((x >= self.r.x) and (x <= self.r.x + self.r.width) and (y >= self.r.y) and
(y <= self.r.y + self.r.height))
42.
43.     def emettre_son(self):
44.         self.son.play()
45.
46. class EntiteAvecEtat(EntiteAnimeeAvecSon):
47.     def __init__(self,rectangle,vitesse,fichier_son,nombre_etats):
48.         super().__init__(rectangle,vitesse,fichier_son)
49.         self.nombre_etats = nombre_etats
50.         self.etat_courant = 0
51.
52.     def prochaine_scene(self):
53.         self.etat_courant=(self.etat_courant+1)%self.nombre_etats
54.         super().prochaine_scene()
55.
56.
57. class BotAnime(EntiteAnimeeAvecSon) :
58.     """ Un objet représente un Bot qui est animé dans une fenêtre Pygame
59.         Sous-classe de EntiteAnimee
60.     """
61.
62.     def dessiner(self):
63.         """ Dessiner un Bot.
64.
65.         Le Bot est inscrit dans le rectangle englobant défini par la variable d'objet r
dans une fenetre de Pygame
66.         """
67.
68.         pygame.draw.ellipse(BotAnime.f, Color('green'),
((self.r.x,self.r.y),(self.r.width, self.r.height/2))) # Dessiner la tête
69.         pygame.draw.rect(BotAnime.f, Color('black'),
((self.r.x+self.r.width/4,self.r.y+self.r.height/8),(self.r.width/10,self.r.height/20))) #
L'oeil gauche
70.         pygame.draw.rect(BotAnime.f, Color('black'), ((self.r.x+self.r.width*3/4-
self.r.width/10,self.r.y+self.r.height/8),(self.r.width/10,self.r.height/20))) # L'oeil droit
71.         pygame.draw.line(BotAnime.f, Color('black'),
(self.r.x+self.r.width/4,self.r.y+self.r.height*3/8),(self.r.x+self.r.width*3/4,self.r.y+self.r
.height*3/8), 2) # La bouche
72.         pygame.draw.rect(BotAnime.f, Color('red'),
((self.r.x,self.r.y+self.r.height/2),(self.r.width,self.r.height/2))) # Le corps
73.
74.
75. class ItiAnimeVolant(EntiteAvecEtat) :
76.     """ Un objet représente un Iti qui est animé dans une fenêtre Pygame
77.         Sous-classe de EntiteAnimee
78.     """
79.
80.     def dessiner(self):
81.         """ Dessiner un Iti.
82.
83.         Le Iti est inscrit dans le rectangle englobant défini par la variable d'objet r
dans une fenetre de Pygame
84.         L'etat courant détermine la hauteur des bras.
85.         """
86.         self.milieux = self.r.x + self.r.width/2;
87.         self.milieuy = self.r.y + self.r.height/2;
88.
89.         pygame.draw.ellipse(ItiAnimeVolant.f, Color('pink'),
((self.r.x+self.r.width/3,self.r.y),(self.r.width/3,self.r.height/4))) # Dessiner la tête
```

```
 90.         pygame.draw.arc(ItiAnimeVolant.f,Color('black'),((self.milieux-
self.r.width/12,self.r.y+self.r.height/8),(self.r.width/6,self.r.height/14)),3.1416,0,2) # Le
sourire
 91.         pygame.draw.ellipse(ItiAnimeVolant.f, Color('black'), ((self.milieux-
self.r.width/8,self.r.y+self.r.height/12),(self.r.width/12,self.r.height/24))) # L'oeil gauche
 92.         pygame.draw.ellipse(ItiAnimeVolant.f, Color('black'),
((self.milieux+self.r.width/8-
self.r.width/12,self.r.y+self.r.height/12),(self.r.width/12,self.r.height/24))) # L'oeil droit
 93.         pygame.draw.line(ItiAnimeVolant.f, Color('black'),
(self.milieux,self.r.y+self.r.height/4),(self.milieux,self.r.y+self.r.height*3/4), 2) # Le
corps
 94.         pygame.draw.line(ItiAnimeVolant.f, Color('black'),
(self.r.x,self.r.y+self.r.height/4+(self.r.height/4)*self.etat_courant),(self.milieux,self.mili
euy), 2) # Bras gauche
 95.         pygame.draw.line(ItiAnimeVolant.f, Color('black'),
(self.r.x+self.r.width,self.r.y+self.r.height/4+(self.r.height/4)*self.etat_courant),(self.mili
eux,self.milieuy), 2) # Bras droit
 96.         pygame.draw.line(ItiAnimeVolant.f, Color('black'),
(self.r.x,self.r.y+self.r.height),(self.milieux,self.r.y+self.r.height*3/4), 2) # Jambe gauche
 97.         pygame.draw.line(ItiAnimeVolant.f, Color('black'),
(self.r.x+self.r.width,self.r.y+self.r.height),(self.milieux,self.r.y+self.r.height*3/4), 2) #
Jambe droite
 98.
 99.
100. class EntiteAnimeeParImages(EntiteAvecEtat):
101.         def __init__(self,rectangle,vitesse,fichier_son,nombre_etats,nom_dossier):
102.             super().__init__(rectangle,vitesse,fichier_son,nombre_etats)
103.             self.image_animation = []
104.             for i in range(nombre_etats):
105.
self.image_animation.append(pygame.transform.scale(pygame.image.load(nom_dossier+"/"+nom_dossie
r+str(i+1)+".gif"),(self.r.width,self.r.height)))
106.         def dessiner(self):
107.
EntiteAnimeeParImages.f.blit(self.image_animation[self.etat_courant],[self.r.x,self.r.y])
108.
```

La figure suivante montre la hiérarchie des classes du module `EntiteDuJeu`.

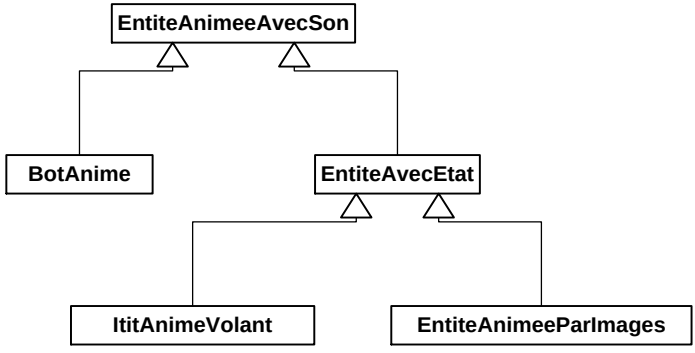


Figure 13. Hiérarchie des entités animées.

La classe `EntiteAnimeeAvecSon` ajoute à la classe `EntiteAnimee` du chapitre précédent une variable d'objet `son` de la classe `pygame.mixer.Sound` L'objet `son` est créé à partir d'un fichier dont le nom `fichier_son` est passé en paramètre à `__init__()` et employé pour le constructeur d'objet de type `pygame.mixer.Sound` :

```
        self.son = pygame.mixer.Sound(fichier_son)
```

Lorsqu'une entité est touchée par le clic de la souris, elle émet le son avec la méthode `emettre_son()` qui ne fait qu'appeler la méthode `play()` sur l'objet `son` :

```
    def emettre_son(self):
        self.son.play()
```

Pour rendre l'animation plus intéressante, certaines entités peuvent non seulement se déplacer mais aussi effectuer des mouvements. Pour simplifier, supposons que l'entité effectue toujours la même série de mouvements en répétant la même séquence d'images. Généralement, on distingue deux techniques de production des images simulant les mouvements :

- *Animation vectorielle*. Chacune des images est produite en invoquant des opérations de dessin.

- *Animation bitmap*. Chacune des images est préalablement créée. Cette approche évite le calcul répété des images à dessiner mais nécessite le stockage des images. D'autre part, dans le cas d'images très complexes, il peut être très difficile de produire le résultat recherché avec des opérations de dessin.

Pour produire la séquence d'animation des mouvements, il faut un moyen de déterminer la forme graphique appropriée pour la prochaine scène. La classe `EntiteAvecEtat` sous-classe de `EntiteAnimeeAvecSon` ajoute deux variables d'objets à cet effet. La variable d'objet `nombre_etats` représente le nombre de formes graphiques de l'entité et `etat_courant` détermine la prochaine forme graphique de l'entité.

Redéfinition d'une méthode par spécialisation et surcharge dynamique

Lorsqu'une méthode d'une sous-classe est définie, il est possible de redéfinir une méthode de la super-classe qui porte le même nom afin de spécialiser le comportement de la méthode pour la sous-classe. Dans notre exemple, la méthode `__init__()` de la sous-classe `EntiteAvecEtat` redéfinit la méthode `__init__()` du même nom que la méthode de la super-classe `EntiteAnimeeAvecSon`. La méthode de la sous-classe ajoute au comportement de la méthode de la super-classe l'initialisation des variables qui sont concernées par le mouvement de l'entité. Il peut ainsi y avoir plusieurs méthodes du même nom. Python détermine

dynamiquement quelle méthode `__init__()` appeler en fonction de la classe de l'objet visé. Ce principe est appelé *surcharge dynamique* (ou encore *polymorphisme dynamique*) *d'un nom de méthode*. Si la méthode est appelée pour un objet de la sous-classe, c'est la méthode de la sous-classe qui est employée. Dans la méthode `__init__()` de la sous-classe, il est aussi possible d'appeler la méthode `__init__()` de la super-classe, comme c'est le cas de notre exemple. Afin d'éviter toute ambigüité, dans la sous-classe, on emploie la syntaxe `super().__init__()` afin de préciser que l'appel se fait sur la méthode de la super-classe. Ainsi dans notre exemple, le `__init__()` de la sous-classe redéfinit le `__init__()` de la super-classe en lui ajoutant un comportement qui consiste à initialiser les variables d'objets `nombre_etats` et `etat_courant`.

```
    def __init__(self,rectangle,vitesse,fichier_son,nombre_etats):
        super().__init__(rectangle,vitesse,fichier_son)
        self.nombre_etats = nombre_etats
        self.etat_courant = 0
```

La méthode `prochaine_scene()` est aussi un exemple de redéfinition de la méthode correspondante de la super-classe. Elle ajoute le comportement de passage à l'état suivant du mouvement de l'entité en ajoutant 1 à `etat_courant`, modulo le `nombre_etats` afin de produire un effet de répétition du mouvement.

```
    def prochaine_scene(self):
        self.etat_courant=(self.etat_courant+1)%self.nombre_etats
        super().prochaine_scene()
```

- **Animation vectorielle de mouvements**

La sous-classe `ItiAnimeVolant` de la classe `EntiteAvecEtat` réalise l'animation des mouvements d'un `Iti` qui vole par une animation vectorielle. La méthode `dessiner()` de la sous-classe `ItiAnimeVolant` utilise la variable `etat_courant` dans le dessin du `Iti` pour faire bouger ses bras et donner l'impression d'un battement d'ailes. La position des bras dépend de la valeur de `etat_courant`.

```
        # Bras gauche
        pygame.draw.line(ItiAnimeVolant.f,                                                    NOIR,
(self.r.x,self.r.y+self.r.height/4+(self.r.height/4)*self.etat_courant),(self.milieux,self.milieuy), 2)
        # Bras droit

pygame.draw.line(ItiAnimeVolant.f,NOIR,(self.r.x+self.r.width,self.r.y+self.r.height/4+(self.r.height/4
)*self.etat_courant),
             (self.milieux,self.milieuy), 2)
```

- **Animation bitmap de mouvements**

La sous-classe `EntiteAnimeeParImages` illustre une autre manière de produire l'animation :

```
class EntiteAnimeeParImages(EntiteAvecEtat):
        def __init__(self,rectangle,vitesse,fichier_son,nombre_etats,nom_dossier):
            super().__init__(rectangle,vitesse,fichier_son,nombre_etats)
            self.image_animation = []
            for i in range(nombre_etats):
                self.image_animation.append(

pygame.transform.scale(pygame.image.load(nom_dossier+"/"+nom_dossier+str(i+1)+".gif"),
                    (self.r.width,self.r.height)))
        def dessiner(self):
            EntiteAnimeeParImages.f.blit(self.image_animation[self.etat_courant],[self.r.x,self.r.y])
```

Plutôt que d'employer les méthodes de dessin de `Pygame`, une séquence d'images pré-enregistrées est utilisée. `Pygame` inclut des classes pour la manipulation des images selon différentes normes telles GIF (*Graphics Interchange Format*), JPEG (*Joint Photographic Experts Group*) et PNG (*Portable Network Graphics*). Dans notre cas, ce sont des images de type GIF.

La figure suivante montre le contenu de 9 fichiers GIF qui représentent une séquence d'images utilisées pour produire une animation d'un `coq` qui bat des ailes pour voler approximativement. Le nom du fichier est montré sous l'image. L'extension `.gif` est une convention pour les fichiers d'images encodés selon la norme GIF.

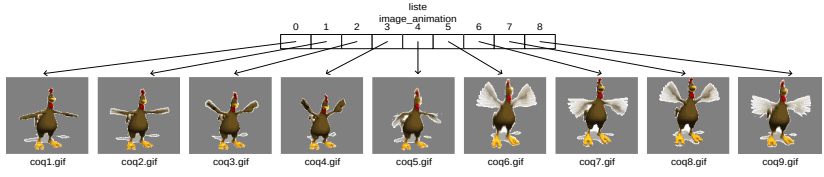


Figure 14. Série d'images pour l'animation du coq qui bât des ailes pour voler approximativement.

Dans la méthode `__init__()` de la classe `EntiteAnimeeParImages`, les images sont stockées dans la variable d'objet `image_animation` de type `list` par la boucle `for`. Une image est représentée par un objet de la classe `pygame.Surface`. L'image est chargée en mémoire à partir d'un fichier par la méthode `pygame.image.load()`.

L'argument

```
nom_dossier+"/"+nom_dossier+str(i+1)+".gif"
```

est le chemin du fichier. L'argument `nom_dossier` passé à `__init__()` est par convention le nom d'un dossier qui contient les fichiers d'images qui sont nommés selon la convention `nom_dossieri` où `i` est un entier de 1 à *n*, *n* étant le nombre d'images de la séquence d'animation. L'entier désigne l'ordre de l'image dans la séquence d'animation. Dans notre exemple de coq, les fichiers sont nommés `coqi.gif` et ils sont dans un dossier nommé `coq` du répertoire du programme Python.

La méthode `pygame.transform.scale()` est appelé sur l'objet `pygame.Surface` pour le mettre à l'échelle désigné par les paramètres `hauteur` et `largeur` de l'entité. La méthode retourne un nouvel objet qui est ajouté à la liste `image_animation`. La méthode `dessiner()` affiche les images de la liste `image_animation` en se servant de `etat_courant` comme indice. Elle fait appel à la méthode `pygame.surface.blit()` qui copie l'image de l'animation sur la `fenetre` à la position courante `(x,y)` de l'entité.

```
        def dessiner(self):

EntiteAnimeeParImages.f.blit(self.image_animation[self.etat_courant],[self.r
.x,self.r.y])
```

Modes de transparence

Lorsqu'une image est affichée, chacun des pixels peut être affiché selon trois modes de transparence :

Opaque. Le pixel apparaît sans altération en remplaçant la couleur existante.

Transparent. Le pixel n'est pas affiché. On voit donc ce qui est déjà affiché. Dans notre exemple de coq, le gris foncé qui entoure le coq correspond à des pixels transparents.

Translucide. Le pixel est affiché en laissant transparaître partiellement ce qui est déjà affiché. Ceci est réalisé en combinant la couleur existante avec la couleur du pixel à afficher.

Le mode de transparence du pixel est spécifié dans l'encodage de l'image. Le format GIF permet à chacun des pixels d'être transparent ou opaque. Le format PNG permet les trois modes alors que le format JPEG ne permet que le mode opaque. Les éditeurs d'images permettent de préciser le mode de transparence de chacun des pixels.

Le programme principal pour notre exemple de jeu est semblable à l'exemple d'animation du chapitre précédent. Il ajoute une interaction avec la souris semblable à celle vue au chapitre 5.

CodePython/chapitre8/ExempleJeuSimple.py

```
"""
Exemple de jeu : programme principal
"""
import EntiteDuJeu
import pygame
from pygame import Color

pygame.init() # Initialiser les modules de Pygame
LARGEUR_FENETRE = 400
HAUTEUR_FENETRE = 600
fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre
EntiteDuJeu.EntiteAnimeeAvecSon.set_fenetre(fenetre)
pygame.display.set_caption("Exemple de jeu avec module EntiteDuJeu")

horloge = pygame.time.Clock() # Pour contrôler la fréquence des scènes

# Création de la liste des entités du jeu
liste_entite = []
liste_entite.append(EntiteDuJeu.BotAnime(pygame.Rect((10,100),(40,80)),[3,3],"Son2.wav"))
liste_entite.append(EntiteDuJeu.BotAnime(pygame.Rect((200,200),(50,100)),[0,2],"Son2.wav"))

liste_entite.append(EntiteDuJeu.ItiAnimeVolant(pygame.Rect((200,50),(50,100)),[3,0],"Son3.wav",
3))

liste_entite.append(EntiteDuJeu.EntiteAnimeeParImages(pygame.Rect((50,100),(100,100)),[5,5],"So
n4.wav",9,"coq"))

# Boucle d'animation
fin = False
while not fin :
    event = pygame.event.poll() # Chercher le prochain évènement à traiter
    if event.type == pygame.QUIT:  # Utilisateur a cliqué sur la fermeture de fenêtre ?
        fin = True  # Fin de la boucle du jeu
    else :
        if event.type == pygame.MOUSEBUTTONUP : # Utilisateur a cliqué dans la fenêtre ?
            x=event.pos[0] # Position x de la souris
            y=event.pos[1] # Position y de la souris
            for une_entite in liste_entite :
                if une_entite.touche(x,y):
                    une_entite.emettre_son()
                    liste_entite.remove(une_entite)

        for une_entite in liste_entite :
            une_entite.prochaine_scene()

        fenetre.fill(Color('white')) # Dessiner le fond de la surface de dessin
        for une_entite in liste_entite :
            une_entite.dessiner()

        pygame.display.flip() # Mettre à jour la fenêtre graphique
        horloge.tick(60) # Pour animer avec 60 images pas seconde

pygame.quit() # Terminer pygame
```

Les entités du jeu sont placées dans `liste_entite` :

```
liste_entite = []
liste_entite.append(EntiteDuJeu.BotAnime(pygame.Rect((10,100),(40,80)),[3,3],"Son2.wav"))
liste_entite.append(EntiteDuJeu.BotAnime(pygame.Rect((200,200),(50,100)),[0,2],"Son2.wav"))

liste_entite.append(EntiteDuJeu.ItiAnimeVolant(pygame.Rect((200,50),(50,100)),[3,0],"Son3.wav",
3))
liste_entite.append(EntiteDuJeu.EntiteAnimeeParImages(pygame.Rect((50,100),(100,100)),[5,5],
```

```
6.      "Son4.wav",9,"coq"))
```

Lorsque l'utilisateur clique avec la souris, le programme vérifie si sa position touche à une entité. Si c'est le cas, l'entité pousse un cri de désarroi et est éliminée :

```
1.          if event.type == pygame.MOUSEBUTTONUP : # Utilisateur a cliqué dans la fenêtre ?
2.              x=event.pos[0] # Position x de la souris
3.              y=event.pos[1] # Position y de la souris
4.              for une_entite in liste_entite :
5.                  if une_entite.touche(x,y):
6.                      une_entite.emettre_son()
7.                      liste_entite.remove(une_entite)
```

Ensuite, les entités passent à la prochaine scène par l'appel à la méthode *prochaine_scene*() sur chacune des entités de la liste. Cet exemple illustre bien la notion de surcharge dynamique de cette méthode. Python détermine la version qui est employée en fonction de la classe de l'objet visé :

```
        for une_entite in liste_entite :
            une_entite.prochaine_scene()
```

Les entités sont affichées en appelant `dessiner()` sur chacune d'elle. Encore ici, la surcharge dynamique est réalisée par Python pour déterminer la version de `dessiner()` qui est appropriée au type d'objet.

```
        fenetre.fill(Color('white')) # Dessiner le fond de la surface de dessin
        for une_entite in liste_entite :
            une_entite.dessiner()
```

Exercice. Ajoutez quelques entités à l'exemple précédent. Vous pouvez créer de nouvelles classes à cet effet. Cherchez à produire des mouvements variés.

Exercice. Ajoutez un raffinement où deux entités ne peuvent se chevaucher. Par exemple, les entités peuvent rebondir lorsqu'elles se frappent.

9 Exceptions

Les exceptions ont été introduites à la section 2.3. Ce chapitre montre comment traiter les exceptions par programmation. Lorsqu'une condition anormale se produit, un programme Python peut lever une exception avec l'énoncé `raise`. Plusieurs exceptions prédéfinies existent. Une exception prédéfinie est un objet d'une classe qui doit être une sous-classe de la classe `BaseException`. Lorsqu'une exception est levée, le résultat est une interruption du programme et l'affichage d'un message explicatif. D'autre part, il est aussi possible de traiter une exception dans le programme lui-même sans provoquer son interruption. L'énoncé `try` est prévu à cet effet.

9.1 Try, except, raise et classes d'exceptions

L'exemple suivant reprend un exemple vu précédemment. Il faut lire une série d'entiers jusqu'à ce que l'entier 0 soit entré, et produire la somme de ces entiers. Une nouvelle spécification est ajoutée : dans le cas où l'utilisateur entre une chaîne qui n'est pas un nombre entier, elle est tout simplement exclue plutôt que de provoquer une interruption du programme.

CodePython/chapitre9/ExempleTry.py

```
"""
Lire une suite d'entiers jusqu'à ce que l'entier 0 soit entré et afficher la somme
des entiers lus. Ignorer les données erronées. Exemple du try.
"""

somme=0
while True:
    chaine = input("Entrez un nombre entier, 0 pour terminer :")
    if chaine != '0' :
        try :
            somme = somme + int(chaine)
        except ValueError :
            print("La chaîne '" + chaine + "' n'est pas un nombre et sera exclue")
    else:
        break
print("La somme des entiers est:",somme)
```

L'entête `try` est suivi d'un bloc d'énoncés dont l'exécution est tentée en anticipant la possibilité qu'une exception soit levée. Dans notre exemple, il y a un seul énoncé dans le bloc :

```
        try :
            somme = somme + int(chaine)
```

Si une exception est levée dans le bloc, elle peut être attrapée par une clause `except`. Dans notre exemple, il y a une seule clause `except` :

```
        except ValueError :
```

```
            print("La chaîne '" + chaine + "' n'est pas un nombre et sera
exclue")
```

La clause `except` spécifie le type d'exception qui est visé. Elle est suivie d'un bloc d'énoncés qui sont exécutés dans le cas où l'exception du type visé est levée dans le `try`. Dans notre exemple, l'exception `ValueError` peut être levée si la chaine passée à `int()` ne représente pas un nombre entier. L'exception est alors attrapée par la clause `except ValueError`. Ceci conduit à l'affichage du message avec `print()`. L'exécution du programme se poursuit en passant à l'itération suivante du `while`. Dans la version précédente de cet exemple, le programme aurait été interrompu par l'`exception`.

Il est possible de définir de nouveaux types d'exceptions. Une classe exception définie par l'utilisateur doit être une sous-classe de la classe `Exception` qui est une sous-classe de `BaseException`. D'autre part, l'énoncé `raise` permet de lever une exception. Le module suivant ajoute des classes d'exceptions au module `EntiteDuJeu` du chapitre précédent.

CodePython/chapitre9/EntiteDuJeuAvecException.py

```
 1. """
 2. Module qui contient la hiérarchie des classes EntiteAnimee : exemples d'exceptions
 3. """
 4. import pygame
 5. from pygame import Color
 6.
 7. class Erreur(Exception):
 8.     """ Classe de base pour les exceptions de ce module
 9.     """
10.     pass
11.
12. class CoordonneesEntiteErreur(Erreur):
13.     """ Les coordonnées dépassent la fenetre d'animation
14.     """
15.     def __init__(self,position):
16.         self.position = position
17.
18. class TailleExcessiveErreur(Erreur):
19.     """ La taille de l'entité est excessive par rapport à la fenetre d'animation
20.     """
21.     def __init__(self, taille):
22.         self.taille=taille
23.
24. class EntiteAnimeeAvecSon :
25.     """ Un objet représente une entité qui est animée dans une fenêtre Pygame.
26.
27.     L'entité est inscrite dans le rectangle englobant r. Il se déplace en diagonale selon
la vitesse v.
28.     Elle émet un son lorsque supprimée.
29.         r : pygame.Rect       Le rectangle englobant
30.         v : [int,int]         Vitesse de déplacement selon les deux axes x et y
31.         son : pygame.mixer.Sound
32.     """
33.
34.     @staticmethod
35.     def set_fenetre(fenetre):
36.         """ Fixer la variable de classe f qui représente la fenetre graphique
37.
38.             fenetre : pygame.Surface
39.         """
```

```
40.         EntiteAnimeeAvecSon.f = fenetre
41.
42.     def __init__(self,rectangle,vitesse,fichier_son):
43.         if rectangle.x < 0 or rectangle.y < 0 or rectangle.x >
EntiteAnimeeAvecSon.f.get_width() or rectangle.y > EntiteAnimeeAvecSon.f.get_height() :
44.             raise CoordonneesEntiteErreur((rectangle.x,rectangle.y))
45.         elif rectangle.width > EntiteAnimeeAvecSon.f.get_width() or rectangle.height >
EntiteAnimeeAvecSon.f.get_height() :
46.             raise TailleExcessiveErreur(rectangle.size)
47.         else:
48.             self.r = rectangle
49.             self.v = vitesse
50.             self.son = pygame.mixer.Sound(fichier_son)
51.
52.     def prochaine_scene(self):
53.         """ Déplacer selon self.v en diagonale en rebondissant sur les bords de la
fenetre"""
54.         if self.r.x+self.v[0] > EntiteAnimeeAvecSon.f.get_width()-self.r.width or
self.r.x+self.v[0] < 0 :
55.             self.v[0] = -self.v[0] # Inverser la direction en x
56.         self.r.x = self.r.x+self.v[0]
57.         if self.r.y+self.v[1] > EntiteAnimeeAvecSon.f.get_height()-self.r.height or
self.r.y+self.v[1] < 0 :
58.             self.v[1] = -self.v[1] # Inverser la direction en y
59.         self.r.y = self.r.y+self.v[1]
60.
61.     def touche(self,x,y):
62.         return ((x >= self.r.x) and (x <= self.r.x + self.r.width) and (y >= self.r.y) and
(y <= self.r.y + self.r.height))
63.
64.     def emettre_son(self):
65.         self.son.play()
66.
67. class EntiteAvecEtat(EntiteAnimeeAvecSon):
68.     def __init__(self,rectangle,vitesse,fichier_son,nombre_etats):
69.         super().__init__(rectangle,vitesse,fichier_son)
70.         self.nombre_etats = nombre_etats
71.         self.etat_courant = 0
72.
73.     def prochaine_scene(self):
74.         self.etat_courant=(self.etat_courant+1)%self.nombre_etats
75.         super().prochaine_scene()
76.
77. class BotAnime(EntiteAnimeeAvecSon) :
78.     """ Un objet représente un Bot qui est animé dans une fenêtre Pygame
79.         Sous-classe de EntiteAnimee
80.     """
81.
82.     def dessiner(self):
83.         """ Dessiner un Bot.
84.
85.         Le Bot est inscrit dans le rectangle englobant défini par la variable d'objet r
dans une fenetre de Pygame
86.         """
87.
88.         pygame.draw.ellipse(BotAnime.f, Color('green'),
((self.r.x,self.r.y),(self.r.width, self.r.height/2))) # Dessiner la tête
89.         pygame.draw.rect(BotAnime.f, Color('black'),
((self.r.x+self.r.width/4,self.r.y+self.r.height/8),(self.r.width/10,self.r.height/20))) #
L'oeil gauche
90.         pygame.draw.rect(BotAnime.f, Color('black'), ((self.r.x+self.r.width*3/4-
self.r.width/10,self.r.y+self.r.height/8),(self.r.width/10,self.r.height/20))) # L'oeil droit
91.         pygame.draw.line(BotAnime.f, Color('black'),
(self.r.x+self.r.width/4,self.r.y+self.r.height*3/8),(self.r.x+self.r.width*3/4,self.r.y+self.r
.height*3/8), 2) # La bouche
92.         pygame.draw.rect(BotAnime.f, Color('red'),
((self.r.x,self.r.y+self.r.height/2),(self.r.width,self.r.height/2))) # Le corps
93.
94.
95. class ItiAnimeVolant(EntiteAvecEtat) :
96.     """ Un objet représente un Iti qui est animé dans une fenêtre Pygame
97.         Sous-classe de EntiteAnimee
98.     """
99.
100.     def dessiner(self):
```

```
101.         """ Dessiner un Iti.
102.
103.         Le Iti est inscrit dans le rectangle englobant défini par la variable d'objet r
dans une fenetre de Pygame
104.         L'etat courant détermine la hauteur des bras.
105.         """
106.         self.milieux = self.r.x + self.r.width/2;
107.         self.milieuy = self.r.y + self.r.height/2;
108.
109.         pygame.draw.ellipse(ItiAnimeVolant.f, Color('pink'),
((self.r.x+self.r.width/3,self.r.y),(self.r.width/3,self.r.height/4))) # Dessiner la tête
110.         pygame.draw.arc(ItiAnimeVolant.f,Color('black'),((self.milieux-
self.r.width/12,self.r.y+self.r.height/8),(self.r.width/6,self.r.height/14)),3.1416,0,2) # Le
sourire
111.         pygame.draw.ellipse(ItiAnimeVolant.f, Color('black'), ((self.milieux-
self.r.width/8,self.r.y+self.r.height/12),(self.r.width/12,self.r.height/24))) # L'oeil gauche
112.         pygame.draw.ellipse(ItiAnimeVolant.f, Color('black'),
((self.milieux+self.r.width/8-
self.r.width/12,self.r.y+self.r.height/12),(self.r.width/12,self.r.height/24))) # L'oeil droit
113.         pygame.draw.line(ItiAnimeVolant.f, Color('black'),
(self.milieux,self.r.y+self.r.height/4),(self.milieux,self.r.y+self.r.height*3/4), 2) # Le
corps
114.         pygame.draw.line(ItiAnimeVolant.f, Color('black'),
(self.r.x,self.r.y+self.r.height/4+(self.r.height/4)*self.etat_courant),(self.milieux,self.mili
euy), 2) # Bras gauche
115.         pygame.draw.line(ItiAnimeVolant.f, Color('black'),
(self.r.x+self.r.width,self.r.y+self.r.height/4+(self.r.height/4)*self.etat_courant),(self.mili
eux,self.milieuy), 2) # Bras droit
116.         pygame.draw.line(ItiAnimeVolant.f, Color('black'),
(self.r.x,self.r.y+self.r.height),(self.milieux,self.r.y+self.r.height*3/4), 2) # Jambe gauche
117.         pygame.draw.line(ItiAnimeVolant.f, Color('black'),
(self.r.x+self.r.width,self.r.y+self.r.height),(self.milieux,self.r.y+self.r.height*3/4), 2) #
Jambe droite
118.
119.
120. class EntiteAnimeeParImages(EntiteAvecEtat):
121.         def __init__(self,rectangle,vitesse,fichier_son,nombre_etats,nom_dossier):
122.             super().__init__(rectangle,vitesse,fichier_son,nombre_etats)
123.             self.image_animation = []
124.             for i in range(nombre_etats):
125.
self.image_animation.append(pygame.transform.scale(pygame.image.load(nom_dossier+"/"+nom_dossie
r+str(i+1)+".gif"),(self.r.width,self.r.height)))
126.         def dessiner(self):
127.
EntiteAnimeeParImages.f.blit(self.image_animation[self.etat_courant],[self.r.x,self.r.y])
128. …
129.
```

Une convention non obligatoire consiste à créer une exception générale `Erreur (Error)` qui est la racine de la hiérarchie des exceptions d'un module. Pour créer une nouvelle exception, il faut créer une classe qui hérite de la classe `Exception` :

```
class Erreur(Exception):
    """ Classe de base pour les exceptions de ce module
    """
    pass
```

L'énoncé `pass` ne fait rien. Il est employé en Python lorsqu'aucun traitement n'est prévu mais que la syntaxe exige un énoncé. La classe suivante définit l'exception `CoordonneesEntiteErreur` qui est levée à la création d'une entité lorsque ses coordonnées `(x,y)` dépassent le cadre de la fenêtre d'affichage.

```
class CoordonneesEntiteErreur(Erreur):
    """ Les coordonnées dépassent la fenetre d'animation
    """
    def __init__(self,position):
        self.position = position
```

Les coordonnées fautives `x` et `y` sont passées au constructeur. Elles pourront ainsi être affichées dans le message d'erreur si l'exception est attrapée. L'exception est levée dans le constructeur de la classe `EntiteAnimeeAvecSon`. Les coordonnées `x` et `y` sont passées au constructeur de l'exception :

```
        if rectangle.x < 0 or rectangle.y < 0 or rectangle.x > EntiteAnimeeAvecSon.f.get_width() or rectangle.y > EntiteAnimeeAvecSon.f.get_height() :
            raise CoordonneesEntiteErreur((rectangle.x,rectangle.y))
```

L'exception `TailleExcessiveErreur` est prévue pour le cas où la `hauteur` ou la `largeur` de l'entité dépasse la taille de la fenêtre.

```
class TailleExcessiveErreur(Erreur):
    """ La taille de l'entité est excessive par rapport à la fenetre d'animation
    """
    def __init__(self, largeur, hauteur):
        self.largeur = largeur
        self.hauteur = hauteur
```

Elle est aussi levée dans le constructeur de la classe `EntiteAnimeeAvecSon` :

```
        elif rectangle.width > EntiteAnimeeAvecSon.f.get_width() or rectangle.height > EntiteAnimeeAvecSon.f.get_height() :
            raise TailleExcessiveErreur(rectangle.size)
```

Le programme *ExempleJeuSimpleAvecException* lève les exceptions créées dans le module précédent :

CodePython/chapitre9/ExempleJeuSimpleAvecException.py

```
"""
Exemple de jeu avec exceptions: programme principal
"""
import EntiteDuJeuAvecException
import pygame
from pygame import Color
pygame.init() # Initialiser les modules de Pygame
LARGEUR_FENETRE = 400
HAUTEUR_FENETRE = 600
fenetre = pygame.display.set_mode((LARGEUR_FENETRE, HAUTEUR_FENETRE)) # Ouvrir la fenêtre
EntiteDuJeuAvecException.EntiteAnimeeAvecSon.set_fenetre(fenetre)
pygame.display.set_caption("Exemple de jeu avec module EntiteDuJeu")
horloge = pygame.time.Clock() # Pour contrôler la fréquence des scènes

# Création de la liste des entités du jeu
liste_entite = []
try :
```

```
18.
liste_entite.append(EntiteDuJeuAvecException.BotAnime(pygame.Rect((10,100),(40,80)),[3,3],"Son2
.wav"))
19.
liste_entite.append(EntiteDuJeuAvecException.BotAnime(pygame.Rect((200,200),(50,100)),[0,2],"So
n2.wav"))
20.
liste_entite.append(EntiteDuJeuAvecException.ItiAnimeVolant(pygame.Rect((200,50),(50,100)),[3,0
],"Son3.wav",3))
21.
liste_entite.append(EntiteDuJeuAvecException.EntiteAnimeeParImages(pygame.Rect((50,100),(100,10
0)),[5,5],"Son4.wav",9,"coq"))
22. except EntiteDuJeuAvecException.CoordonneesEntiteErreur as e :
23.     print("Les coordonnées de l'entité dépassent la taille de la fenetre",e)
24. except EntiteDuJeuAvecException.TailleExcessiveErreur as e :
25.     print("L'entité a une taille excessive par rapport à la taille de la fenetre",e)
26. except :
27.     print("Une exception a été levée lors de la création des entités du jeu")
28.     raise
29. else :
30.     # Boucle d'animation
31.     fin = False
32.     while not fin :
33.         event = pygame.event.poll() # Chercher le prochain évènement à traiter
34.         if event.type == pygame.QUIT:  # Utilisateur a cliqué sur la fermeture de fenêtre ?
35.             fin = True  # Fin de la boucle du jeu
36.         else :
37.             if event.type == pygame.MOUSEBUTTONUP : # Utilisateur a cliqué dans la fenêtre
?
38.                 x=event.pos[0] # Position x de la souris
39.                 y=event.pos[1] # Position y de la souris
40.                 for une_entite in liste_entite :
41.                     if une_entite.touche(x,y):
42.                         une_entite.emettre_son()
43.                         liste_entite.remove(une_entite)
44.
45.             for une_entite in liste_entite :
46.                 une_entite.prochaine_scene()
47.
48.             fenetre.fill(Color('white')) # Dessiner le fond de la surface de dessin
49.             for une_entite in liste_entite :
50.                 une_entite.dessiner()
51.
52.             pygame.display.flip() # Mettre à jour la fenêtre graphique
53.             horloge.tick(60) # Pour animer avec 60 images pas seconde
54. finally:
55.     pygame.quit() # Terminer pygame
56.
```

Le `try` peut contenir plusieurs clauses `except` les unes à la suite des autres pour attraper divers types d'exceptions comme c'est le cas du `try` de l'exemple. Une clause `except` sans type peut terminer la liste pour attraper n'importe quel type d'exception qui n'a pas été attrapée dans les clauses précédentes :

```
except :
    print("Une exception a été levée lors de la création des entités du jeu")
    raise
```

Dans cet exemple, un message est affiché et l'exception qui a été attrapée est relancée à nouveau par l'énoncé `raise`. Ceci provoquera un arrêt du programme et l'affichage de la trace de l'exception.

Une clause `else` peut suivre le dernier `except` et elle est exécutée si aucune exception n'est attrapée. Dans notre exemple, la boucle d'animation est alors démarrée.

Enfin une clause `finally` peut être ajoutée. Elle sera exécutée qu'une exception ne soit attrapée ou pas. Dans notre exemple, le module `Pygame` est alors terminé, ce qui provoque la fermeture de la fenêtre dans tous les cas :

```
finally :
    pygame.quit() # Terminer pygame
```

Exercice. Faites des tests pour soulever chacune des exceptions de l'exemple.

Exercice. Créez une nouvelle classe pour une exception qui représente le cas où la vitesse de l'entité est excessive, c'est-à-dire qu'elle dépasse 20% de la taille de la fenêtre. Soulevez l'exception dans le constructeur de la classe `EntiteAnimeeAvecSon`. Attrapez l'exception dans le programme et affichez un message à cet effet comme pour les autres cas d'exceptions de l'exemple.

9.2 Assert et programmation défensive

La programmation défensive est une approche qui consiste à ajouter des vérifications dans un programme pour assurer que des conditions non cohérentes avec l'intention d'un programme ne soient réalisées par erreur. Ceci permet de détecter un bogue dans le code pendant son exécution. L'énoncé `assert` est un mécanisme bien adapté à cette approche en permettant de vérifier si une condition est respectée et de lever une exception `AssertionError` dans le cas contraire. Le `assert` est désactivé si la variable native `__debug__` est `False`. Par défaut elle est `True`, mais elle est `False` si une exécution optimisée est demandée (option `-o` au démarrage de Python).

L'exemple suivant lit deux entiers pour les diviser et vérifie que le diviseur est non nul par un `assert`. Le premier paramètre du `assert` est la condition à vérifier et le second, un message d'erreur. Si la condition est fausse, une exception est levée et le message d'erreur est affiché.

CodePython/chapitre9/ExempleAssert.py

```
1. '''
2. Ce programme saisit deux entiers et en affiche le quotient.
Exemple de assert.
3. '''
4.
5. chaine1 = input("Entrez le dividende :")
6. chaine2 = input("Entrez le diviseur :")
7.
```

```
 8. dividende = int(chaine1)
 9. diviseur = int(chaine2)
10.
11. assert diviseur != 0, "Le diviseur ne peut être 0"
12.
13. quotient = dividende / diviseur
14. print("Le quotient est ",quotient)
15.
```

L'exécution avec un diviseur nul lève l'exception et affiche le message suivant :

```
AssertionError: Le diviseur ne peut être 0
```

Dans l'exemple de code suivant, deux `assert` sont ajoutés au code de l'exemple de jeu de la section précédente afin de vérifier que l'entité ne déborde pas du cadre du jeu après avoir modifié ses coordonnées. Pour illustrer le fonctionnement du `assert`, un bogue est introduit dans l'inversion de la vitesse en mettant un `+` plutôt qu'un `-`. L'entité va donc déborder du cadre de dessin, ce qui sera détecté par un `assert`. Les modifications au code précédent sont surlignées en jaune et le bogue en rouge.

CodePython/chapitre9/EntiteDuJeuAssert.py

```
 1. """
 2. Module qui contient la hiérarchie des classes EntiteAnimee : exemple de assert
 3. """
 4. import pygame
 5. from pygame import Color
 6.
 7. class Erreur(Exception):
 8.     """ Classe de base pour les exceptions de ce module
 9.     """
10.     pass
11.
12. class CoordonneesEntiteErreur(Erreur):
13.     """ Les coordonnées dépassent la fenetre d'animation
14.     """
15.     def __init__(self,position):
16.         self.position = position
17.
18. class TailleExcessiveErreur(Erreur):
19.     """ La taille de l'entité est excessive par rapport à la fenetre d'animation
20.     """
21.     def __init__(self, taille):
22.         self.taille=taille
23.
24. class EntiteAnimeeAvecSon :
25.     """ Un objet représente une entité qui est animée dans une fenêtre Pygame.
26.
27.     L'entité est inscrite dans le rectangle englobant r. Il se déplace en diagonale selon
la vitesse v.
28.     Elle émet un son lorsque supprimée.
29.         r : pygame.Rect       Le rectangle englobant
30.         v : [int,int]         Vitesse de déplacement selon les deux axes x et y
31.         son : pygame.mixer.Sound
32.     """
33.
34.     @staticmethod
35.     def set_fenetre(fenetre):
36.         """ Fixer la variable de classe f qui représente la fenetre graphique
```

```
37.
38.             fenetre : pygame.Surface
39.         """
40.         EntiteAnimeeAvecSon.f = fenetre
41.
42.     def __init__(self,rectangle,vitesse,fichier_son):
43.         if rectangle.x < 0 or rectangle.y < 0 or rectangle.x >
EntiteAnimeeAvecSon.f.get_width() or rectangle.y > EntiteAnimeeAvecSon.f.get_height() :
44.             raise CoordonneesEntiteErreur((rectangle.x,rectangle.y))
45.         elif rectangle.width > EntiteAnimeeAvecSon.f.get_width() or rectangle.height >
EntiteAnimeeAvecSon.f.get_height() :
46.             raise TailleExcessiveErreur(rectangle.size)
47.         else:
48.             self.r = rectangle
49.             self.v = vitesse
50.             self.son = pygame.mixer.Sound(fichier_son)
51.
52.     def prochaine_scene(self):
53.         """ Déplacer selon self.v en diagonale en rebondissant sur les bords de la
fenetre"""
54.         if self.r.x+self.v[0] > EntiteAnimeeAvecSon.f.get_width()-self.r.width or
self.r.x+self.v[0] < 0 :
55.             self.v[0] = +self.v[0] # Inverser la direction en x (bug introduit pour
l'exemple !)
56.         self.r.x = self.r.x+self.v[0]
57.         if self.r.y+self.v[1] > EntiteAnimeeAvecSon.f.get_height()-self.r.height or
self.r.y+self.v[1] < 0 :
58.             self.v[1] = -self.v[1] # Inverser la direction en y
59.         self.r.y = self.r.y+self.v[1]
60.         assert self.r.x >= 0 and self.r.x <= EntiteAnimeeAvecSon.f.get_width()-
self.r.width, "La position x dépasse du cadre du jeu"
61.         assert self.r.y >= 0 and self.r.y <= EntiteAnimeeAvecSon.f.get_height()-
self.r.height, "La position y dépasse du cadre du jeu"
62.
63.     def touche(self,x,y):
64.         return ((x >= self.r.x) and (x <= self.r.x + self.r.width) and (y >= self.r.y) and
(y <= self.r.y + self.r.height))
65.
66. …
67.
```

Le programme principal est modifié en important le module qui contient les modifications.

CodePython/chapitre9/ExempleDuJeuAvecAssert.py

```
 1. """
 2. Exemple de jeu avec assert: programme principal
 3. """
 4. import EntiteDuJeuAssert
 5. import pygame
 6. from pygame import Color
 7.
 8. pygame.init() # Initialiser les modules de Pygame
 9. LARGEUR_FENETRE = 400
10. HAUTEUR_FENETRE = 600
11. …
12.
```

L'exécution avec l'erreur de codage lève l'exception et affiche :

```
AssertionError: La position x dépasse du cadre du jeu
```

9.3 Développement de tests avec unittest

Plutôt que d'introduire des vérifications dans le code, l'emploi de tests pendant le développement d'un logiciel permet de détecter des anomalies avant de déployer celui-ci dans son environnement d'exécution sans encombrer le code lui-même. Des logiciels permettent de faciliter le processus de vérification par l'utilisation systématique de tests. Le module `unittest` de Python permet le développement de tests unitaires.

Test Unitaire (*Unit Test*)

Un test unitaire permet de tester une partie d'un logiciel, comme une fonction, une méthode, une classe ou un module.

Prenons l'exemple du développement de la fonction `surface()` introduit à la section 5.2.4.

CodePython/chapitre9/ExempleFonctionSurface.py

```
1. """
2. Exemple de fonction surface qui retourne une valeur
3. """
4. def surface(largeur, hauteur):
5.     return largeur*hauteur
6. s = surface(3,5)
7. print("La surface est :",s)
8.
```

Pour vérifier le bon fonctionnement de la fonction `surface()`, on peut coder un ensemble de tests en exploitant le module `unittest` :

CodePython/chapitre9/TestExempleFonctionSurface.py

```
 1. """
 2. Exemple d'utilisation de unittest
 3. """
 4.
 5. import ExempleFonctionSurface
 6. import unittest
 7.
 8. class TestFonctionSurface(unittest.TestCase):
 9.
10.     def test_1(self):
11.         self.assertEqual(ExempleFonctionSurface.surface(3,5),
15)
12.
13.     def test_2(self):
```

```
14.         self.assertEqual(ExempleFonctionSurface.surface(2.5,4),
10)
15.
16.     def test_negatif(self):
17.         with self.assertRaises(Exception):
18.             ExempleFonctionSurface.surface(-3,4)
19.
20. unittest.main()
21.
```

Résultat :

```
..FLa surface est : 15

======================================================================
FAIL: test_negatif (__main__.TestFonctionSurface)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\Robert\CodePython\TestExempleFnctionSurface.py", line 19, in
test_negatif
    ExempleFonctionSurface.surface(-3,4)
AssertionError: Exception not raised

----------------------------------------------------------------------
Ran 3 tests in 0.002s

FAILED (failures=1)
```

La classe `TestFonctionSurface` qui contient les tests doit hériter de `unittest.TestCase`. Chacun des tests correspond à une méthode définie dans cette classe. Par convention, le nom d'une méthode de test débute par `test_` suivi d'une chaîne qui sert à identifier le test. Chacune de ces fonctions fait un appel à une méthode prédéfinie `assert` qui vérifie un aspect de l'exécution de l'élément à tester. Dans le code suivant, la fonction `assertEqual()` vérifie l'égalité entre le résultat de l'appel à fonction `ExempleFonctionSurface.surface(3,5)` et le résultat attendu 15.

```
self.assertEqual(ExempleFonctionSurface.surface(3,5), 15)
```

Dans la ligne suivante, la fonction vérifie si une exception est levée par l'appel à `surface(-3,4)`, ce qui devrait être le cas pour une valeur négative.

```
        with self.assertRaises(Exception):
            ExempleFonctionSurface.surface(-3,4)
```

Plusieurs autres méthodes `assert` sont disponibles dans la classe `unittest.TestCase`.

La méthode `unittest.main()` provoque l'exécution de chacun des tests et affiche un rapport. Dans notre exemple, les deux premiers tests sont

exécutés avec succès mais le troisième fait ressortir un problème parce que la fonction ne lève pas d'exception avec un argument négatif.

Exercice. Modifiez la fonction `surface()` de manière à ce qu'une exception soit levée si un des arguments a une valeur négative.

Solution minimaliste :

CodePython/chapitre9/ExempleFonctionSurfaceExceptionNeg.py

```
1. """
2. Exemple de fonction surface qui retourne une valeur
3. Exception si un argument est négatif
4. """
5. def surface(largeur, hauteur):
6.     if largeur < 0 or hauteur < 0 :
7.         raise(Exception)
8.     return largeur*hauteur
9.
```

En retestant le code modifié, le résultat suivant devrait montrer son bon fonctionnement :

```
----------------------------------------------------------------------
Ran 3 tests in 0.002s

OK
```

Exercice. Raffiner les exceptions levées en affichant un message explicatif au sujet de l'exception.

Dans le cadre d'un processus de développement de logiciel de qualité, une stratégie de développement très performante consiste à effectuer un ensemble de tests à chacune des mises à jour pour détecter des anomalies qui pourraient être introduites suite aux modifications.

Méthode de développement pilotée par les tests (*Test-Driven Development*)

La méthode de développement de logiciels pilotée par les tests met l'emphase sur les tests dans le processus de développement. Le logiciel est produit de manière itérative et incrémentale en codant les tests avant chacune des itérations du développement et en effectuant les tests après chacune des modifications du code avant de passer à l'itération suivante.

10 Formatage et analyse de chaînes de caractères

Ce chapitre approfondie la manipulation des chaînes de caractères. La première section traite du formatage des chaînes et la seconde de l'analyse avec les expressions régulières.

10.1 Formatage de chaînes de caractères

La fonction `print()` utilisée jusqu'ici a une capacité limitée de formatage et fait appel à la fonction `str()` pour produire une représentation sous forme de chaîne `str`. La fonction `str()` retourne une représentation de n'importe quel objet sous forme d'une chaîne de caractère en vue d'une consommation humaine alors que la fonction `repr()` vise une représentation sous forme d'une chaîne de caractère que l'interprète peut facilement manipuler. Lorsqu'une expression est exécutée dans le *shell*, c'est la fonction `repr()` qui est appelée pour afficher le résultat. Dans plusieurs cas, les deux fonctions retournent la même chose. Pour les classes définies par l'utilisateur, il est possible de redéfinir ces fonctions au besoin.

La méthode `format()`, introduite dans la version 3 de Python, permet de composer des chaînes de caractères d'une manière plus élaborée en employant des opérations de formatage. Ceci remplace l'ancienne approche avec le `%` qui est issue du langage C. Le formatage est utile pour produire des résultats faciles à lire et à interpréter. Une chaîne à formater contient du texte fixe avec des parties variables appelées *champs de remplacement* (*replacement field*). Un champ de remplacement est désigné par une paire d'accolades qui contient une spécification de formatage possiblement vide. La spécification de formatage désigne un élément à formatter et la manière de le formatter.

```
>>> un_entier = 230
>>> "Voici un entier: {}".format(un_entier)
'Voici un entier: 230'
```

Comme la spécification entre `{}` est vide, c'est le résultat de la fonction `str()` appliquée à l'argument de `format()` qui est employé directement et qui remplace les accolades dans la chaîne à formatter.

Il est possible d'inclure plusieurs champs de remplacement. Les valeurs à substituer suivent l'ordre des arguments.

```
>>> 'La somme de {} et {} est {}'.format(2,3,2+3)
'La somme de 2 et 3 est 5'
```

Il est aussi possible de faire référence aux arguments de `format()` en désignant leur position par un indice. Ceci permet de changer l'ordre des valeurs à substituer.

```
>>> 'La somme de {1} et {2} est {0}'.format(2+3,2,3)
'La somme de 2 et 3 est 5'
```

Il est aussi possible de faire référence aux noms des paramètres dans le cas de paramètres nommés.

```
>>> 'La somme de {a} et {b} est {c}'.format(c=20+35,a=20,b=35)
'La somme de 20 et 35 est 55'
```

Plusieurs options de formatage sont disponibles pour chacun des champs. Après la désignation de la valeur, il est possible de mettre un `:` suivi d'une spécification formulée à l'aide du *mini langage de spécification de format* (https://docs.python.org/fr/3/library/string.html#formatspec). Dans l'exemple suivant, `>15` aligne la valeur à droite dans un champ de taille `15`. `12` spécifie un champ de taille `12` exactement. Ce genre de formatage est utile pour produire un tableau de données dont les champs sont alignés.

```
>>> '|{:>15}|{:>15}|{:12}|'.format('Nom', 'Prenom', 'Téléphone')
'|            Nom|         Prenom|Téléphone   |'
>>>   '|{nom:>15}|{prenom:>15}|{tel:12}|'.format(nom='Gingras',prenom='Guy',
tel='514-333-4444')
'|        Gingras|            Guy|514-333-4444|'
```

Voir la documentation pour les nombreuses options offertes.

f-string

Depuis la version 3.6, Python a introduit le formatage avec les `f-strings` qui offre une syntaxe simplifiée par rapport à `format()` et qui est maintenant la manière suggérée. Un `f-string` débute par `f` ou `F` et contient des champs de remplacement comme pour `format()`. Par opposition à `format()`, une valeur est exprimée directement par une expression à évaluer entre les accolades:

```
>>> f'La somme de {2} et {3} est {2+3}'
'La somme de 2 et 3 est 5'
>>> f"{'Gingras':>15}|{'Guy':>15}|{'514-333-4444':12}|"
'        Gingras|            Guy|514-333-4444|'
```

10.2 Analyse et extraction de chaînes de caractères avec les expressions régulières (module re)

Il est souvent nécessaire de vérifier qu'une chaîne de caractères possède une structure particulière. Par exemple, on peut vouloir vérifier qu'un code postal a une structure valide de la forme « LCL CLC » où C est un chiffre

et L une lettre majuscule. Le module `re`[17] permet d'effectuer des appariements de chaînes sophistiqués basés sur la notion d'expression régulière (*regular expression*) aussi appelée *expression rationnelle*.

```
>>> import re
>>> code_postal_re = re.compile(r'[A-Z][0-9][A-Z] [0-9][A-Z][0-9]')
>>> code_postal_re.match('J2S 4H8')
<_sre.SRE_Match object; span=(0, 7), match='J2S 4H8'>
>>> code_postal_re.match('JDS 4H8')
```

Le motif à vérifier est exprimé dans un langage d'expression de motifs. Dans l'exemple précédent, le motif est :

```
r'[A-Z][0-9][A-Z] [0-9][A-Z][0-9]'
```

Il débute par `r`, suivi entre apostrophes d'une séquence de motifs à apparier. Le `r` sert à préciser que la chaîne n'emploie pas le `\` comme début d'une séquence d'échappement. Ceci est important dans le contexte des motifs où le `\` a une autre signification comme nous le verrons plus loin.

Pour apparier une lettre majuscule, on emploie la syntaxe `[A-Z]` qui représente un caractère dans l'ensemble des caractères entre `A` et `Z`. La syntaxe `[0-9]` représente un chiffre entre `0` et `9`. Pour le caractère espace, on n'a qu'à placer un espace dans le motif. L'appel à la fonction `re.compile()` compile le motif en un objet `re.Pattern`. La méthode `match()` de l'objet `re.Pattern`, effectue l'appariement du motif compilé avec une chaîne passée en paramètre. Elle retourne un objet `re.Match` si l'appariement a été effectué avec succès et `None` dans le cas contraire. L'appariement avec `match()` se fait à partir du début de la chaîne. L'attribut `span` de l'objet `Match` indique la position où l'appariement a été effectué.

Il est aussi possible d'effectuer l'appariement en appelant la fonction `re.match(motif, chaîne)` :

```
>>> re.match(r'[A-Z][0-9][A-Z] [0-9][A-Z][0-9]','J2S 4H8')
<_sre.SRE_Match object; span=(0, 7), match='J2S 4H8'>
```

Avec la fonction `re.match(motif, chaîne)`, le motif est recompilé à chacun des appels, ce qui est moins performant que la compilation à part, si l'appariement est effectué à répétition.

Voici quelques notions de base au sujet de la formation d'un motif. Ceci n'est qu'un aperçu des principales fonctions. Pour plus de détails, voir la documentation du module `re`.

[17] Le module regex est compatible avec re et offre des fonctionnalités étendues.

Motif simple

Un caractère quelconque est un motif.

```
>>> re.match(r'd','d')
<_sre.SRE_Match object; span=(0, 1), match='d'>
>>> re.match(r'd','dtaj')
<_sre.SRE_Match object; span=(0, 1), match='d'>
>>> re.match(r'd','atdk')
```

Dans le dernier exemple, l'appariement n'a pas fonctionné parce que le avec `match()` effectue celui-ci à partir du début de la chaîne. Le `search()` permet la recherche à n'importe quelle position.

```
>>> re.search(r'd','atdk')
<_sre.SRE_Match object; span=(2, 3), match='d'>
```

Exercice. Vérifiez qu'une chaîne de caractère commence par la lettre `z`.

```
>>> re.match(r'z','zorro')
<re.Match object; span=(0, 1), match='z'>
>>> re.match(r'z','torro')
>>>
```

Exercice. Vérifiez qu'une chaîne de caractère contient la lettre `z`.

```
>>> re.search(r'z','raz')
<re.Match object; span=(2, 3), match='z'>
```

Suite de motifs : $m_1\ m_2 \ldots m_n$

Un motif peut être composé d'une suite de motifs $m_1\ m_2 \ldots m_n$. Ainsi une suite de caractères est un motif.

```
>>> re.match(r'babfc','babfc')
<_sre.SRE_Match object; span=(0, 5), match='babfc'>
```

Exercice. Vérifiez qu'une chaîne contient la chaîne de caractères `Python`.

```
>>> re.search(r'Python','Le langage Python est facile à apprendre')
<re.Match object; span=(11, 17), match='Python'>
```

Caractère spécial `.`

Le `.` désigne un caractère quelconque sauf le `<fin de ligne>`.

```
>>> re.match(r'a..c.d','agsced')
<_sre.SRE_Match object; span=(0, 6), match='agsced'>
>>> re.match(r'a..c.d','afced')
>>>
```

Exercice. Vérifiez qu'une chaîne débute par `x` suivi de deux caractères quelconques suivi de `y`.

```
>>> re.match(r'x..y','xefyab')
<re.Match object; span=(0, 4), match='xefy'>
>>> re.match(r'x..y','xeyab')
>>>
```

Début de chaîne ^ et fin de chaîne $

Le `^` représente le début de la chaîne et `$` la fin.

```
>>> re.match(r':$',':')
<re.Match object; span=(0, 1), match=':'>
>>> re.match(r':$',':a')
>>>
```

Exercice. Vérifiez qu'une chaîne est égale à `'123'`.

```
>>> re.match(r'123$','123')
<re.Match object; span=(0, 3), match='123'>
>>> re.match(r'123$','1234')
>>>
```

Exercice. Vérifiez qu'une chaîne se termine par `'123'`.

```
>>> re.search(r'123$','abc12123')
<re.Match object; span=(5, 8), match='123'>
>>> re.search(r'123$','abc121234')
>>>
```

Ensemble de caractères : [ ]

On peut spécifier un ensemble de caractères en les plaçant entre accolades. On peut mettre un caractère individuel ou encore un intervalle avec la syntaxe c_1-c_2. Il est aussi possible d'exclure un ensemble de caractères en débutant le motif avec un ^. L'ensemble des lettres majuscules ou minuscules peut être désigné par `[a-zA-Z]`. L'ensemble des caractères excluant les lettres est spécifié par `[^a-zA-Z]`.

```
>>> re.match(r'[a-zA-Z]','dodo')
<re.Match object; span=(0, 1), match='d'>
>>> re.match(r'[a-zA-Z]','Dede')
<re.Match object; span=(0, 1), match='D'>
>>> re.match(r'[^0-9]','ABC123')
<re.Match object; span=(0, 1), match='A'>
```

Exercice. Vérifiez qu'une chaîne débute par une lettre majuscule.

```
>>> re.match(r'[A-Z]','Abcd')
<re.Match object; span=(0, 1), match='A'>
```

Répétition de motif : `m*`

Le `m*` désigne la répétition d'un motif `m`, 0 fois ou plus. L'appariement est vorace en ce sens qu'il est effectué avec le nombre maximal d'occurrences dans le cas d'une répétition.

```
>>> re.match(r'a*','aaaa')
<_sre.SRE_Match object; span=(0, 4), match='aaaa'>
>>> re.match(r'a*','')
<_sre.SRE_Match object; span=(0, 0), match=''>
```

Exercice. Trouvez la suite des chiffres au début d'une chaîne.

```
>>> re.match(r'[0-9]*','123xyz')
<re.Match object; span=(0, 3), match='123'>
```

Répétition, une fois ou plus : `m+`

Le **+** désigne la répétition d'un motif, une fois ou plus. Comme pour le *, l'appariement se fait sur le plus grand nombre de répétition.

```
>>> re.match(r'a+','aaaa')
<_sre.SRE_Match object; span=(0, 4), match='aaaa'>
>>> re.match(r'a+','')
>>> re.match(r'fc+','fccc')
<_sre.SRE_Match object; span=(0, 4), match='fccc'>
>>> re.match(r'fc+','fcfc')
<_sre.SRE_Match object; span=(0, 2), match='fc'>
```

Le dernier appariement a uniquement détecté la première occurrence de `'fc'` parce que la répétition est uniquement pour le `'c'` !

Exercice. Vérifiez qu'une chaîne débute par une suite d'une majuscule ou plus.

```
>>> re.match(r'[A-Z]+','Allo')
<re.Match object; span=(0, 1), match='A'>
>>> re.match(r'[A-Z]+','ABCDabcd')
<re.Match object; span=(0, 4), match='ABCD'>
```

0 ou 1 fois : `m?`

Le motif `m?` désigne l'apparition de `m` 0 ou une fois.

```
>>> re.match(r'[A-Z]?','Allo')
<re.Match object; span=(0, 1), match='A'>
>>> re.match(r'[A-Z]?','allo')
<re.Match object; span=(0, 0), match=''>
>>> re.match(r'[A-Z]?','ABCDabcd')
<re.Match object; span=(0, 1), match='A'>
```

Appariement non vorace : `m*?, m+? et m??`

L'appariement avec `*`,`+` et `?` est vorace en ce sens qu'il apparie le plus grand nombre de caractères possibles. Pour limiter l'appariement au nombre minimal de caractères, on emploie `m*?`, `m+?` et `m??`.

```
>>> re.match(r'a.+b','addebsbsb')
<_sre.SRE_Match object; span=(0, 9), match='addebsbsb'>
>>> re.match(r'a.+?b','addebesbsb')
<_sre.SRE_Match object; span=(0, 5), match='added'>
```

Dans le premier cas, toute la chaîne est appariée, alors que dans le deuxième, le nombre minimal de caractères qui correspond au motif est apparié.

Répétition fixe : `m{n}`

Il est possible de fixer le nombre de répétition `n` du motif `m` avec `m{n}`.

```
>>> re.match(r'a{4}','aaaa')
<_sre.SRE_Match object; span=(0, 4), match='aaaa'>
>>> re.match(r'a{4}','aaa')
>>>
```

Exercice. Vérifiez qu'une chaîne débute par une séquence de trois lettres.

```
>>> re.match(r'[A-Z]{3}','ABC123')
<re.Match object; span=(0, 3), match='ABC'>
>>> re.match(r'[A-Z]{3}','AB123')
>>>
```

Intervalle de répétitions : `m{n1,n2}`

La notation `m{n1,n2}`désigne un intervalle de répétitions entre `n1` et `n2`.

```
>>> re.match(r'a{2,4}','aa')
<_sre.SRE_Match object; span=(0, 2), match='aa'>
>>> re.match(r'a{2,4}','aaaa')
<_sre.SRE_Match object; span=(0, 4), match='aaaa'>
>>> re.match(r'a{2,4}','aaaaa')
<_sre.SRE_Match object; span=(0, 4), match='aaaa'>
```

La notation `m{n1,n2}?` est la version non vorace qui apparie le minimum de caractères.

Échappement pour caractère spécial : `\s`

Le `\s` permet d'employer les caractères spéciaux dans les motifs.

```
>>> re.match(r'a\*\*\?b','a**?b')
```

```
<_sre.SRE_Match object; span=(0, 5), match='a**?b'>
```

Exercice. Vérifiez qu'une chaîne contient un point.

```
>>> re.search(r'\.','abc.com')
<re.Match object; span=(3, 4), match='.'>
```

Exercice. Extraire l'extension d'un nom de fichier (suite de 2 à quatre lettres après le dernier `.`).

```
>>> re.search(r'\.[a-zA-Z]{2,4}$','abc.com')
<re.Match object; span=(3, 7), match='.com'>
```

Groupage : `(m)`

Les parenthèses permettent de regrouper une suite de motifs. Ceci peut servir à appliquer une répétition à un groupe.

```
>>> re.match(r'(fc)+','fcfc')
<_sre.SRE_Match object; span=(0, 4), match='fcfc'>
>>> re.match(r'k(ed)+b','kedededb')
<_sre.SRE_Match object; span=(0, 8), match='kedededb'>
```

Exercice. Vérifiez qu'une chaîne est de la forme `CLCLCL` où `C` est un chiffre et `L` une lettre majuscule.

```
>>> re.match(r'([0-9][A-Z]){3}$','1A2C6B')
<re.Match object; span=(0, 6), match='1A2C6B'>
>>> re.match(r'([0-9][A-Z]){3}$','1A2C6B4S')
>>> re.match(r'([0-9][A-Z]){3}$','1AEC6B')
```

Extraction de chaîne : méthode `group()`

Un groupe permet aussi d'extraire une sous-chaîne qui est appariée au groupe. La méthode `group()` appelée sur l'objet `Match` retourne la chaîne complète qui a été appariée. Dans l'exemple suivant, le motif représente une date de la forme `JJ-MM-AAAA`.

```
>>> re.match(r'([0-3][0-9])-([0-1][0-9])-([0-9]{4})','15-10-2015').group()
'15-10-2015'
```

La méthode `group(i)` retourne le ième groupe. L'indice 0 correspond à la chaîne entière qui a été appariée. L'exemple suivant retourne le deuxième groupe formé des deux chiffres du mois.

```
>>> re.match(r'([0-3][0-9])-([0-1][0-9])-([0-9]{4})','15-10-2015').group(2)
'10'
```

S'il y a plusieurs paramètres à la méthode `group(i1, i2, …)`, un `tuple` formé par chacun des groupes appariés est retourné.

```
>>>re.match(r'([0-3][0-9])-([0-1][0-9])-([0-9]{4})','15-10-
2015').group(1,2,3)
('15', '10', '2015')
```

Exercice. Un identifiant est de la forme `LLLLJJMMAACC`, `LLLL` est une séquence de quatre lettres, `JJ` le jour en chiffre, `MM` le mois en chiffres, `AA` les deux derniers chiffres de l'année et `CC` est un numéro de séquence de deux chiffres. Validez qu'une chaîne respecte la syntaxe d'un identifiant et extrayez chacun des groupes si la chaîne est valide.

```
>>>re.match(r'([A-Z]{4})([0-3][0-9])([0-1][0-9])([0-9]{2})([0-
9]{2})','BEDA10069001').group(1,2,3,4,5)
('BEDA', '10', '06', '90', '01')
```

Alternatives : $m_1|m_2$

L'expression $m_1|m_2$ permet de représenter une alternative entre deux expressions.

```
>>> re.match(r'[A-Z][0-9][A-Z]|[0-9][0-9][0-9]','B6R')
<re.Match object; span=(0, 3), match='B6R'>
>>> re.match(r'[A-Z][0-9][A-Z]|[0-9][0-9][0-9]','234')
<re.Match object; span=(0, 3), match='234'>
>>> re.match(r'[A-Z][0-9][A-Z]|[0-9][0-9][0-9]','2A4')
```

11 Traitement de fichiers

Pour conserver des données de manière persistante au-delà de l'exécution des programmes, il faut employer les mémoires secondaires. Les langages de programmation fournissent des interfaces programmatiques pour la manipulation des fichiers en mémoire secondaire. Ces interfaces réalisent des abstractions simples qui isolent le programme de plusieurs des détails de bas niveau des mécanismes des mémoires secondaires. La simplicité de ces interfaces est pertinente pour le développement d'applications nécessitant des services de base pour la persistance des données. Cependant, ces interfaces sont souvent trop limitées pour des applications complexes nécessitant des services plus sophistiqués. Les systèmes de gestion de bases de données sont prévus à cet effet. Ce chapitre présente les concepts de base pour le traitement de fichier en Python.

11.1 Lecture d'un fichier en mode texte

L'exemple suivant lit un fichier texte ligne par ligne et affiche chacune des lignes lues.

CodePython/chapitre11/ExempleLectureFichier.py

```
1. """
2. Exemple de lecture d'un fichier en mode texte
3. """
4. fichier = open("Fichier1.txt", "rt")
5. for ligne in fichier:
6.     print(ligne,end="")
7. fichier.close()
8.
```

La fonction `open()` ouvre le fichier et retourne un objet itérable qui représente le fichier ouvert.

```
fichier = open("Fichier1.txt", "rt")
```

Le premier paramètre du `open()` désigne le chemin du fichier dans la hiérarchie des répertoires du système de gestion de fichier. Dans notre exemple, `"Fichier1.txt"` est un nom de fichier relatif au répertoire courant de Python. La spécification d'un chemin de fichier à travers différents systèmes comporte de nombreuses complications qui sont prises en compte par le module `pathlib`.

Le deuxième paramètre représente le mode d'ouverture. Le mode `"rt"` désigne le mode lecture pour un fichier texte. C'est le mode de défaut. Les options possibles sont les suivantes :

Option	Description
"r"	*read* : en lecture, exception si le fichier n'existe pas
"a"	*append* : en écriture en ajout à la fin du fichier, crée le fichier si n'existe pas
"w"	*write* : en écriture, écrase le contenu existant, crée le fichier si n'existe pas
"x"	*create* : crée le fichier, exception si le fichier existe
"t"	*text* : mode texte
"b"	*binary* : mode binaire
"+"	en lecture et en écriture

Un fichier texte permet de lire et d'écrire des objets de type `str` sans avoir à se préoccuper de l'encodage employé par le système sous-jacent. Le `for` permet d'itérer ligne par ligne jusqu'à la fin du fichier :

```
for ligne in fichier:
```

Par défaut, le contenu du fichier est décodé selon le mode d'encodage des caractères du fichier. Ce mode d'encodage est déterminé par des conventions qui peuvent varier d'un système à l'autre. Il est possible de le modifier par le paramètre `encoding`. Chacune des lignes est délimitée par une `<fin de ligne>` selon le mode d'encodage du système d'exploitation. Le `str` retourné contient une représentation de la fin de ligne par `'\n'` indépendamment de la représentation réelle employée par le système.

L'exemple suivant produit le même effet avec le `with` qui a l'avantage de fermer automatiquement le fichier après l'énoncé.

CodePython/chapitre11/ExempleWith.py

```
1. """
2. Exemple du with
3. """
4. with open("Fichier1.txt", "r") as fichier:
5.     for ligne in fichier:
6.         print(ligne,end="")
```

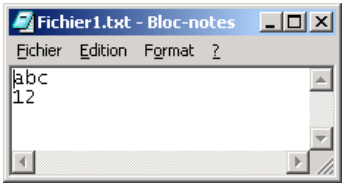
La méthode `read()` retourne un objet `str` qui contient tout le contenu du fichier jusqu'à la marque de fin de fichier.

```
"""
Exemple du read()
"""
fichier = open("Fichier1.txt", "rt")
contenu_fichier = fichier.read()
print("Taille fichier:", len(contenu_fichier))
print("Contenu:")
print(contenu_fichier)
fichier.close()
```

Exemple de fichier texte ASCII

Supposons que le fichier `Fichier1.txt` soit édité avec l'éditeur de texte *Notepad* et que le contenu soit conservé selon le mode d'encodage de défaut des caractères sous Windows, soit le code ASCII huit bits. Un exemple de contenu est affiché à la figure suivante :

CodePython/Fichier1.txt



L'éditeur de texte nous affiche le contenu du fichier sous forme de caractères. Le contenu détaillé du fichier est illustré à la figure suivante. La première ligne montre le contenu réel du fichier en binaire. La deuxième ligne indique les caractères ASCII correspondants.

```
01100001 01100010 01100011 00001101 00001010 00110001 00110010 00001101 00001010

a        b        c        \r       \n       1        2        \r       \n
```

Sous Windows, la fin de ligne est représentée par la séquence des caractères spéciaux ASCII, *retour de chariot* (`\r`) et *saut de ligne* (`\n`)[18]. Un code supplémentaire, absent dans l'exemple, est ajouté à la fin de la séquence pour indiquer la fin du fichier (0X0A sur Windows). Le résultat de l'exécution du programme serait le suivant avec notre exemple :

```
Taille fichier: 7
Contenu:
abc
```

[18] La manière de représenter les fins de ligne peut différer en fonction de la plate-forme. Unix, par exemple, emploie uniquement le *saut de ligne* (`\n`).

```
12
```

La chaîne `contenu_fichier` retournée par le `read()` est un `str` qui contient deux `\n` pour les fins de ligne. La combinaison `\r\n` d'encodage des fins de ligne est transformée en `\n` pour obtenir une représentation indépendante du système. D'un point de vue d'un fichier texte, la taille du fichier est de 7 caractères, c'est-à-dire 5 caractères (`abc` et `12`) auquel on ajoute 2 caractères pour chacune des fins de ligne.

Exemple de fichier texte UTF-16

Si le contenu du fichier était encodé selon le codage UTF-16 avec l'ordre gros-boutiste, c'est-à-dire avec l'octet le plus significatif en premier, le contenu du fichier serait le suivant :

```
11111111 11111110 00000000 01100001 00000000 01100010 00000000 01100011 00000000 00001101 00000000 00001010
UTF-16 Big Endian a               b               c               \r              \n

00000000 00110001 00000000 00110010 00000000 00001101 00000000 00001010
1               2               \r              \n
```

Les deux premiers octets représentent un indicateur d'ordre des octets (*Byte Order Mark*) du code UTF-16 avec un ordre gros-boutiste (*big endian*), ce qui signifie que le premier octet des deux octets employés pour chacun des caractères (16 bits) est l'octet d'ordre supérieur. Le tableau suivant montre quelques exemples d'indicateur d'ordre des octets.

Indicateurs d'ordre des octets

Codage	BOM hexadécimal
UTF-819	EF BB BF
UTF-16 gros boutiste (Big Endian)	FE FF
UTF-16 petit boutiste (Little Endian)	FF FE
UTF-32 gros boutiste (Big Endian)	00 00 FE FF
UTF-32 petit boutiste (Little Endian)	FF FE 00 00
ASCII	absent

En exécutant le programme de lecture en mode texte avec le fichier encodé sous UTF-16, on obtiendrait exactement le même résultat. Ainsi l'interface programmatique isole le programme des détails du mécanisme d'encodage employé par le système d'exploitation sous-jacent.

11.2 Lecture d'un fichier en mode binaire

Dans la section précédente, le contenu du fichier lu est interprété comme du texte décodé selon une convention d'encodage des caractères. Il est aussi

[19] La norme Unicode ne recommande pas d'inclure un BOM dans le cas des chaînes UTF-8.

possible de lire le contenu d'un fichier sans décodage, en mode dit *binaire*, tel qu'illustré par l'exemple suivant, qui correspond au mode `"rb"`.

CodePython/chapitre11/ExempleReadBinaire.py

```
"""
Exemple du read() binaire
"""
fichier = open("Fichier1.txt", "rb")
contenu_fichier = fichier.read()
print("Taille fichier:", len(contenu_fichier))
print("Contenu en bytes:")
print(contenu_fichier)
print("Contenu en binaire:")
for un_octet in contenu_fichier :
    print("{:0=8b}".format(un_octet))
fichier.close()

```

Le `contenu_fichier` retourné par le `read()` est un objet de type `bytes` qui est une séquence d'octets. Le `print(contenu_fichier)` affiche l'objet `bytes` avec un `b` suivie des codes ASCII des octets. Le résultat de l'exécution avec l'exemple de la section précédente codé en ASCII est :

```
Taille fichier: 9
Contenu type bytes:
b'abc\r\n12\r\n'
Contenu en binaire:
01100001
01100010
01100011
00001101
00001010
00110001
00110010
00001101
00001010
```

Le résultat du même programme avec le fichier codé en UTF-16 est :

```
Taille fichier: 20
Contenu type bytes:
b'\xfe\xff\x00a\x00b\x00c\x00\r\x00\n\x001\x002\x00\r\x00\n'
Contenu en binaire:
11111110
11111111
00000000
01100001
00000000
01100010
00000000
01100011
00000000
```

```
00001101
00000000
00001010
00000000
00110001
00000000
00110010
00000000
00001101
00000000
00001010
```

La lecture en binaire permet de lire le contenu du fichier sans décodage. Plus loin, le décodage du contenu en types de données autres que texte est présenté.

11.3 Écriture dans un fichier en mode texte

Le mode `"w"` permet d'écrire dans un fichier. Par défaut l'écriture est en mode texte. Si le fichier existe, son contenu est écrasé et remplacé par le nouveau contenu. Si le fichier n'existe pas, il est créé. L'exemple suivant écrit dans un fichier en mode texte avec le même contenu que l'exemple produit avec Notepad.

CodePython/chapitre11/ExempleWriteTexte.py

```
1. """
2. Exemple du write() fichier texte
3. """
4. fichier = open("FichierWrite.txt", "w")
5. fichier.write("abc\n12\n")
6. fichier.close()
```

L'écriture dans un fichier en mode texte a l'avantage de permettre à un humain d'en interpréter le contenu sans difficulté. Cependant c'est un mode qui est peu performant d'un point de vue de la consommation d'espace.

11.4 Écriture dans un fichier en mode binaire

Supposons que l'on veuille conserver un entier dans un fichier. Le stockage en mode texte est plus coûteux que le mode binaire. Par exemple, le programme suivant stocke l'entier 1629696561 en mode texte avec encodage ASCII.

CodePython/chapitre11/ExempleWriteText1629696561.py

```
1. """
2. Exemple du write() entier en texte
3. """
4. fichier = open("Fichier1629696561.txt", "w")
```

```
5. fichier.write("1629696561")
6. fichier.close()
```

La sortie suivante serait produite en lisant ce fichier avec le programme de lecture en binaire :

```
Taille fichier: 10
Contenu type bytes:
b'1629696561'
Contenu en binaire:
00110001
00110110
00110010
00111001
00110110
00111001
00110110
00110101
00110110
00110001
```

Il faudrait donc 10 octets pour stocker cette donnée, soit 1 octet par chiffre. On peut économiser beaucoup d'espace en stockant la donnée en mode binaire tel qu'illustré par l'exemple suivant. Le mode `"wb"` permet d'écrire en mode binaire.

CodePython/chapitre11/ExempleWriteBin1629696561.py

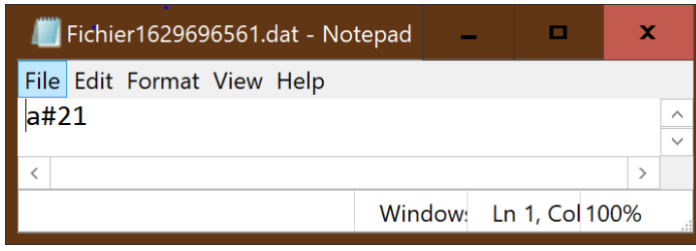
```
1. """
2. Exemple du write() entier en binaire
3. """
4. fichier = open("Fichier1629696561.dat", "wb")
5. i = 1629696561
6. fichier.write(i.to_bytes(4, byteorder='big', signed=True))
7. fichier.close()
```

Le contenu du fichier a une taille de 4 octets plutôt que 10 comme c'était le cas dans la version texte. Ceci permet de représenter n'importe quel entier de 32 bits.

```
Taille fichier: 4
Contenu type bytes:
b'a#21'
Contenu en binaire:
01100001
00100011
00110010
00110001
```

Le stockage en binaire produit habituellement une représentation plus compacte qu'en texte. Par ailleurs, si le fichier `Fichier1629696561.dat` est ouvert avec l'éditeur de texte *Notepad* de *Windows*, le contenu semble

incompréhensible parce que le programme *Notepad* interprète le contenu du fichier comme une suite de caractères ASCII et non un entier de 4 octets :



Pour récupérer le contenu du fichier sous une forme appropriée, il faut décoder le contenu en un entier de 4 octets.

```
1. """
2. Exemple du read() entier en binaire
3. """
4. fichier = open("Fichier1629696561.dat", "rb")
5. octets = fichier.read(4)
6. i = int.from_bytes(octets, byteorder='big', signed=True)
7. print(i)
8. fichier.close()
```

Le stockage en binaire nécessite de bien définir la disposition des données.

11.5 Lecture d'un fichier texte en format csv

Il est souvent nécessaire de lire des données provenant d'un fichier en format `csv` (*comma-separated values*) qui est fréquemment employé pour échanger des données. C'est un format texte qui sert à représenter des données tabulaires organisées en lignes et en colonnes, typiques des chiffriers et des bases de données. Chacune des lignes du fichier représente une ligne d'un tableau. La première ligne est optionnellement employée pour indiquer les noms des colonnes, séparés habituellement par des virgules. La virgule agit comme *délimiteur*. Chacune des lignes contient une valeur pour chacune des colonnes du tableau tel qu'illustré par l'exemple suivant. Le fichier contient des données d'un catalogue de plants à vendre d'une pépinière. Il y a trois colonnes, le numéro de catalogue du plant, sa description et son prix unitaire. Les guillemets peuvent encadrer les chaînes de caractères pour permettre d'inclure le caractère délimiteur mais ils sont optionnels dans le cas contraire.

```
10,Cèdre en boule,10.99
20,Sapin,12.99
40,Epinette bleue,25.99
50,Chêne,22.99
```

```
60,Erable argenté,15.99
70,Herbe à puce,10.99
80,Poirier,26.99
81,Catalpa,25.99
90,Pommier,25.99
95,Génévrier,15.99
```

Le module `csv` permet de lire un fichier `csv` sous forme d'un itérateur et de retourner chacune des lignes comme une liste de chaînes de caractères (`str`), une par colonne :

CodePython/chapitre11/ExempleReadCSV.py

```
1. """
2. Lecture du fichier Plants.csv en format csv
3. """
4. import csv
5. with open('Plants.csv', mode='r') as fichier_csv:
6.     lecteur_csv = csv.reader(fichier_csv)
7.     for ligne in lecteur_csv:
8.         print(ligne)
```

Le résultat affiché est :

```
['10', 'Cèdre en boule', '10.99']
['20', 'Sapin', '12.99']
['40', 'Epinette bleue', '25.99']
['50', 'Chêne', '22.99']
['60', 'Erable argenté', '15.99']
['70', 'Herbe à puce', '10.99']
['80', 'Poirier', '26.99']
['81', 'Catalpa', '25.99']
['90', 'Pommier', '25.99']
['95', 'Génévrier', '15.99']
```

11.6 Traitement d'enregistrements dans un fichier binaire

Pour écrire et lire plusieurs données dans un fichier en mode binaire, il faut planifier la manière de placer la séquence des octets pour chacune des données. Supposons que l'on veuille stocker les données du catalogue de plants de l'exemple précédent sous forme de fichier binaire. Une manière de procéder consiste à stocker chacun des plants sous forme d'un enregistrement composé d'une suite d'octets de taille fixe. Les plants sont disposés les uns à la suite des autres. Pour chacun des enregistrements, il faut convertir les données d'un plant sous forme d'une suite d'octets.

Une manière systématique d'opérer la conversion est d'employer le package `struct`. Ce package permet de convertir un ensemble de données en une suite d'octets en reprenant les conventions du langage C. La fonction

`struct.pack()` permet de convertir un ensemble de données de types divers en une séquence d'octets en spécifiant un format de conversion qui détermine la manière de convertir chacune des données. L'exemple suivant convertit un entier, une chaîne de caractères et un réel en une séquence d'octets et inverse la conversion avec `struct.unpack()`.

```
>>> import struct
>>> octets = struct.pack('i5sf',10,b'abcde',3.4)
>>> struct.unpack('i5sf',octets)
(10, b'abcde', 3.4000000953674316)
```

Le masque `i5sf` signifie que la première donnée est un entier (format `i`), suivie d'une chaîne de 5 octets (format `5s`), suivie d'un nombre en point flottant (code `f`). Chacun des formats détermine comment la conversion est opérée. Le format `i` spécifie que l'entier est converti en une suite de 4 octets, le format `5s` spécifie que la chaîne d'octets est convertie en une suite de 5 octets et le format `f`, que le nombre en point flottant est converti en une suite de 4 octets. Donc, au total, on obtient une suite de 13 octets. Pour plus de détails au sujet des formats et de leurs significations, voir la documentation de `struct`.

L'exemple suivant lit le fichier `Plants.csv` et stocke les données converties en binaire dans le fichier `Plants.dat`. Chacun des plants est représenté par un enregistrement de 28 octets, selon le format de conversion `i20sf` passé à la fonction `struct.pack()`.

CodePython/chapitre11/ExempleWriteFichierPlantsBinaire.py

```
 1. """
 2. Lecture du fichier Plants.csv en format csv et stockage en un
fichier binaire
 3. Conversion des données avec struct
 4. """
 5.
 6. import csv
 7. import struct
 8.
 9. with open('Plants.csv', mode='r') as fichier_csv:
10.     with open('Plants.dat',mode = 'wb') as fichier_binaire:
11.         lecteur_csv = csv.reader(fichier_csv)
12.         for ligne in lecteur_csv:
13.
fichier_binaire.write(struct.pack('i20sf',int(ligne[0]),str.encode(
ligne[1]),float(ligne[2])))
14.
```

L'exemple suivant parcoure les enregistrements du fichier binaire `Plants.dat` un par un et en affiche le contenu.

CodePython/chapitre11/ExempleReadFichierPlantsBinaire.py

```
"""
Lecture du fichier binaire Plants.dat encodé avec struct
"""

import csv
import struct

with open('Plants.dat', mode='rb') as fichier_binaire:
    while True:
        r=fichier_binaire.read(28)
        if not r:
            break
        print(struct.unpack('i20sf',r))

```

Dans certaines applications, il peut être utile de lire ou d'écrire un enregistrement particulier sans être obligé de parcourir tout le fichier. Ceci devient important au fur et à mesure que le taille du fichier augmente. À cet effet, la méthode `seek()` permet de sauter à un octet particulier dans un fichier. Comme tous les enregistrements occupe le même espace, il est possible de calculer la position d'un enregistrement. Par exemple, pour se positionner au 3ième enregistrement, il faut faire un `seek(56)` parce que chacun des enregistrements occupe 28 octets. L'exemple suivant effectue une lecture de cet enregistrement.

CodePython/chapitre11/ExempleSeekFichierBinaire.py

```
"""
Lecture en accès direct avec seek() dans le fichier binaire
Plants.dat encodé avec struct
"""
import csv
import struct
with open('Plants.dat', mode='rb') as fichier_binaire:
    fichier_binaire.seek(56)
    enregistrement=fichier_binaire.read(28)
    print(enregistrement)
    print(struct.unpack('i20sf',enregistrement))

```

Il est aussi possible d'écrire un enregistrement pour le modifier. S'il faut permettre de gérer diverses modifications, telles que l'insertion, la suppression, la modification de données, de manière sécuritaire et partagée, ceci peut devenir complexe à programmer. Pour des applications qui exigent une gestion de données volumineuses de manière performante, flexible et sécuritaire, il est avantageux de recourir aux services d'un système

de gestion de bases de données. Ces systèmes emploient habituellement une représentation binaire afin de minimiser l'espace consommé.

11.7 Stockage d'objets avec pickle

Le stockage des données en binaire vu dans l'exemple précédent est plutôt fastidieux et sujet à erreur. Une manière plus simple de stocker des données de manière persistante dans un fichier est d'employer le module `pickle`. Il permet de stocker des objets et de les récupérer d'un fichier sans avoir à se préoccuper de la disposition exacte des octets dans le fichier. À cet effet, un objet est converti en une série d'octets qui contient l'information nécessaire (*métadonnées*) pour le restituer sous sa forme originale. La *sérialisation* (*pickling*) est le processus de conversion d'un objet et des objets auxquels il fait référence sous forme d'une suite d'octets. Différents protocoles sont disponibles à cet effet. La *désérialisation* est le processus inverse. Le stockage est un peu moins compact qu'une représentation binaire parce qu'il inclut les métadonnées.

L'exemple de programme suivant lit le fichier `Plants.csv`, convertit les données en une liste d'objets de la classe `Plant`. Par la suite, la liste des objets est stockée dans le fichier `Plants.pickle` à l'aide de la fonction `pickle.dump()`.

CodePython/chapitre11/ExemplePickle.py

```
"""
Lecture du fichier Plants.csv en format csv, création d'une liste d'objets de la classe Plant
Ecriture de l'objet liste_plants dans Plants.pickle avec le module pickle
"""

import csv
import struct
import pickle

# Classe Plant
class Plant:
    def __init__(self,numero,description,prix):
        self.numero = numero
        self.description = description
        self.prix = prix
    def __str__(self):
        return '|{:>10}|{:>20}|{:8.2f}|'.format(self.numero,self.description,self.prix)

liste_plants = []

with open('Plants.csv', mode='r') as fichier_csv:
    lecteur_csv = csv.reader(fichier_csv)
    for ligne in lecteur_csv:
        un_plant = Plant(int(ligne[0]),ligne[1],float(ligne[2]))
        print(un_plant)
        liste_plants.append(un_plant)

with open('Plants.pickle', mode = 'wb') as fichier_binaire:
    pickle.dump(liste_plants, fichier_binaire, pickle.HIGHEST_PROTOCOL)
```

L'exemple suivant lit le fichier `Plants.pickle` et convertit son contenu sous forme de la liste des objets de la classe `Plant`.

CodePython/chapitre11/ExemplePickleLoad.py

```
"""
Lecture du fichier Plants.pickle avec pickle.load()
"""

import csv
import struct
import pickle

# Classe Plant
class Plant:
    def __init__(self,numero,description,prix):
        self.numero = numero
        self.description = description
        self.prix = prix
    def __str__(self):
        return '|{:>10}|{:>20}|{:8.2f}|'.format(self.numero,self.description,self.prix)

with open('Plants.pickle', mode = 'rb') as fichier_binaire:
    liste_plants = pickle.load(fichier_binaire)

for un_plant in liste_plants:
    print(un_plant)
```

11.8 Stockage d'objets sous le format JSON

Un format de données textuelles souvent employé pour l'échange de données est le format `JSON` (*Java Script Object Notation*). Il a l'avantage d'être facilement interprétable par l'humain et plus élaboré que le format `csv`. Il permet en particulier d'imbriquer les structures de manière hiérarchique et d'avoir des parties variables. En revanche, il est moins compact que le format *csv*. Voir section 13.1.5 pour plus de détails sur ce format.

Le programme suivant illustre la création d'un fichier dans le format `JSON`.

CodePython/chapitre11/ExempleWriteJson.py

```
"""
Lecture du fichier Plants.csv en format csv, création d'une liste d'objets de la classe
Plant
Ecriture des données de liste_plants dans Plants.json avec json.dump()
"""

import csv
import struct
import json

# Classe Plant
class Plant:
    def __init__(self,numero,description,prix):
        self.numero = numero
        self.description = description
        self.prix = prix
    def __str__(self):
        return '|{:>10}|{:>20}|{:8.2f}|'.format(self.numero,self.description,self.prix)

liste_plants = []

with open('Plants.csv', mode='r') as fichier_csv:
```

```
    lecteur_csv = csv.reader(fichier_csv)
    for ligne in lecteur_csv:
        un_plant = Plant(ligne[0],ligne[1],ligne[2])
#        print(un_plant)
        liste_plants.append(ligne)

with open('Plants.json', mode = 'w') as fichier_json:
    json.dump(liste_plants, fichier_json)

```

Le programme suivant lit le fichier `JSON` créé par l'exemple précédent.

CodePython/chapitre11/ExempleReadJson.py

```
"""
Lecture du fichier Plants.json avec json.load()
"""

import json

# Classe Plant
class Plant:
    def __init__(self,numero,description,prix):
        self.numero = numero
        self.description = description
        self.prix = prix
    def __str__(self):
        return '|{:>10}|{:>20}|{:8.2f}|'.format(self.numero,self.description,self.prix)

with open('Plants.json', mode = 'r') as fichier_json:
    liste_plants = json.load(fichier_json)

for un_plant in liste_plants:
    print(un_plant)

```

11.9 Gestion de répertoire avec os

Le module `os` contient diverses opérations système dont des opérations de gestion de répertoire de fichier. Quelques fonctions de base sont illustrées par l'exemple suivant qui crée un nouveau répertoire dans le répertoire de travail et ajoute un fichier.

CodePython/chapitre11/ExempleOsRepertoire.py

```
"""
Exemples de base de manipulation de répertoire de fichier avec le module os
"""

import os

repertoire_de_travail = os.getcwd() # Chercher le répertoire de travail
print("Répertoire de travail :",repertoire_de_travail)

# Création d'un chemin par concaténation avec os.path.join
sous_repertoire = os.path.join(repertoire_de_travail, "repertoire_test")

if not os.path.exists(sous_repertoire):
    print("Création du sous-répertoire 'repertoire_test' dans le répertoire de travail")
    os.mkdir(sous_repertoire)

print("Changement du répertoire de travail")
os.chdir(sous_repertoire) # Changer le répertoire de travail

nouveau_repertoire_de_travail = os.getcwd() # Chercher le nouveau répertoire de travail
print("Nouveau répertoire de travail :",nouveau_repertoire_de_travail)

```

```
23. print("Création du fichier 'FichierTest.txt' dans le répertoire de travail")
24. fichier = open("FichierTest.txt", "w")
25. fichier.write("abc\n12\n")
26. fichier.close()
27.
28. print("Contenu du répertoire de travail:")
29. print(os.listdir(nouveau_repertoire_de_travail))
```

Résultat :

```
Répertoire de travail : C:\Users\Robert\CodePython
Changement du répertoire de travail
Nouveau répertoire de travail : C:\Users\Robert\CodePython\repertoire_test
Création du fichier 'FichierTest.txt' dans le répertoire de travail
Contenu du répertoire de travail:
['FichierTest.txt']
```

La fonction `os.getcwd()` retourne le chemin du répertoire de travail :

```
repertoire_de_travail = os.getcwd() # Chercher le répertoire de travail
```

Avec Windows, les sous-répertoires du chemin exprimé sous forme textuelle sont séparés par un \ par opposition à Unix et IOS qui utilisent plutôt le /. Cette incompatibilité est source d'erreurs et de complications qui sont traitées par la module `os.path`.

La fonction `os.path.join()` crée un nouveau chemin en traitant cette incompatibilité par l'utilisation du séparateur approprié au système sous-jacent :

```
sous_repertoire = os.path.join(repertoire_de_travail, "repertoire_test")
```

La fonction `os.path.exists()` vérifie l'existence d'un chemin et la fonction `os.mkdir()` crée un nouveau chemin :

```
if not os.path.exists(sous_repertoire):
    print("Création du sous-répertoire 'repertoire_test' dans le répertoire
de travail")
    os.mkdir(sous_repertoire)
```

La fonction `os.listdir()` affiche le contenu du répertoire passé en paramètre :

```
print(os.listdir(nouveau_repertoire_de_travail))
```

Si le répertoire n'est pas spécifié, c'est le contenu du répertoire de travail qui est produit. Il est possible de spécifier un chemin selon la syntaxe Windows tel qu'illustré par l'exemple suivant :

CodePython/chapitre11/ExempleCheminWindows.py

```
1. """
2. Exemple de chemin selon la syntaxe Windows
3. """
```

```
 4.
 5. import os
 6.
 7. os.chdir("C:\\Users\\Robert\\CodePython\\data") # Changer le répertoire de travail
 8.
 9. nouveau_repertoire_de_travail = os.getcwd() # Chercher le nouveau répertoire de travail
10. print("Nouveau répertoire de travail :",nouveau_repertoire_de_travail)
11.
12. print("Contenu du répertoire de travail:")
13. print(os.listdir(nouveau_repertoire_de_travail))
14.
```

Résultat :

```
Nouveau répertoire de travail : C:\Users\Robert\CodePython\data
Contenu du répertoire de travail:
['FashionMNIST', 'MNIST']
```

Il est aussi possible d'employer la syntaxe Unix avec le système Windows :

CodePython/chapitre11/ExempleCheminUnix.py

```
 1. """
 2. Exemple de chemin selon la syntaxe Unix
 3. """
 4.
 5. import os
 6.
 7. os.chdir("/Users/Robert/CodePython/data") # Changer le répertoire de travail
 8.
 9. nouveau_repertoire_de_travail = os.getcwd() # Chercher le nouveau répertoire de travail
10. print("Nouveau répertoire de travail :",nouveau_repertoire_de_travail)
11.
12. print("Contenu du répertoire de travail:")
13. print(os.listdir(nouveau_repertoire_de_travail))
```

Ceci fonctionne correctement avec Windows. Cependant la compatibilité inverse de la syntaxe Windows vers Unix n'est pas supportée.

Le module `pathlib` permet de produire du code portable par l'utilisation de la classe `Path` pour représenter un chemin de manière indépendante du système d'exploitation.

CodePython/chapitre11/ExemplePathlib.py

```
 1. """
 2. Exemple de chemin avec Pathlib compatible Windows ou Unix
 3. """
 4.
 5. from pathlib import Path
 6. import os
 7.
 8. repertoire = Path("/Users/Robert/CodePython/data")
 9.
10. os.chdir(repertoire) # Changer le répertoire de travail
11.
12. nouveau_repertoire_de_travail = os.getcwd() # Chercher le nouveau répertoire de travail
13. print("Nouveau répertoire de travail :",nouveau_repertoire_de_travail)
14.
15. print("Contenu du répertoire de travail:")
16. print(os.listdir(nouveau_repertoire_de_travail))
```

Le chemin est avec un objet de la classe `Path` en employant la syntaxe avec `/`. La conversion dans la syntaxe appropriée est effectuée automatiquement, ce qui produit du code portable peu importe le système.

12 Structures de données, algorithmes et complexité

Ce chapitre introduit les notions de base des structures de données, des algorithmes et de leur complexité. Ces sujets sont très vastes et des ouvrages entiers y sont consacrés. Cette section vise à introduire les concepts de base et les enjeux sous-jacents. Précédemment, différents types ont été présentés pour la manipulation des collections (`str`, `list`, `tuple`, `set`, `dict`). D'autres types prédéfinis sont aussi disponibles à cet effet. Chacun de ces types possède une structure de données particulière adaptée aux traitements envisagés. Ce chapitre prend le problème de recherche dans une collection comme exemple pour illustrer les concepts.

Un algorithme est une recette pour résoudre un problème. Cette notion est surtout employée dans le cas de la recherche d'une solution à un problème non trivial. Il peut y avoir plusieurs solutions au problème, plusieurs algorithmes. Par exemple, pour rechercher un élément dans une liste, il peut y avoir plusieurs algorithmes possibles. L'algorithmique désigne l'étude des algorithmes et la recherche des meilleurs algorithmes. Les algorithmes sont intimement liés aux structures de données visées. Le meilleur algorithme pour la recherche dans une liste est différent du meilleur algorithme de recherche dans une liste triée.

Pour caractériser la performance des algorithmes, différentes mesures sont employées, tel que la mesure du temps nécessaire et de la mémoire consommée. La notation grand *O* (*big O*) est souvent employée à cet effet et sera introduite dans ce chapitre.

12.1 Recherche linéaire dans une liste Python

Pour illustrer les concepts de base, prenons un problème simple mais pour lequel des algorithmes bien connus ont été développés, la recherche d'un élément dans une collection. Supposons dans un premier temps que la structure des données utilisée pour représenter la collection est une liste Python. La recherche d'un élément est déjà implémentée en Python avec l'opérateur `in` (voir 4.4.3) entre autres. Un algorithme possible pour ce problème est la recherche linéaire illustrée par le code suivant :

CodePython/chapitre12/SimulationComplexiteRechercheLineaire.py

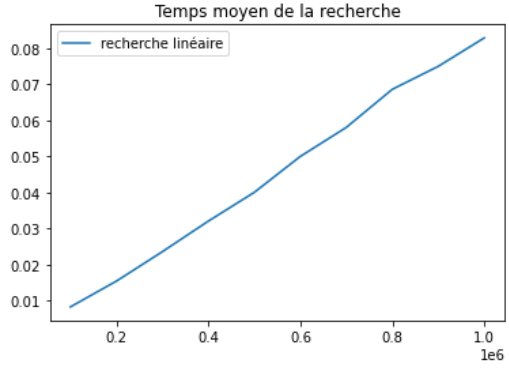
```
 1. def recherche_lineaire(element, liste):
 2.     """ Recherche linéaire.
 3.         Retourne vrai si element est dans la liste, faux sinon
 4.     """
 5.     for indice in range(len(liste)):
 6.         if liste[indice] == element:
 7.             return True
 8.     return False
 9.
10. liste = [ 7,5,10,1,25,8,20 ]
11. element = 8
12. print("------------- Recherche linéaire de ",element," dans ", liste)
13. if recherche_lineaire(element, liste):
14.     print("Element présent")
15. else:
16.     print("Element absent")
```

Résultat:

```
------------- Recherche linéaire de  8  dans  [7, 5, 10, 1, 25, 8, 20]
Element présent
```

Chacun des éléments de la liste est vérifié un par un avec un `for` jusqu'à ce que l'élément soit trouvé ou jusqu'à ce que la liste soit épuisée. Pour analyser la performance d'un algorithme, on peut mesurer le temps nécessaire dans une implémentation.

La figure suivante a été produite par simulation en variant *n*, la taille de la liste, de 100 000 à 1 000 000 par pas de 100 000. Les éléments de la liste sont générés aléatoirement selon une distribution uniforme dans l'intervalle [0,10*n*-1]. Pour chacune des valeurs de *n*, 100 échantillons différents d'éléments à rechercher sont générés et le temps de traitement est mesuré à partir de l'horloge système. On observe que le temps de traitement moyen croît linéairement avec *n*.



Quoique ce genre d'analyse soit utile, il dépend de l'implémentation particulière choisie. Il est intéressant d'analyser un algorithme en faisant abstraction de l'implémentation. On distingue souvent le meilleur cas, le cas moyen et le pire cas. Pour chacune des possibilités, le nombre d'opérations nécessaires est compté en fonction de la taille du problème, ici *n*. Il peut être compliqué de chercher à mesurer le temps de chacune des opérations mais il y a souvent un sous-ensemble plus critique qui peut être étudié. Dans le cas de la recherche linéaire, le nombre de comparaisons *comp*(*n*) avec l'élément recherché est une mesure fondamentale. Supposons que l'élément n'est pas présent. Le nombre maximum de comparaisons en meilleur cas, en moyenne et en pire cas est *n* car il faut vérifier tous les éléments de la liste. Si l'élément est présent, le meilleur cas est 1, le pire cas est encore *n*, la moyenne est $n/2$. Le meilleur cas est rare et il se produit lorsque l'élément est en première position.

comp(n)	Meilleur cas	Cas moyen	Pire cas
Élément absent	*n*	*n*	*n*
Élément présent	1	$n/2$	*n*

Pour caractériser la performance d'un algorithme, la notation avec un grand *O* (*big O*) est souvent privilégiée. Elle mesure la croissance du temps de traitement lorsque la taille des données devient très grande en mettant l'accent sur le facteur dominant. Formellement,

$f(n) = O(g(n))$si il existe deux constantes c et m tel que

$|f(n)| \leq c\ |g(n)|$

La croissance est mesurée à une constante près parce que le facteur constant est souvent peu significatif d'un point de vue pratique. Si *n* est la taille de la liste, la recherche linéaire en moyenne et en pire cas est $O(n)$, ce qui signifie que le temps de traitement croît linéairement. Pour le meilleur cas, la recherche est $O(n)$ si l'élément est absent et $O(1)$ si l'élément est présent.

comp(n)	Meilleur cas	Cas moyen	Pire cas
Élément absent	$O(n)$	$O(n)$	$O(n)$
Élément présent	*O(1)*	$O(n)$	$O(n)$

Pour améliorer la performance de la recherche linéaire, il est utile d'employer une sentinelle, qui consiste à ajouter l'élément à rechercher à la fin de la liste. Ainsi, il est certain que l'élément recherché soit trouvé. Ceci

permet d'éviter le test de dépassement de l'indice par rapport à la taille de la liste.

Exercice. Codez la recherche linéaire avec sentinelle.

12.2 Recherche dans une liste triée

Un cas particulièrement intéressant de la recherche est celui d'une liste triée. Il est possible d'améliorer la recherche linéaire dans le cas d'une liste triée en arrêtant la recherche aussitôt que l'élément recherché devient plus petit que l'élément en cours de traitement, tel que le montre le code suivant :

CodePython/chapitre12/ExempleRechercheDansListe.py

```
 1. def recherche_lineaire_triee(element, liste_triee):
 2.     """ Recherche linéaire améloirée. Suppose que la liste est
triée.
 3.         Retourne vrai si element est dans la liste, faux sinon
 4.     """
 5.     for indice in range(len(liste_triee)):
 6.         if liste_triee[indice] == element:
 7.             return True
 8.         if liste_triee[indice] > element:
 9.             return False
10.     return False
```

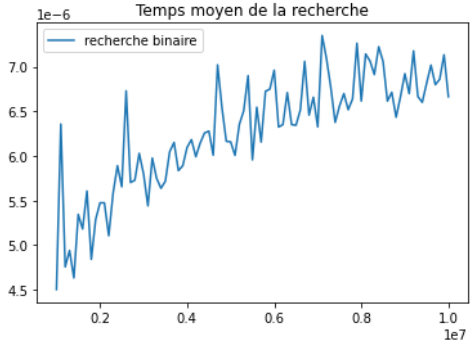
C'est algorithme est un peu amélioré mais la complexité moyenne et en pire cas demeure $O(n)$. Une solution qui apporte une amélioration spectaculaire est l'algorithme de recherche binaire. Plutôt que de parcourir les éléments un après l'autre, le principe est de découper la liste en deux en comparant l'élément recherché avec celui du milieu. S'il est plus petit, la recherche est poursuivie en éliminant la partie supérieure de la liste, ce qui élimine d'un coup la moitié des éléments. Et s'il est plus grand, la partie inférieure est éliminée. S'il est égal, la recherche est terminée. Ce processus est poursuivi jusqu'à ce que l'élément soit trouvé ou qu'il ne reste plus d'élément dans la partie inférieure ou supérieure choisie. Le code suivant implémente la recherche binaire.

```
 1. def recherche_binaire(element, liste_triee):
 2.     """ Recherche binaire. Suppose que la liste est triée.
 3.         Retourne vrai si element est dans la liste, faux sinon
 4.     """
 5.     borne_inferieure = 0
 6.     borne_superieure = len(liste_triee) - 1
 7.     milieu = 0
 8.
 9.     while borne_inferieure <= borne_superieure:
10.         milieu = (borne_superieure + borne_inferieure) // 2
```

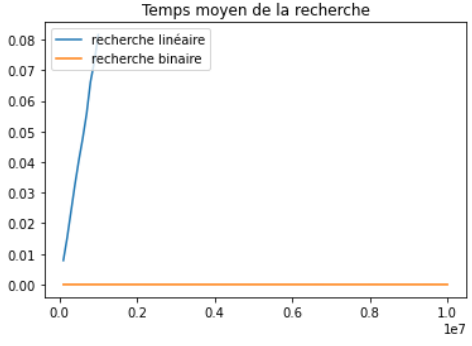
```
11.         if liste_triee[milieu] < element:
12.             borne_inferieure = milieu + 1
13.         elif liste_triee[milieu] > element:
14.             borne_superieure = milieu - 1
15.         else: # liste_triee[milieu] == element
16.             return True
17.     return False
```

À chacune des itérations, la taille de l'intervalle des éléments à rechercher est divisée par 2, ce qui conduit à un nombre d'itérations borné par $\log_2(n)$ et une complexité $O(\log_2(n))$.

La croissance logarithmique est illustrée par la figure suivante générée par simulation en prenant 100 échantillons différents d'éléments à rechercher pour chacune des valeurs de n de 1 000 000 à 10 000 000 par pas de 1 000 000.



La figure comparative suivante montre l'amélioration spectaculaire du temps de traitement de la recherche binaire par rapport à la recherche linéaire dans le cadre de la simulation.



12.3 Formulation récursive de la recherche binaire

Une fonction est dite récursive lorsqu'elle s'appelle elle-même. Plusieurs fonctions peuvent naturellement être formulée récursivement en décomposant un problème en un sous-problème de même nature mais de taille inférieure dans ses paramètres. La recherche binaire peut être formulée de manière récursive de manière très naturelle. Dans la formulation précédente de la recherche binaire, la recherche dans une liste d'éléments vérifie dans quelle sous-liste devrait se trouver l'élément par comparaison avec l'élément milieu et la recherche se poursuit dans la sous-liste. Cette recherche peut être exprimée récursivement en ajoutant deux paramètres à la fonction de recherche, la borne inférieure et la borne supérieure de la sous-liste à rechercher. La recherche dans la sous-liste est ainsi exprimée par la même fonction en spécifiant les nouvelles valeurs des bornes inférieures et supérieures.

```
 1. def recherche_binaire_recursive(element,liste_triee, borne_inferieure, borne_superieure):
 2.     """ Recherche binaire récursive. Suppose que la liste est triée.
 3.         Retourne vrai si element est dans la liste, faux sinon
 4.     """
 5.
 6.     if borne_superieure >= borne_inferieure:
 7.         milieu = (borne_superieure + borne_inferieure) // 2
 8.         if liste_triee[milieu] == element:
 9.             return True
10.         elif liste_triee[milieu] > element:
11.             return recherche_binaire_recursive(liste_triee, borne_inferieure, milieu - 1,
element)
12.         else:
13.             return recherche_binaire_recursive(liste_triee, milieu + 1, borne_superieure,
element)
14.     else:
15.         # L'élément est absent
16.         return False
17.
18. liste = [ 7,5,10,1,25,8,20 ]
19. liste.sort()
20. element = 8
21. print("------------- Recherche binaire récursive de ",element," dans liste triée", liste)
22. if recherche_binaire_recursive(element, liste,0,len(liste)):
23.     print("Element présent")
24. else:
25.     print("Element absent")
26.
```

Exercice. Codez une fonction `factoriel(n : int)` qui effectue le calcul suivant de manière récursive. N! = 1*2* … *N = N*(N-1)!

Si la recherche dans une liste est un traitement critique d'une application, il peut être judicieux de maintenir la liste triée. Cependant, ce maintien a un coût important pour la mise-à-jour (ajout, suppression) de la collection des éléments de la liste. D'autres structures de données permettent une mise-à-jour plus performante tout en maintenant un temps de recherche efficace. Des structures arborescentes permettent aussi un temps de recherche $O(\log_2(n))$ tout en permettant un temps de mise-à-jour $O(\log_2(n))$. Aussi, le

hachage permet une recherche $O(1)$ en moyenne, c'est-à-dire un temps constant peu importe la taille de la collection. De plus, la mise-à-jour est aussi $O(1)$, en moyenne, ce qui en fait une structure de données très populaire pour la recherche dans de grands volumes de données.

Cette étude brève du problème de recherche dans une liste montre l'importance des structures de données et algorithmes. La mesure grand O permet d'évaluer le potentiel d'un algorithme dans le cas de son applicabilité sur de grands volumes de données.

Les classes de complexité $O(n)$ et $O(\log_2(n))$ sont fréquemment rencontrées en pratique. La classe $O(1)$ est le cas d'un temps constant indépendant de la taille des données à traiter. Pour la recherche dans une collection, le hachage permet d'approcher une complexité $O(1)$, ce qui en fait une structure de données très attrayante pour la recherche dans de grands volumes de données.

13 Développement d'applications Web

Une application Web est une application à laquelle on accède généralement par l'entremise d'un navigateur Web (par ex., Safari, Chrome, Firefox, etc.). L'application tourne généralement sur un serveur Web. Il est commun d'héberger les services Web chez des fournisseurs d'infonuagiques tels que Amazon (AWS) ou Microsoft (Azure). Les entreprises disposent parfois de leurs propres serveurs Web, parfois hébergés au sein de salles de serveurs.

Nous sommes tous familiers avec les applications Web, celles-ci étant omniprésentes : par ex., les sites de commerce électronique (Amazon), les sites Web gouvernementaux, etc. comportent une ou plusieurs applications Web.

Il est aussi possible de produire des applications Web avec lesquelles les êtres humains n'interagissent pas directement : on parle alors souvent de services Web. Par exemple, une application mobile sur votre téléphone pourrait communiquer avec une application Web sans que vous en preniez pleinement conscience. C'est le cas notamment de l'application mobile Facebook. Les jeux vidéo en ligne communique aussi parfois avec des applications Web dédiées.

13.1 Notions Web : HTTP, HTML, JSON, etc.

Une application Web est un programme qui reçoit des requêtes par l'entremise du réseau en utilisant le plus souvent le protocole HTTP, ou la contrepartie sécurisée, HTTPS. C'est toujours le client (par ex., un navigateur Web) qui initie la communication en faisant une *requête* HTTP.

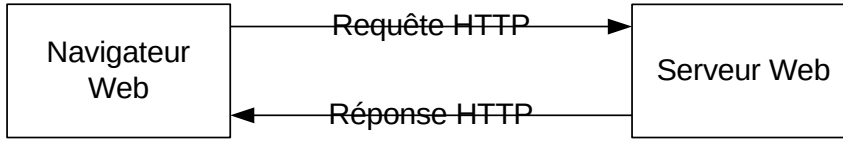


Figure 15. Mécanisme de base du protocole HTTP.

L'application Web répond à la requête de différentes façons. Le plus souvent, la réponse prendra la forme d'un document HTML ou JSON, même si plusieurs autres possibilités existent (images, XML, etc.). La réponse débute par un code de résultat qui peut correspondre à un code d'erreur, par exemple, 404 indique que la ressource n'est pas disponible.

Ensuite, un entête identifie le type de réponse qui suit. Par exemple, Content-type: text/html indique que le contenu qui suit est sous format HTML. D'autres informations sont aussi incluses dans l'entête tel que la version du protocole, la date, l'identification du serveur, la date de dernière modification, la taille du fichier, l'encodage des caractères, etc.

13.1.1 HTTP/HTTPS

Le protocole HTTP utilise un protocole de communication réseau de plus bas niveau pour transmettre les informations par le réseau, tel que TCP (*Transmission Control Protocol*) et UDP (*User Datagram Protocol*). TCP est un protocole de transport orienté connexion, ce qui signifie qu'il établit une connexion entre deux hôtes avant de transférer des données. UDP, en revanche, est un protocole de transport sans connexion, ce qui signifie qu'il ne nécessite pas d'établir une connexion avant de transférer des données. Certains protocoles de la couche application peuvent utiliser UDP comme protocole de transport sous-jacent. Par exemple, le protocole DNS (Domain Name System) utilise UDP pour transférer des requêtes et des réponses entre les clients DNS et les serveurs DNS.

TCP et UDP sont deux protocoles de communication réseau qui ont des avantages et des inconvénients. Avec TCP, une connexion est initiée et les données sont échangées en séquence. Le protocole UDP permet la transmission des données dans le désordre avec perte des données occasionnellement. TCP est parfois plus lent et utilise plus de bande passante que UDP. Plus le réseau est encombré et peu fiable, plus TCP perd en vitesse.

HTTP (*Hypertext Transfer Protocol*) est un protocole de la couche application qui est utilisé pour transférer des données sur le Web. HTTP utilise TCP comme protocole de transport sous-jacent pour transférer des données. Cela signifie que lorsque vous accédez à un site Web via HTTP, votre navigateur établit une connexion TCP avec le serveur Web distant et utilise cette connexion pour transférer les données HTTP. HTTP est un protocole non sécurisé qui transfère les données en texte clair. Cela signifie que toute personne qui intercepte les données peut les lire. HTTPS (*Hypertext Transfer Protocol Secure*), en revanche, utilise SSL (*Secure Sockets Layer*) ou TLS (*Transport Layer Security*) pour chiffrer les données. Cela signifie que les données sont sécurisées et ne peuvent pas être lues par des tiers.

Le protocole visé est spécifié par l'URL. Un URL (*Uniform Resource Locator*) est une chaîne de caractères qui identifie de manière unique une ressource sur Internet. Il est utilisé pour localiser des pages Web, des images, des vidéos, des fichiers, etc. sur le Web. Un URL est composé de plusieurs

parties, notamment le protocole, le nom d'hôte, le chemin et la chaîne de requête.

Le protocole est la méthode utilisée pour accéder à la ressource, par exemple HTTP ou HTTPS. Le nom d'hôte est le nom du serveur qui héberge la ressource. Le chemin est l'emplacement de la ressource sur le serveur. La chaîne de requête est utilisée pour transmettre des paramètres supplémentaires à la ressource.

Par exemple, l'URL suivant pointe vers la page d'accueil de google :

```
https://www.google.com/
```

Dans cet exemple, HTTPS est le protocole utilisé pour accéder à la ressource, `www.google.com` est le nom d'hôte et `/` est le chemin. Les URL HTTPS commencent par `https://` tandis que les URL HTTP commencent par `http://`.

Les requêtes HTTP les plus simples sont de type GET. Une requête GET comporte un hyperlien, et la réponse peut être, par exemple, un document HTML. Bien que ça ne soit pas obligatoire, le navigateur peut supposer que la même requête GET devrait toujours donner la même réponse : les navigateurs peuvent utiliser ce fait pour mettre le résultat de la requête en mémoire, et éviter de répéter la requête. L'autre type de requête communément utilisé est la requête POST qui est normalement suivi par la transmission d'une information détaillée. Il existe plusieurs autres types de requêtes HTTP : PUT, DELETE, TRACE, CONNECT, etc.

Dans la pratique, il n'est pas nécessaire de connaître les détails du protocole HTTP pour créer une application Web. Le protocole HTTPS est une variante sécurisée : pour les fins du développement d'une application Web, il peut être plus simple de toujours exiger le protocole HTTPS si le site comporte des informations sensibles (par ex., l'échange d'un mot de passe). À toute fin pratique, il y a peu de différences pour le développeur d'applications entre HTTP et HTTPS.

Les ports TCP et UDP sont utilisés pour la communication réseau. Les ports numérotés de 0 à 1023 sont réservés aux services système. Les services système qui utilisent ces ports incluent:

- **HTTP** (port 80): utilisé pour les transferts de pages Web.
- **HTTPS** (port 443): utilisé pour les transferts de pages Web sécurisées.

- **FTP** (port 21): utilisé pour le transfert de fichiers.
- **SSH** (port 22): utilisé pour les connexions sécurisées à distance.
- **SMTP** (port 25): utilisé pour le transfert de courrier électronique.
- **DNS** (port 53): utilisé pour la résolution de noms de domaine.
- **Telnet** (port 23): utilisé pour les connexions à distance non sécurisées.
- **POP3** (port 110): utilisé pour la récupération de courrier électronique à partir d'un serveur.
- **IMAP4** (port 143): utilisé pour la récupération de courrier électronique à partir d'un serveur.

Les ports numérotés de 1024 à 49151 sont appelés ports enregistrés et peuvent être utilisés par des applications utilisateur. Les ports numérotés de 49152 à 65535 sont appelés ports dynamiques ou privés et peuvent être utilisés par des applications utilisateur pour des connexions sortantes. Les applications peuvent utiliser n'importe quel port enregistré ou dynamique disponible pour les connexions sortantes. Il est fréquent de développer une application Web en utilisant un port comme 5000, 8080 ou 3000. Il n'est pas recommandé d'utiliser le même port pour deux applications de type serveur sur la même machine. Quand l'application est déployée sur Internet, il faut soit changer l'adresse du port ou utiliser une redirection des ports réservés (80 ou 443) vers le port utilisé par l'application.

Une application Web peut consommer et produire des données en divers formats. Les formats les plus courants sont le XML, le HTML et le JSON. Ce sont tous des formats textuels : il est possible de charger un document XML, un document HTML ou un document JSON dans un éditeur de texte.

Afin d'identifier la nature du contenu, le protocole HTTP utilise des types MIME. Les types MIME (Multipurpose Internet Mail Extensions) sont des identificateurs de format de données standardisés qui indiquent la nature et le format d'un document, d'un fichier ou d'un ensemble d'octets. Les types MIME sont utilisés pour identifier les types de fichiers sur Internet et sont utilisés par les navigateurs Web pour déterminer comment afficher ou traiter différents types de fichiers.

Les types MIME sont généralement représentés sous forme de chaînes de caractères qui contiennent deux parties: le type de média et le sous-type. Par exemple, le type MIME pour les fichiers HTML est text/html, où text est le type de média et html est le sous-type.

Les types MIME sont souvent utilisés dans les en-têtes HTTP pour spécifier le type de contenu d'une réponse HTTP. Par exemple, un serveur Web peut renvoyer une réponse avec l'en-tête `Content-Type: text/html` pour indiquer que la réponse contient du contenu HTML. Il existe des centaines de types MIME différents, chacun étant associé à un type de fichier spécifique.

13.1.2 XML

Le XML constitue une technologie importante dans le contexte de l'échange des données en ligne. Le XML est un « métalangage » permettant d'échanger de l'information. Nous disons que c'est un métalangage parce qu'il est une façon pratique de créer de nouveaux *langages* pour échanger des informations, mais qu'il ne constitue pas un langage en soi. On dit que le XML est « extensible » (peut être étendu) et que c'est un métalangage : les deux affirmations vont dans le même sens.

Un document XML est essentiellement du texte contenant des balises. Une balise est un segment de texte commençant par `<` et se terminant par `>`, comme `<personne>`. Voici un exemple simple de document XML :

```
<personne> <nom>Jean</nom> <age>42</age> </personne>
```

Le XML définit une grammaire stricte. On dit d'un document qui respecte cette grammaire qu'il est **bien formé**.

Revenons sur le concept de balise. Une balise est un segment de texte commençant par `<` et se terminant par `>`. Par exemple, `<lavie>` est une balise qui marque le début de l'élément `lavie`. Une balise commençant par `</`, comme `</lavie>`, est une balise de fin ; dans cet exemple, elle termine l'élément `lavie`. Le **nom XML** de la balise est le texte suivant le symbole `<` (ou `</` pour une balise de fin) et pouvant contenir n'importe quelle lettre ou chiffre (`a, b,... 0, 1, 2,...`) ou les trois symboles de ponctuation, soit la marque de soulignement (_), le trait d'union (-), les deux-points (:)ou le point (`.`); un nom XML ne peut contenir d'autres symboles de ponctuation, ni un espace. En outre, il ne peut pas commencer par un chiffre, un trait d'union ou un point. Par exemple, le nom XML de la balise `<lavie>` est `lavie`, alors que la balise `<7lavie>` ne serait pas autorisée. On réserve les noms débutant par `xml`, `Xml`, `xMl`, `xmL`, `XMl`, `xML`,

`Xml` et `XML` pour les spécifications XML. Ainsi, la balise `<xmldanslavie>` est à éviter.

Pour les balises de début, aussi dites d'ouverture, on peut ajouter un attribut au nom XML de la balise. Un attribut porte un nom XML qui doit respecter les mêmes règles que les noms XML des balises ; il est suivi du symbole `=` et d'une valeur placée entre guillemets ou apostrophes. Par exemple, la balise `<lavie age="5">` indique que l'élément `lavie` a un attribut (`age="5"`) qui a comme nom XML `age` et comme valeur `5`. Une balise peut avoir plusieurs attributs, comme `<lavie age="5" sexe="M">`, mais ils doivent tous porter des noms XML différents : `<lavie age="5" age="7">` n'est pas autorisée.

Par convention, on réserve l'attribut `xml:lang` à la spécification de la langue dans laquelle le texte est écrit. Cette spécification est optionnelle et assez rarement utilisée. Si on veut indiquer que le contenu d'un élément est en français, on fera comme dans cet exemple :

```
<explication xml:lang="fr">J'avais besoin d'une voiture.</explication>
```

On peut ajouter au code de la langue, un code de région comme dans `fr-CA` ou `en-US`. Le code de la langue et le code de la région doivent être séparés par un tiret. Les codes de langue ou de région qui peuvent être utilisées sont disponibles à l'adresse

http://www.iana.org/assignments/language-subtag-registry

Un **élément** est l'ensemble du texte borné par deux balises ayant le même nom XML, comme `<lavie>` et `</lavie>`. On dit que l'élément `<lavie></lavie>` a pour nom XML `lavie`. L'élément hérite des attributs de sa balise de départ : l'élément `<lavie age="5"></lavie>` a l'attribut `age="5"`. Il est à noter que la casse est significative en XML : les balises `<A>` et `<a>` n'ont pas le même nom XML. Dans le cas particulier où l'élément est vide (sans contenu), on peut remplacer `<lavie></lavie>` par `<lavie/>` pour abréger. Un élément peut contenir d'autres éléments, comme dans `<lavie><a></a></lavie>`, ou du texte, ou du texte et des éléments comme `<lavie>fd<a>fsd</a>fd</lavie>`.

Si un élément contient d'autres éléments, il doit aussi contenir, entre ses balises de début et de fin, les balises de début et de fin de chaque élément. Notez, de plus, que deux éléments ne peuvent se *chevaucher*, comme `<b><a></b></a>` ; avant de passer à un autre élément, il faut terminer le premier avec sa balise de fin. L'exemple précédent est du XML mal formé !

Un document XML ne doit avoir qu'un et un seul élément appelé « élément-racine », qui doit contenir tous les autres éléments. Il faut correctement ouvrir et fermer les éléments en séquence, ainsi `<a><b></a></b>` n'a aucun sens en XML, il faut plutôt écrire `<a><b></b></a>` ou alors `<a></a><b></b>`. Les éléments peuvent eux aussi avoir des attributs, par exemple `<a att="test"></a>`, mais il faut obligatoirement mettre la valeur de l'attribut entre guillemets (") ou apostrophes (').

Dans un document XML, l'ordre dans lequel les éléments sont présentés importe. Par exemple, les deux documents XML suivants ne sont pas équivalents.

```
1. <racine>
2.  <element1>http://www.google.com</element1>
3.  <element2>Un moteur de recherche</element2>
4. </racine>
5. <racine>
6.  <element2>Un moteur de recherche</element2>
7.  <element1>http://www.google.com</element1>
8. </racine>
```

En revanche, l'ordre dans lequel les attributs sont présentés est sans importance. Les deux documents XML suivants sont donc équivalents.

```
1. <racine attr1="test" attr2="retest"></racine>
2. <racine attr2="retest" attr1="test" ></racine>
```

Un élément peut contenir du texte, mais ne peut contenir le symbole `<`, comme dans l'exemple `<a><</a>`, parce que cela mène à de la confusion. On ne peut non plus y trouver le symbole `&` ou la séquence `]]>`. Que faire alors s'il faut utiliser le caractère `<` dans un texte mathématique, par exemple? Il faut utiliser un **appel d'entité**. Un appel d'entité est un bout de texte qui commence par une esperluette (`&`) et se termine par un point-virgule (`;`). On représente `<` par `<`, `&` par `&`, `>` par `>`, `"` par `&quot;` et `'` par `&apos;`. Ainsi, si le nom de votre compagnie est John&Smith, il ne faut pas utiliser un élément comme ceci `<nom>John&Smith</nom>` ; il faut plutôt utiliser :

```
<nom>John&amp;Smith</nom>
```

On utilise aussi les appels d'entités pour noter les valeurs des attributs. Supposons que la valeur d'un attribut est une apostrophe suivie d'un guillemet ('"). Les deux choix possibles `<nom att="'"">` et `<nom att='&apos;&quot;'>` ne sont pas valables. Dans ce cas, il faudra écrire `<nom att="'&quot;">`, `<nom att='&apos;"'>`, `<nom att="&apos;&quot;">` ou `<nom att='&apos;&quot;'>`.

Il arrive parfois qu'il soit trop lourd d'utiliser des appels d'entités, et on peut alors utiliser un segment `CDATA`. Un tel segment débute par `<![CDATA[` et se termine par `]]>`. Tout le texte au sein du segment est rendu tel quel, sans modification ou interprétation (*verbatim*). Ainsi, le texte `<![CDATA[<monelement />]]>` est équivalent à `<monelement />`.

Dans un document XML, on peut ajouter un commentaire qui est normalement destiné à être lu par un humain. Par exemple, dans un fichier de configuration XML, les commentaires pourraient être utilisés pour expliquer la signification des différents éléments pour qu'un humain puisse faire des modifications au besoin.

Un commentaire commence par `<!--` et se termine par `-->` et ne doit pas contenir deux tirets de suite (`--`) entre ces deux bornes, il ne doit pas se terminer par un tiret, mais peut contenir n'importe quel autre texte. Un commentaire peut apparaître avant ou après l'élément-racine, dans un élément, entre deux éléments, etc. Cependant, un commentaire ne peut pas apparaître au sein d'une balise, comme `<a <!--ceci est la balise a-->>`. C'est du XML mal formé !

La norme Unicode est de plus en plus utilisée et elle est popularisée en partie par son utilisation dans le XML. En effet, par défaut, un document XML utilise Unicode (spécifiquement UTF-8 ou UTF-16) et ce sont les seuls jeux de caractères qu'un processeur XML est requis de connaître. L'avantage principal d'Unicode est sa richesse : la version 5.0 du format Unicode permet de représenter 99 000 caractères différents incluant les caractères Klingon de Star Trek.

Il arrive fréquemment qu'on utilise des hyperliens dans les documents XML. On distingue deux types d'hyperliens : les hyperliens contenant une adresse absolue (telle que `http://domaine.ca/pong.png`) et les hyperliens contenant des adresses relatives (`pong.png`). On reconnaît les adresses relatives parce qu'elles ne débutent pas par un protocole tel que http ou ftp. Pour compléter les adresses relatives et les transformer en adresses absolues, on utilise fréquemment l'adresse du document qui contient l'adresse relative. Ainsi, si le document suivant se trouve à l'adresse `http://domaine.ca/fichier.xml`, alors l'adresse relative `pong.png` devient l'adresse absolue `http://domaine.ca/pong.png` :

```
1. <racine>
2. adresse relative:
3. <image lien="pong.png" />
4. </racine>
```

Le XML permet de spécifier un attribut `xml:base` qui sert à interpréter les adresses relatives du document. Le document suivant pointe vers une image à l'adresse `http://domaine.ca/image/pong.png` :

```
1. <racine xml:base="http://domaine.ca/" >
2. <p xml:base="image/">
3. <image lien="pong.png" />
4. </p>
5. </racine>
```

On peut extraire des données d'un document XML en Python de différentes façons. Une approche conventionnelle est de convertir le document XML en arbre (DOM, *Document Object Model*) et d'accéder aux nœuds du documents, comme dans cet exemple :

```
1. import xml.dom.minidom
2. document =
xml.dom.minidom.parseString("<element><t1>1</t1></element>")
3. print(getElementsByTagName("t1")[0].childNodes[0].nodeValue) #
affiche '1'
```

Nous pouvons créer de nouveaux documents XML d'une manière semblable :

```
1. document = xml.dom.minidom.Document()
2. e = document.createElement("element")
3. t1 = document.createElement("t1")
4. e.appendChild(t1)
5. document.appendChild(e)
6. text = document.createTextNode('1')
7. t1.appendChild(text)
8. document.toprettyxml() //'<?xml version="1.0"
?>\n<element>\n\t<t>1</t>\n</element>\n'
```

Exercice. Concevez un programme Python qui représente un tableau d'entiers en XML. Votre programme doit créer un document XML et le lire ensuite pour récupérer le tableau à partir du XML.

Solution. La solution suivante comprend deux fonctions. Premièrement, `tableau_vers_xml(tableau)` est une fonction qui prend un tableau de nombres comme paramètre et renvoie une chaîne de caractères au format XML. Elle utilise les fonctions du module `xml.dom.minidom` pour créer un document XML, y ajouter un élément racine nommé `<tableau>`, et y insérer des éléments enfants nommés `<element>` contenant les valeurs du tableau converties en chaînes de caractères. Elle utilise la méthode `toprettyxml()` pour formater le document XML avec des indentations et des sauts de ligne, et le renvoie comme résultat. Deuxièmement, `xml_vers_tableau(xmlchaine)` est une fonction qui prend une chaîne

de caractères au format XML comme paramètre et renvoie un tableau de nombres. Elle utilise la fonction `parseString()` du module `xml.dom.minidom` pour créer un document XML à partir de la chaîne de caractères. Elle utilise ensuite la méthode `getElementsByTagName()` pour obtenir une liste de tous les éléments nommés `<element>` dans le document XML. Elle parcourt cette liste et ajoute à un tableau vide les valeurs numériques des éléments, en utilisant la méthode `firstChild()` pour accéder au contenu textuel des éléments, et la fonction `int()` pour convertir les chaînes de caractères en nombres entiers. Elle renvoie ensuite le tableau comme résultat.

CodePython/chapitre14/Exercice1.py

```
1.  import xml.dom.minidom
2.
3.  def tableau_vers_xml(tableau):
4.      document = xml.dom.minidom.Document()
5.      tab = document.createElement("tableau")
6.      document.appendChild(tab)
7.      for i in tableau:
8.          element = document.createElement("element")
9.          element.appendChild(document.createTextNode(str(i)))
10.         tab.appendChild(element)
11.     return document.toprettyxml()
12.
13. def xml_vers_tableau(xmlchaine):
14.     tableau = []
15.     document = xml.dom.minidom.parseString(xmlchaine)
16.     for element in document.getElementsByTagName("element"):
17.         tableau.append(int(element.firstChild.data))
18.     return tableau
19.
20. tableau = [1, 2, 3, 4, 5]
21. xmlchaine = tableau_vers_xml(tableau)
22. print(xmlchaine)
23. print(xml_vers_tableau(xmlchaine))
```

13.1.3 HTML

Le HTML a été inventé par Tim Berners-Lee, alors qu'il travaillait pour le centre de recherche CERN en Suisse. L'intention de Berners-Lee était de proposer un système révolutionnaire de gestion de l'information qu'il appela World Wide Web. Comme pièce maîtresse de son architecture, il avait besoin d'un format de documents qu'il appela HTML (*HyperText Markup Language*). La première page Web, dont l'adresse était `http://nxoc01.cern.ch/hypertext/WWW/TheProject.html`, vit le jour en 1990.

Un document HTML comprend un élément-racine « html » contenant deux éléments : un élément `head` et un élément `body`. L'élément `head` doit contenir un élément `title`, alors que l'élément body peut être vide, mais peut aussi posséder du contenu XML mixte (texte et diverses balises).

L'élément `body` contient des balises et du texte. Les retours de ligne sont traités comme des espaces normaux.

```
 1. <!doctype html>
 2. <html>
 3. <html>
 4. <head>
 5. <title>Titre de mon document</title>
 6. </head>
 7. <body>
 8. Voici mon document.
 9. Voici ma vie.
10. Voici mon chat.
11. </body>
12. </html>
```

La question qui se pose alors est : comment faire des paragraphes? En effet, avec notre dernier exemple, le texte s'affichera sur une seule ligne. La solution consiste à utiliser l'élément `p` pour `paragraphe`. Pour avoir trois paragraphes, on remplace le document HTML précédent par celui qui suit. Observez que chaque balise `p` ouverte doit être fermée.

```
 1. <!doctype html>
 2. <html>
 3. <head>
 4. <title>Titre de mon document</title>
 5. </head>
 6. <body>
 7. <p>Voici mon document.</p>
 8. <p>Voici ma vie.</p>
 9. <p>Voici mon chat.</p>
10. </body>
11. </html>
```

Supposons que l'on veuille faire une liste, comme ceci :

- Premier point : le chat est noir.
- Second point : le chat est blanc.
- Dernier point : le chat est marron.

On peut obtenir ce résultat avec un élément `ul` pour *unordered list* (liste sans ordre), contenant des éléments `li`. Observez que chaque balise `li` ouverte doit être fermée.

```
 1. <!doctype html>
 2. <html>
 3. <head>
 4. <title>Titre de mon document</title>
 5. </head>
 6. <body>
```

```
 7. <ul>
 8. <li>Premier point: le chat est noir.</li>
 9. <li>Second point: le chat est blanc.</li>
10. <li>Dernier point: le chat est marron.</li>
11. </ul>
12. </body>
13. </html>
```

Supposons maintenant que nous voulions une liste avec un compteur :

1. Le chat est noir.
2. Le chat est blanc.
3. Le chat est marron.

Il suffit alors de remplacer l'élément `ul` par l'élément `ol` pour *ordered list* (liste ordonnée) :

```
 1. <!doctype html>
 2. <html>
 3. <head>
 4. <title>Titre de mon document</title>
 5. </head>
 6. <body>
 7. <ol>
 8. <li>Le chat est noir.</li>
 9. <li>Le chat est blanc.</li>
10. <li>Le chat est marron.</li>
11. </ol>
12. </body>
13. </html>
```

Supposons maintenant que nous voulions produire un tableau. On peut l'obtenir à l'aide d'un élément « table ». Cet élément contiendra plusieurs éléments `tr` (éléments correspondant à une ligne) qui eux-mêmes contiennent des éléments `td` (éléments correspondant à une cellule).

```
 1. <!doctype html>
 2. <html>
 3. <head>
 4. <title>Titre de mon document</title>
 5. </head>
 6. <body>
 7. <table>
 8. <tr>
 9. <td>Nom</td>
10. <td>Valeur</td>
11. </tr>
12. <tr>
13. <td>Mustang</td>
14. <td>50 $</td>
15. </tr>
16. <tr>
```

```
17. <td>Ferrari</td>
18. <td>500 $</td>
19. </tr>
20. </table>
21. </body>
22. </html>
```

Tous les tableaux ne sont pas si simples. On peut faire en sorte qu'une même cellule occupe deux colonnes (`<td colspan="2">`) ou deux rangées (`<td rowspan="2">`). Aussi, souvent, on utilise l'élément `th` au lieu de l'élément `td` pour désigner la première rangée d'un tableau lorsque celle-ci forme l'entête descriptive du tableau. Il est aussi possible d'utiliser un élément `caption` au sein d'un tableau pour noter le titre du tableau :

```
 1. <!doctype html>
 2. <html>
 3. <head>
 4. <title>Titre de mon document</title>
 5. </head>
 6. <body>
 7. <table border="1">
 8. <caption>Valeur de différents véhicule</caption>
 9. <tr>
10. <th>Nom</th>
11. <th>Valeur</th>
12. </tr>
13. <tr>
14. <td>Mustang</td>
15. <td>50 $</td>
16. </tr>
17. <tr>
18. <td>Ferrari</td>
19. <td>500 $</td>
20. </tr>
21. </table>
22. </body>
23. </html>
```

On peut très facilement utiliser des effets de polices de caractères en HTML. Pour obtenir des caractères en italique, par exemple *maman*, il suffit d'utiliser un élément `i` comme ceci : `<i>maman</i>`. Pour des caractères en gras, comme **maman**, il suffit d'utiliser un élément `b` comme ceci : `<b>maman</b>`. On peut également combiner les deux, comme ***maman***, en écrivant `<i><b>maman</b></i>` ou bien `<b><i>maman</i></b>`. Il est cependant préférable d'utiliser `em` (emphase) au lieu de `i` et `strong` (fort) au lieu de `b` : le navigateur choisira alors de rendre le texte dans un élément `em` avec un italique ou une autre technique appropriée, et de rendre le texte dans un élément `strong` en caractères gras ou une autre technique appropriée. On évite ainsi de confondre la présentation (italique ou gras) et la sémantique (emphase ou point fort). Dans le cas où un terme est défini,

vous devriez utiliser un élément `dfn` (définition) comme dans cet exemple:

```
La <dfn>mort</dfn> est la fin de la vie
```

La plupart des navigateurs afficheront alors le mot « mort » en caractères italiques.

Pour insérer une image dans un document HTML, il suffit d'utiliser une balise `img` avec comme attribut `src` pour source, soit l'URL. Par exemple, le code

```
<img src="https://monsite.ca/image.svg">
```

permettra d'insérer une image dans un document. Il est préférable de prévoir que l'image pourrait ne pas être trouvée ou affichée en ajoutant un attribut `alt` qui contient du texte décrivant le contenu de l'image; si l'image n'est pas disponible, le texte contenu dans l'attribut `alt` s'affichera. Le résultat final prend la forme :

```
<img src="https://monsite.ca/image.svg " alt="logo de mon entreprise" />
```

Pour inclure un hyperlien, il suffit d'utiliser la syntaxe :

```
<a href="http://monsite.ca/">mon site</a>
```

On peut également ajouter des marqueurs dans une page Web en utilisant la syntaxe :

```
<a name="point1">Premier point de mon document</a>
```

Contrairement à la syntaxe `<a href="...">`, l'attribut `name` n'ajoute pas un hyperlien, mais un marqueur généralement invisible dans la page. Par exemple, si la page contient plusieurs sections, un marqueur peut être ajouté au début de chaque section. On pourra alors faire des liens non seulement vers le document, mais aussi vers la section marquée dans le document. Ainsi, l'hyperlien

```
<a href="pageweb.html#point1">aller vers le premier point du document pageweb.html</a>
```

mènera l'utilisateur dans le document `pageweb.html`, précisément au marqueur du `point 1`, s'il existe, évidemment.

Dans un texte, il arrive qu'on veuille utiliser des exposants et des indices. Les éléments `sup` et `sub` servent à cette fonction. Par exemple, « premier » peut s'écrire `1<sup>er</sup>`. Il n'est malheureusement pas possible de noter automatiquement des notes en bas de page en HTML.

Il y a plusieurs éléments permettant de traiter de la programmation informatique ou des mathématiques. Le code informatique peut s'écrire dans un élément `code`. Le texte saisi à l'écran par un utilisateur peut s'écrire dans un élément `kbd`. Les exemples de sortie à l'écran peuvent s'écrire dans un élément `samp` (pour *sample*) et les variables peuvent s'écrire dans un élément `var`. Voici un exemple :

```
<p>La valeur de la variable <var>i</var> est obtenue avec ce code:</p><code>int i = 1; i+=1</code><p>On s'attend à ce que l'utilisateur tape <kbd>Yes</kbd> pour oui. Voici un exemple de résultat: <samp>Error!</samp>.</p>
```

Pour citer quelqu'un, on peut utiliser un élément `q` lorsqu'il s'agit d'une courte citation au sein d'un paragraphe ou élément `blockquote` lorsque la citation doit former un court paragraphe. Plusieurs navigateurs mettent automatiquement le contenu de l'élément `q` entre guillemets et le contenu de l'élément `blockquote` en retrait. Voici un exemple :

```
 <p> Jean a dit: <q>mon cher!</q>. Par la suite, je lui ai lu ce texte fatidique.</p><blockquote><p>Oh! Comme la mer a merée! Oh! Comme j'ai jéjé!</p></blockquote>
```

Les éléments `del` et `ins` permettent de noter un retrait et un ajout, respectivement, comme dans cet exemple:

```
J'ai <del>marié</del><ins>épousé</ins> ta mère
```

Le navigateur se chargera d'afficher les retraits et les ajouts de manière compréhensible.

Le HTML respecte la spécification XML voulant qu'on indique la langue dans laquelle est écrite un texte avec l'attribut `xml:lang`. Cet attribut est optionnel, mais peut s'avérer pratique. Dans le cas où un texte en langue étrangère est présent dans un paragraphe, on peut utiliser l'élément `span` pour en indiquer la langue. Voici un exemple :

```
<p xml:lang="fr-CA">Jean aime les <span xml:lang="en-US">computers</span></p>
```

L'élément `span` ne sert qu'à nous permettre de sélectionner un texte au sein d'un autre élément.

Ajouter des commentaires en HTML est facile. Les commentaires du XML s'appliquent: il suffit de débuter le commentaire par `<!--` et de le terminer par `-->`. Il n'est pas permis d'inclure au sein d'un commentaire deux tirets (`--`) ni de terminer un commentaire par un tiret.

On peut aussi ajouter une figure à un document avec une légende:

```
1. <figure>
2.  <img src="http://lemire.me/fr/images/JPG/profile2011B_152.jpg"
/>
3.  <legend>Photo du professeur</legend>
4. </figure>
```

Exercice. Produisez une page HTML représentant un document ayant trois sections. Votre document doit comprendre une table des matières qui comportent des liens vers les trois sections.

Solution. Pour créer la table des matières, nous utilisons des balises `<a>` dont la fonction est de créer un lien hypertexte, c'est-à-dire un texte cliquable qui renvoie vers une autre page ou une autre partie de la même page. Elle a un attribut href qui indique l'adresse du lien, et un attribut id qui indique un identifiant unique pour l'élément. On peut utiliser cet identifiant pour créer des liens internes, en utilisant le symbole `#` suivi de l'identifiant, comme `href="#section1"`. Si vous document est trop court, il peut être difficile de voir l'intérêt de la table des matières. Dans la solution proposée, nous utilisons `margin-bottom` pour définir la marge inférieure d'un élément, c'est-à-dire l'espace entre le bord inférieur de l'élément et le bord supérieur de l'élément suivant. La valeur peut être exprimée en pixels (`px`), en centimètres (`cm`), en pourcentage (`%`), etc. Par exemple, `style="margin-bottom:10cm;"` crée une marge inférieure de 10 centimètres pour l'élément. Une telle marge nous permet de faire un document qui couvre beaucoup d'espace vertical sans avoir à inclure beaucoup de contenu.

CodePython/chapitre14/Exercice2.html

```
1.    <!DOCTYPE html>
2.    <html>
3.    <head>
4.        <meta charset="UTF-8">
5.        <title>Exemple de page html</title>
6.    </head>
7.    <body>
8.        <h1>Exemple de page html</h1>
9.        <p>Cette page contient trois sections et une table des matières comprenant des liens
vers les trois sections.</p>
10.       <h2>Table des matières</h2>
11.       <ul>
12.           <li><a href="#section1">Section 1</a></li>
13.           <li><a href="#section2">Section 2</a></li>
14.           <li><a href="#section3">Section 3</a></li>
15.       </ul>
16.       <h2 id="section1">Section 1</h2>
17.
18.       <p style="margin-bottom:10cm;">Ceci est la deuxième section de la page. Elle contient
du texte. </p>
19.
20.
21.       <h2 id="section2">Section 2</h2>
22.       <p>Ceci est la deuxième section de la page. Elle contient du texte et une liste.</p>
23.       <ul style="margin-bottom:10cm;">
24.           <li>Un élément de la liste</li>
25.           <li>Un autre élément de la liste</li>
```

```
26.          <li>Encore un élément de la liste</li>
27.      </ul>
28.
29.
30.      <h2 id="section3">Section 3</h2>
31.      <p>Ceci est la troisième et dernière section de la page. Elle contient du texte et un
tableau.</p>
32.      <table style="margin-bottom:30cm;">
33.          <tr>
34.              <th>Nom</th>
35.              <th>Prénom</th>
36.              <th>Age</th>
37.          </tr>
38.          <tr>
39.              <td>Dupont</td>
40.              <td>Jean</td>
41.              <td>25</td>
42.          </tr>
43.          <tr>
44.              <td>Durand</td>
45.              <td>Marie</td>
46.              <td>30</td>
47.          </tr>
48.          <tr>
49.              <td>Martin</td>
50.              <td>Pierre</td>
51.              <td>35</td>
52.          </tr>
53.      </table>
54.
55.  </body>
56.  </html>
```

13.1.4 CSS

Au sein d'un navigateur, la forme un document HTML ou XML est précisée par un ou plusieurs documents CSS. Un document CSS permet de spécifier comment le contenu du document XML ou HTML sera affiché.

Pour l'essentiel, le langage CSS prend la forme d'une succession d'affirmations de la forme élément `{propriété: valeur; autrepropriété: valeur;}`.

Supposons un document XML comme celui-ci :

```
 1.   <?xml version="1.0" ?>
 2.   <comptearecevoir>
 3.   <facture>
 4.    <personne>Jean Rochond</personne>
 5.   <montant>10.10</montant>
 6.   <raison>Achat d'ordinateur</raison>
 7.   </facture>
 8.   <facture>
 9.   <personne>Madeleine Bédard</personne>
10.   <montant>20.10</montant>
11.   <raison>Achat d'un crayon</raison>
12.   </facture>
13.   </comptearecevoir>
```

Il est possible de l'afficher avec de la couleur ou de l'italique, comme ceci :

Nous pouvons obtenir ce résultat à l'aide du fichier CSS suivant :

```
facture {
display: block;
margin-bottom: 30pt;
}

montant {
color: red;
}

raison {
display: block;
font-style: italic;
margin-left: 1cm;
}
```

Pour vérifier que c'est bien le cas, il suffit de créer un fichier `chap12.css` avec le contenu CSS précédent et de modifier le fichier XML en y ajoutant une ligne pointant vers le fichier CSS (`<?xml-stylesheet type="text/css" href="chap12.css"?>`), comme ceci :

```
<?xml version="1.0" ?>
<?xml-stylesheet type="text/css" href="chap12.css"?>
<comptearecevoir>
<facture>
 <personne>Jean Rochond</personne>
<montant>10.10</montant>
<raison>Achat d'ordinateur</raison>
</facture>
<facture>
<personne>Madeleine Bédard</personne>
<montant>20.10</montant>
<raison>Achat d'un crayon</raison>
</facture>
</comptearecevoir>
```

Si le fichier XML est dans le même répertoire que le fichier CSS, le navigateur devrait présenter le document XML avec le montant en rouge et le commentaire (raison) en italique, comme nous l'avons présenté plus haut.

Examinons maintenant les différentes instructions du fichier CSS.

- L'instruction `display: block;` déclare que l'élément devrait former son propre paragraphe. L'instruction `display: none;` rend l'élément invisible.

- Les instructions `margin-bottom: 30pt;` et `margin-left: 1cm;` définissent des marges en bas et à gauche de 30 points et de 1 cm respectivement.

- L'instruction `color: red;` affirme que le contenu de l'élément devrait être écrit en rouge, alors que `font-style: italic;` nous dit que le texte de l'élément devrait être en italique. On pourrait aussi contrôler la couleur de fond avec un instruction comme `background-color:red`.

Pour définir une couleur précise, et non les couleurs courantes comme `red`, `green`, `blue`, `yellow`, `white`, `black`, etc., elle peut-être spécifiée selon sa composition en couleurs de base RGB (« `red` », « `green` , « `blue` ») avec une instruction comme `background-color:rgb(200,200,200);` où chaque valeur numérique est entre 0 et 255 inclusivement.

En CSS, la taille d'un objet peut être spécifiée avec plusieurs unités de mesure, par exemple `cm` pour centimètre ou `px` pour pixel. Ainsi donc l'instruction `width:1px` spécifie une largeur de 1 pixel. On peut aussi utiliser des unités relatives comme `em`, `rem` ou `%`. Une mesure de `50%` indique que l'objet devrait occuper la moitié de l'espace disponible. Une mesure de `1em` correspond à la taille de la police de caractère dans l'élément courant alors que `1rem` correspond à la taille de la police de caractère dans l'élément-racine du document. On peut aussi combiner les unités, par exemple, pour spécifier une dimension correspondant à tout l'espace disponible moins 80 pixels, on peut utiliser la valeur `calc(100% - 80px)`.

L'affichage d'un élément peut être contrôlé avec la propriété `display` qui peut prendre plusieurs valeurs dont celles-ci :

- `display: none` : l'élément ne doit pas s'afficher. Par exemple, l'instruction `img{display: none;}` fait en sorte que les images ne s'affichent plus en HTML.
- `display: inline` : l'élément s'affiche à la suite du précédent comme s'il s'agissait d'un caractère.
- `display: block` : l'élément s'affiche dans un bloc distinct, comme un nouveau paragraphe, par exemple.
- `display: list-item` : l'élément s'affiche comme un élément d'une liste.

Voici un exemple :

```
1.   p { display: block }
2.   strong { display: inline }
3.   li { display: list-item }
4.   img { display: none }
```

La propriété `float` d'un élément lui permet de sortir du flot normal des éléments et de se placer à gauche ou à droite. Par exemple, une image en HTML s'affiche normalement comme un bloc. On peut forcer l'image à s'intégrer au paragraphe suivant avec l'instruction `float: right` ou `float: left`. La propriété `float` permet aussi de créer plusieurs colonnes de texte comme dans un journal.

Bien que la justification du texte puisse être changée avec une instruction comme `text-align: center`, centrer un élément requiert plutôt une manipulation des marges avec la valeur spéciale `auto`, comme dans cet exemple :

```
1.   p { width: 5cm;
2.   margin-left: auto;
3.   margin-right: auto;}
```

Il aurait été sans doute préférable d'avoir une instruction dédiée pour centrer les éléments comme il s'agit d'une opération courante.

Des commentaires peuvent être ajoutés à un fichier CSS et ils sont systématiquement ignorés par la machine. Un bloc de commentaire débute par `/*` et se termine par `*/`.

```
1.  /* mon fichier css */
2.  montant {
3.   color: red; /* la couleur rouge */
4.  }
```

On peut sélectionner la première ligne d'un élément s'affichant en mode `block` et le premier caractère de tout élément avec les sélecteurs `:first-line` et `:first-letter` respectivement. Voici un exemple :

```
1.   p:first-line {text-transform: uppercase}
2.   p:first-letter {font-size: 200%;float: left;}
```

Avec CSS, on peut demander qu'une certaine chaîne de caractères apparaisse avant ou après un élément. L'exemple suivant montre comment ajouter automatiquement des guillemets avant et après un élément :

```
1.   blockquote:before {content:"«";}
2.   blockquote:after {content:"»";}
```

Ceci n'est pas limité au texte cependant. Il est possible, par exemple, d'ajouter automatiquement une image avant chaque élément comme ceci :

```
1.   p:before {content:url("monimage.png";}
```

Certains sélecteurs n'agissent qu'en réponse aux comportements de l'utilisateur. Par exemple, le sélecteur `p:hover` sélectionne les éléments `p` qui sont survolés par le curseur de la souris. Il existe plusieurs sélecteurs d'interaction dont `:link` (lien non visité), `:visited` (lien visité), `:active` (l'utilisateur utilise un élément), `:focus` (l'élément est sélectionné par l'utilisateur). On peut aussi combiner les sélecteurs comme dans `a:hover:focus`. On appelle aussi ces sélecteurs pseudo-class.

Dans le cas du HTML, les navigateurs utilisent une liste de règles par défaut. Ces règles vont varier d'un navigateur à l'autre, mais voici un exemple de règles utilisées par des navigateurs :

```
 1. html, div {
 2.   display: block;
 3. }
 4.
 5. body {
 6.   display: block;
 7.   margin: 8px;
 8. }
 9.
10. p, dl, multicol {
11.   display: block;
12.   margin: 1em 0;
13. }
14.
15. blockquote {
16.   display: block;
17.   margin: 1em 40px;
18. }
19.
20. h1 {
21.   display: block;
22.   font-size: 2em;
23.   font-weight: bold;
24.   margin: .67em 0;
25. }
26.
27. h2 {
28.   display: block;
29.   font-size: 1.5em;
30.   font-weight: bold;
31.   margin: .83em 0;
32. }
33.
```

```
34.
35. pre {
36.   display: block;
37.   white-space: pre;
38.   margin: 1em 0;
39. }
40.
41. b, strong {
42.   font-weight: bolder;
43. }
44.
45. i, cite, em, var, dfn {
46.   font-style: italic;
47. }
48.
49.
50. u, ins {
51.   text-decoration: underline;
52. }
53.
54. s, strike, del {
55.   text-decoration: line-through;
56. }
57.
58. big {
59.   font-size: larger;
60. }
61.
62. small {
63.   font-size: smaller;
64. }
65.
66. sub {
67.   vertical-align: sub;
68.   font-size: smaller;
69.   line-height: normal;
70. }
71.
72. sup {
73.   vertical-align: super;
74.   font-size: smaller;
75.   line-height: normal;
76. }
77.
78. ul, menu, dir {
79.   display: block;
80.   list-style-type: disc;
81.   margin: 1em 0;
82. }
83.
84. ol {
85.   display: block;
86.   list-style-type: decimal;
```

```
87.   margin: 1em 0;
88. }
89.
90. li {
91.   display: list-item;
92. }
93.
94. area, base, basefont, head, meta, script, style, title,
95. noembed, param {
96.    display: none;
97. }
```

L'astérisque nous permet d'appliquer une règle à tous les éléments, comme dans `*{color:red;}`.

Avec les CSS, en utilisant les crochets, nous pouvons sélectionner tous les éléments ayant un attribut donné. Par exemple, l'instruction `*[monattribut] {color:red;}` met en rouge tous les éléments ayant un attribut portant le nom `monattribut`. Nous pouvons aussi, bien sûr, limiter la sélection à des éléments portant un nom donné comme dans `maman[monattribut] {color:red;}` où les éléments `maman` ayant un attribut `monattribut` seront en rouge. Finalement, nous pouvons de plus limiter la sélection à des attributs ayant une certaine valeur, comme dans `maman[monattribut="papa"] {color:red;}`.

Il arrive fréquemment qu'une valeur d'attribut contiennent plusieurs mots, comme dans `<amerique pays="États-Unis Canada" />`. Pour sélectionner tous les éléments dont un attribut contient un mot particulier, on remplace `=` par `~=` comme dans l'instruction `*[pays~="Canada"] {color:red;}` qui met en rouge tout élément dont l'attribut `pays` contient le mot `Canada`. Les mots doivent être séparés par des espaces. Dans l'éventualité où les mots sont séparés par des tirets, comme dans `<paragraphe langue="français-anglais" />`, on peut obtenir le même résultat avec `|=` comme dans `*[langue|="français"] { color:red;}`.

Il est possible en XML ou en HTML de spécifier la langue dans laquelle est écrit un texte avec l'attribut `xml:lang`. On pourrait penser que pour mettre le texte déclaré comme étant en anglais en rouge, il suffirait de l'instruction `*[lang="fr"] {color:red;}`, mais que se passera-t-il si on a utilisé un code de région avec la langue comme `fr-CA` ? Une solution plus élégante consiste alors à utiliser la sélection sur la langue avec une instruction comme `:lang(en) {color:red;}`.

Supposons maintenant que nous voulions afficher en rouge tous les éléments `facture` et `maison`. Nous pouvons le faire avec deux instructions :

```
facture {color:red;}
maison {color:red;}
```

En pratique cependant, il est préférable d'utiliser la virgule pour grouper les éléments, comme ceci :

```
facture, maison {color:red;}
```

Les deux instructions et cette dernière forme sont exactement équivalentes.

Supposons maintenant que nous ne voulions pas afficher tous les éléments `personne` en rouge, mais seulement les éléments `personne` contenus dans un élément `facture`. Ce résultat est obtenu en plaçant les deux noms d'élément côte-à-côte (séparé par un espace). Ainsi, l'instruction `facture personne {color:red;}` affichera en rouge tous les éléments « personne » contenus dans un élément `facture`, comme dans l'exemple qui suit :

```
1.  <?xml version="1.0" ?>
2.  <?xml-stylesheet type="text/css" href="chap12.css"?>
3.  <comptearecevoir>
4.  <facture>
5.  <personne>Jean Rochond</personne>
6.  </facture>
7.  </comptearecevoir>
```

Nous pourrions vouloir que seuls les éléments immédiatement contenus dans l'élément « facture », comme dans le premier exemple, soient en rouge, et non pas ceux qui sont contenus dans des éléments eux-mêmes dans un élément « facture » (deuxième exemple). Nous pouvons obtenir ce résultat avec l'instruction `facture > personne { color:red;}`.

En somme, la règle `a b { color:red;}` s'applique à l'élément `b`, si l'élément `b` est contenu dans un élément `a` comme dans `<a><b></b></a>` ou `<a><c><b></b></c></a>`.

Par contre, la règle `a > b { color:red;}` s'applique à l'élément `b`, si et seulement si l'élément `b` est immédiatement contenu dans un élément `a`, comme dans `<a><b></b></a>`. Elle ne s'applique toutefois pas si `b` est contenu dans un élément lui-même contenu dans `a`, comme dans `<a><c><b></b></c></a>`.

Supposons maintenant, dans l'exemple suivant, que nous voulions indenter le premier paragraphe (élément `p`) suivant le titre (élément `titre`) :

```
1.   <?xml version="1.0"  ?>
2.   <titre>Mon histoire</titre>
3.   <p>C'est triste.</p>
4.   <p>Oui, c'est triste.</p>
```

Ce résultat s'obtient avec l'instruction `titre + p {text-indent: 0cm;}`. La syntaxe `a + b { ... }` s'applique à l'élément `b` quand les éléments `a` et `b` sont contenus dans le même élément, et que `b` est immédiatement après `a`. Notez que la règle `a + b { ... }` s'applique à `b`, mais ne s'applique pas à `a`.

Supposons que nous désirions indenter le premier paragraphe (élément `p`) dans l'élément `section` :

```
1.   <?xml version="1.0" ?>
2.   <section titre="Mon histoire" >
3.   <p>C'est triste.</p>
4.   <p>Oui, c'est triste.</p>
5.   </titre>
```

On peut obtenir ce résultat avec le sélecteur `section > p:first-child { ... }` où `:first-child` signifie que seuls les éléments `p` étant le premier élément au sein d'un autre élément sont sélectionnés. En fait, dans cet exemple particulier, on obtiendrait aussi le résultat voulu avec le sélecteur `p:first-child { ... }`.

Par ailleurs, nous pouvons combiner les espaces, les `+`, les virgules et les `>` dans un même sélecteur. Par exemple, `a + b, c { ... }` s'applique aux éléments `c` et aux éléments « b » qui suivent immédiatement un élément `a`.

Si vous avez des éléments ayant des attributs ayant une valeur de type `ID`, leur valeur est un nom XML et elle ne doit être utilisée qu'une seule fois. C'est le cas des attributs de la forme `id="xxx"` que l'on peut utiliser avec pratiquement tous les éléments HTML. On peut sélectionner un élément basé sur la valeur d'un tel attribut en utilisant le symbole `#` :

```
1.  #xxx {
2.   color: red;
3.  }
```

Dans ce cas, le contenu d'un élément comme une balise HTML `<p id='xxx'>test</p>` s'affichera en rouge. On peut combiner les sélecteurs `#` avec les autres règles que nous avons traitées, par exemple, le code suivant

ne mettra en rouge que les éléments `i` contenus dans un élément ayant un attribut de type `ID` avec pour valeur `xxx` :

```
1.  #xxx i{
2.   color: red;
3.  }
```

Plusieurs instructions peuvent s'appliquer en même temps : un texte peut être en rouge et souligné. Il peut arriver cependant que deux instructions CSS se contredisent. Par exemple, un texte ne peut être à la fois en rouge et en bleu. Dans ce cas, la sélection la plus spécifique l'emporte. Ainsi, dans l'exemple qui suit...

```
 1.   * {
 2.    color: black;
 3.   }
 4.
 5.   montant {
 6.    color: red;
 7.   }
 8.
 9.   montant > montant {
10.    color: yellow;
11.   }
```

Les éléments `montant` seront en rouge, sauf s'ils sont immédiatement contenus dans un autre élément `montant`, auquel cas ils seront en jaune.

Par ailleurs, si deux sélections de même spécificité sont rencontrées, c'est la dernière qui l'emporte. Ainsi, dans l'exemple qui suit les éléments `montant` seront en noir.

```
1.  montant {
2.   color: red;
3.  }
4.
5.  montant {
6.   color: black;
7.  }
```

On peut cependant forcer le navigateur à considérer une règle qui apparaît avant une autre comme ayant tout de même priorité (à spécificité égale). Il suffit d'utiliser la mention `!important`.

```
1.  montant {
2.   color: red !important;
3.  }
4.
5.  montant {
6.   color: black;
```

```
7.  }
```

Les éléments `montant` seront en rouge. Notez que la mention `!important` permet souvent d'ignorer la spécificité de la règle : les règles avec cette mention l'emportent toujours sur les autres. (Si une autre règle avec la mention `!important` a une plus grande spécificité, elle aura bien sûr priorité.)

Les règles par défaut utilisées par votre navigateur dans le cas du HTML sont lues en premier. De cette manière, toutes les règles que vous stipulez dans vos propres fichiers CSS et à même le XML ou HTML ont priorité pour une même spécificité. On peut considérer que les règles avec la mention `!important` sont lues en dernier dans la mesure où elles l'emportent toujours sur les autres.

En CSS, on peut spécifier la hauteur et la largeur d'un objet à l'aide des instructions `width` et `height`. Par exemple, pour indiquer qu'un paragraphe devrait n'avoir qu'une largeur de 20 pixels, on utilise un sélecteur comme celui-ci :

```
1.  p {
2.   width: 20px;
3.  }
```

Le CSS supporte plusieurs unités de mesure outre les pixels dont les pouces (`in`), les centimètres (`cm`), pourcentage (`%`), etc.

On peut ensuite définir une marge (`margin` en anglais) autour de tout objet. La marge est une région où rien ne peut apparaître, incluant un autre objet. Mais la marge ne fait pas partie de l'objet lui-même. Ainsi, si un objet fait 10 pixels de hauteur et de largeur, et que vous définissez une marge de 10 pixels tout autour de l'objet, un espace d'une superficie de 900 pixels sera occupé sur l'écran. Voici un exemple de marge :

```
1.  p {
2.   margin-top:10px;
3.   margin-bottom:10px;
4.   margin-right:10px;
5.   margin-left:10px;
6.  }
```

Outre la marge, on peut aussi définir une région d'espacement (`padding` en anglais). Contrairement à la marge, la région d'espacement fait partie de l'objet dans la mesure où, si vous définissez une couleur de fond pour l'objet, la région d'espacement sera colorée, car elle fait partie de l'objet. Tout comme la marge, rien n'occupe cette région et elle n'est pas comptabilisée dans la hauteur et la largeur de l'objet.

```
p {
  padding-top:10px;
  padding-bottom:10px;
  padding-right:10px;
  padding-left:10px;
}
```

On peut définir une bordure (`border` en anglais) afin de tracer une ligne tout autour de l'objet. La bordure se place entre la région d'espacement et la marge. La bordure peut prendre différentes épaisseurs (telles que `thin`, `medium`, et `thick`) et aussi une couleur.

```
p {
  border-color: black;
  border-width: medium;
}
```

Il arrive qu'une image ou qu'un texte excède la taille de l'élément dans lequel il a été placé. Par défaut, ce texte ou cet image s'affichera au-delà du cadre de sa boîte (`overflow:visible`). L'instruction `overflow:hidden` permet de faire disparaître la partie d'un texte ou d'une image qui excède la boîte, alors que l'instruction `overflow:scroll` va faire apparaître des barres de défilement pour permettre à l'utilisateur d'avoir accès à tout le contenu sans pour autant avoir un dépassement.

Il arrive qu'on veuille qu'une page s'affiche différemment selon qu'on utilise un écran, qu'on l'imprime sur papier ou qu'on utilise un téléphone cellulaire. L'instruction `@media` permet de préciser quel médium est concerné par la règle CSS. Plusieurs média sont reconnus dont `handheld` (appareil portable comme un téléphone cellulaire), `screen` (à l'écran), `tv` (sur un téléviseur), `print` (sur papier), etc. Lorsque l'instruction `@media` n'est pas précisée, la règle s'applique tout le temps. Voici un exemple :

```
@media print, handheld {
  img { display: none }
  h1 {color: black }
}
@media screen, tv {
  h1 {color: blue }
}
```

Il arrive parfois qu'on veuille adopter une approche modulaire et créer plusieurs petits fichiers CSS. L'instruction `@import` apparaissant avant le début des règles, mais après une éventuelle instruction `@charset` permet d'inclure un ou plusieurs fichiers CSS. On peut aussi spécifier le type de médium concerné.

```
@import "mineure.css";
```

```
2.   @import "print-mineure.css" print;
```

On peut indenter la première ligne d'un paragraphe avec l'instruction `text-indent`. Par exemple, pour indenter la première ligne de tous les paragraphes par 1 cm, on peut utiliser l'instruction `text-indent: 1cm`.

L'instruction `text-transform` permet de modifier la casse du texte ce qui est particulièrement utile avec les titres. Par exemple, `text-transform: capitalize` modifie le texte de manière à ce que le premier caractère de chaque mot soit en majuscule. Les instructions `text-transform: uppercase` et `text-transform: lowercase` permettent de passer à une casse uniforme (majuscule ou minuscule).

Il est possible de modifier de diverses façons les polices : `font-style:italic`, `font-style:oblique`, `font-family:sans-serif`, `font-family:serif`, `font-weight:bold`, `font-weight:bolder`, `font-weight:lighter`. On peut modifier la taille des caractères : `font-size:large`, `font-size:small`. On peut ajouter des lignes sous, sur ou au-dessus le texte, ou faire clignoter le texte : `text-decoration:underline`, `text-decoration:overline`, `text-decoration:line-through`, `text-decoration:blink`. Vous pouvez aussi spécifier l'apparence que prendra le curseur avec l'attribut `cursor` qui prend la valeur `text` par défaut, et la valeur `pointer` pour les hyperliens.

On peut définir des compteurs pour, par exemple, numéroter des chapitres ou paragraphes. Ce premier exemple numérote les paragraphes (éléments `p`) contenus entre chaque élément `h1` avec des nombres romains :

```
1.   p {counter-increment: par-num}
2.   h1 {counter-reset: par-num}
3.   p:before {content: counter(par-num, upper-roman) ". "}
```

Ce second exemple numérote les éléments `h1` avec des nombres arabes :

```
1.   h1 {counter-increment: h1-num}
2.   h1:before {content: counter(h1-num, decimal) ". "}
```

Ce troisième exemple « numérote » les éléments `h1` et `h2` avec des lettres (a,b,c,...) :

```
1.   h1 {counter-reset: h2counter; counter-increment: h1counter}
2.   h2 {counter-increment: h1counter}
3.   h1:before {content: counter(h1counter, lower-latin) ". "}
4.   h2:before {
5.   content: counter(h1counter, lower-latin) "."
6.   counter(h2counter, lower-latin) ". "  }
```

Exercice. Écrivez un document CSS qui permet de donner aux lignes successives d'un tableau les couleurs « rouge, bleu et blanc ». L'entête du tableau ne doit pas recevoir de couleur.

Solution. Nous pouvons résoudre ce problème avec le pseudo-sélecteur `:nth-child`.

CodePython/chapitre14/Exercice3.html

```
1.    <!DOCTYPE html>
2.    <html>
3.    <head>
4.        <meta charset="UTF-8">
5.        <title>Exemple de page html</title>
6.        <style type>
7.            /* On utilise le pseudo-sélecteur :nth-child(n) pour cibler les lignes selon leur
position */
8.            /* On utilise la notation an+b pour indiquer un motif répétitif */
9.            /* Par exemple, :nth-child(3n+4) signifie "tous les éléments dont la position est
un multiple de 3 plus 4" */
10.
11.           /* On sélectionne les lignes dont la position est un multiple de 3 plus 1, c'est-
à-dire la première, la quatrième, la septième, etc. */
12.           tr:nth-child(3n+4) {
13.               /* On définit la couleur de fond de ces lignes à rouge */
14.               background-color: red;
15.           }
16.
17.           /* On sélectionne les lignes dont la position est un multiple de 3 plus 2, c'est-
à-dire la deuxième, la cinquième, la huitième, etc. */
18.           tr:nth-child(3n+2) {
19.               /* On définit la couleur de fond de ces lignes à bleu */
20.               background-color: blue;
21.           }
22.
23.           /* On sélectionne les lignes dont la position est un multiple de 3, c'est-à-dire
la troisième, la sixième, la neuvième, etc. */
24.           tr:nth-child(3n) {
25.               /* On définit la couleur de fond de ces lignes à blanc */
26.               background-color: white;
27.           }</style>
28.   </head>
29.   <body>
30.       <h1>Exemple de page html</h1>
31.       <table>
32.           <tr>
33.               <th>Nom</th>
34.               <th>Prénom</th>
35.               <th>Age</th>
36.           </tr>
37.           <tr>
38.               <td>Dupont</td>
39.               <td>Jean</td>
40.               <td>25</td>
41.           </tr>
42.           <tr>
43.               <td>Durand</td>
44.               <td>Marie</td>
45.               <td>30</td>
46.           </tr>
47.           <tr>
48.               <td>Martin</td>
49.               <td>Pierre</td>
50.               <td>35</td>
51.           </tr>
52.           <tr>
53.               <td>Luc</td>
54.               <td>Roger</td>
```

```
55.             <td>33</td>
56.         </tr>
57.         <tr>
58.             <td>Annie</td>
59.             <td>Marie</td>
60.             <td>31</td>
61.         </tr>
62.     </table>
63.
64.  </body>
65.  </html>
```

13.1.5 JSON

Il existe plusieurs formats de documents semi-structurés basés sur du texte (par exemple, HTML, XML, JSON). Le JSON est peut-être le format le plus populaire en ligne pour l'échange de données. Plusieurs systèmes de base de données tels que CouchDB, RethinkDB, MongoDB, SimpleDB et JSON Tiles utilisent JSON comme principal format d'échange.

Un document JSON doit être stocké dans une chaîne Unicode (UTF-8) valide. La syntaxe JSON est presque un sous-ensemble strict du langage de programmation populaire JavaScript. Elle comporte quatre types primitifs (chaîne, nombre, booléen, null) et deux types composés (tableaux et objets). Un objet se présente sous la forme d'une série de paires clé-valeur entre accolades, où les clés sont des chaînes de caractères et les valeurs des types primitifs ou composés (par exemple, `{"nom":"Jack", "âge":22}`). Un tableau est une liste de valeurs séparées par des virgules (qu'elles soient primitives ou composées) entre crochets (par exemple, `[1, "abc", null]`). La spécification JSON comporte six caractères structurels (`[`, `]`, `{`, `}`, `:`, `"`), pour délimiter l'emplacement et la structure des objets et des tableaux.

Parce qu'un objet ou un tableau peuvent, eux-mêmes, comprendre des objets et des tableaux, la syntaxe JSON permet des structures en arbre ressemblant au XML.

Voici, par exemple, les mêmes données présentées à la fois en JSON et en XML :

```
1. {
2.    "nom": "Daniel Lemire",
3.    "âge": 72,
4.    "téléphone": ["442-4321","442-4323"]
5. }
```

```
1. <personne>
2.    <nom>Daniel Lemire</nom>
3.    <age>72</age>
4.    <telephone>442-4321</telephone>
5.    <telephone>442-4323</telephone>
6. </personne>
```

Il est relativement facile de convertir un document JSON en structures de données Python :

```
1. import json
2. data = json.loads('{"nom": "Daniel Lemire", "âge": 72,
"téléphone": ["442-4321","442-4323"],}')
3. // résultat : {'nom': 'Daniel Lemire', 'âge': 72, 'téléphone':
['442-4321', '442-4323']}
```

Nous pouvons aussi générer du JSON à partir d'une structure de données Python :

```
1. import json
2. json.dumps({'nom': 'Daniel Lemire', 'âge': 72, 'téléphone':
['442-4321', '442-4323']}, ensure_ascii=False)
3. // résultat'{"nom": "Daniel Lemire", "âge": 72, "téléphone":
["442-4321", "442-4323"]}'
```

Exercice. Écrivez programme Python qui charge un document JSON intitulé exercice4.json contenant des numéros de téléphones et qui trouve le numéro de téléphone de John Galt.

```
1.  [
2.    {
3.      "name": "Alice",
4.      "phone": "+1 234 567 8901"
5.    },
6.    {
7.      "name": "Bob",
8.      "phone": "+1 345 678 9012"
9.    },
10.   {
11.     "name": "John Galt",
12.     "phone": "+1 456 789 0123"
13.   },
14.   {
15.     "name": "Eve",
16.     "phone": "+1 567 890 1234"
17.   }
18. ]
```

Solution. Notre solution est un programme Python qui fait les choses suivantes. Il importe le module `json`, qui permet de manipuler des données au format. Il ouvre le fichier `exercice4.json`, qui contient une liste de numéros de téléphone au format JSON, en mode lecture (`r`) et l'assigne à la variable `f`. Il utilise la fonction `json.load(f)` pour charger les données JSON du fichier `f` dans une variable appelée `data`. Cette variable est une liste de dictionnaires, où chaque dictionnaire représente un numéro de téléphone avec deux clés: `name` et `phone`. Il parcourt la liste de numéros de téléphone avec une boucle `for`, en utilisant la variable `item` pour accéder

à chaque dictionnaire de la liste. Il vérifie si la valeur associée à la clé `name` dans le dictionnaire `item` est égale à `John Galt` avec une condition `if`. Si c'est le cas, il affiche le numéro de téléphone de `John Galt` avec la fonction `print()`, en utilisant la valeur associée à la clé `phone` dans le dictionnaire `item`. Il utilise l'instruction `break` pour arrêter la boucle `for` dès qu'il a trouvé le numéro de `John Galt`, sans parcourir le reste de la liste.

CodePython/chapitre14/Exercice4.py

```
1.  # Importer le module json
2.  import json
3.
4.  # Ouvrir le fichier JSON qui contient la liste de numéros de
    téléphone
5.  with open("exercice4.json", "r") as f:
6.      # Charger les données JSON dans une variable
7.      data = json.load(f)
8.
9.  # Parcourir la liste de numéros de téléphone
10. for item in data:
11.     # Vérifier si le nom correspond à John Galt
12.     if item["name"] == "John Galt":
13.         # Afficher le numéro de téléphone de John Galt
14.         print("Le numéro de téléphone de John Galt est:",
    item["phone"])
15.         # Arrêter la boucle
16.         break
```

13.1.6 JavaScript

Tout comme Python, le JavaScript est un langage conçu pour être accessible. Il comporte un nombre limité de types de base : les nombres, les valeurs booléennes, les fonctions, les objets, les tableaux, etc.

En JavaScript, il est possible d'afficher des informations dans la console du navigateur. Pour cela, il faut utiliser la fonction `console.log()`. Cette fonction prend en paramètre la valeur à afficher dans la console. Dans un navigateur, la console est accessible via le menu Outils de développement. Il faut consulter la documentation de votre navigateur.

En JavaScript, on peut déclarer de nouvelles variables grâce à l'un de ces trois mots-clés : `let`, `const`, ou `var`. Le mot-clé let permet de déclarer des variables qui pourront être utilisées dans un bloc. La variable déclarée avec let est uniquement disponible dans le bloc qui contient la déclaration.

```
1. for (let i = 0; i < 5; i++) {
2.   // i peut être utilisée ici
3. }
4.
```

```
5. // i n'est pas utilisable ici
```

Le mot-clé `const` permet de déclarer des variables dont la valeur ne doit pas changer. Une variable déclarée avec `const` est disponible dans le bloc dans lequel elle est déclarée.

Le mot-clé `var` est le mot-clé le plus fréquemment utilisé pour déclarer des variables. Ce mot-clé était disponible avant `let` et `const`. Une variable qu'on déclare avec `var` est disponible dans la fonction dans laquelle elle est déclarée.

```
1. for (var i = 0; i < 5; i++) {
2.   // i *est* également disponible ici
3. }
4.
5. // i *est* toujours disponible ici
```

Il y a plusieurs manières de définir une chaîne de caractères en JavaScript incluant les guillemets droits (simples ou doubles), comme en Python. Le caractère ` en JavaScript sert à définir une chaîne de caractères qui peut contenir des expressions ou des variables à l'intérieur. On appelle cela une chaîne de caractères littérale étendue (ou *template literal* en anglais). Par exemple:

```
1.  const nom = "Alice";
2.  const age = 25;
3.  const message = `Bonjour, je m'appelle ${nom} et j'ai ${age}
    ans.`;
4.  console.log(message); // Affiche "Bonjour, je m'appelle Alice
    et j'ai 25 ans."
```

Dans cet exemple, la chaîne de caractères littérale étendue est délimitée par les caractères ` et contient deux expressions entre `${` et `}`. Ces expressions sont évaluées et insérées dans la chaîne de caractères au moment de son exécution. Les chaînes de caractères littérales étendues peuvent aussi contenir des sauts de ligne, des caractères d'échappement et des fonctions de mise en forme.

Les objets en JavaScript sont des collections de paires clé-valeur. Ils sont similaires aux dictionnaires en Python.

```
1. let objet = {
2.   nom: "Daniel",
3.   age: 12,
4. };
5. console.log(objet.nom); // Affiche "Daniel"
6. console.log(objet.age); // Affiche 12
7. console.log(objet["nom"]); // Affiche "Daniel"
8. objet.nom = "Lucie";
```

On peut facilement créer de nouveaux tableaux en JavaScript avec les crochets `[]`.

```
1. let a = ["Daniel", "Jacques"];
2. a.length; // 2
```

Il y a deux manières de traverser un tableau en JavaScript avec des boucles `for`, selon que nous avons besoin de l'index…

```
1. for (let i = 0; i < a.length; i++) {
2.   // Faire quelque chose avec a[i]
3. }
```

Ou que nous n'en avons pas besoin…

```
1. for (const i of a) {
2.   // Faire quelque chose avec i
3. }
```

Un tableau JavaScript peut être convertit en chaîne de caractères avec l'attribut `toString : a.toString()`.

On définit des fonctions avec le mot-clé `function`. On peut ensuite appeler la fonction en utilisant son nom suivi de parenthèses.

```
1. function moyenne(a, b) {
2.   return (a + b) / 2;
3. }
4. moyenne(2, 3); // 2.5
```

On peut aussi définir des fonctions anonymes. Une fonction anonyme est une fonction qui n'a pas de nom. On peut par exemple stocker une fonction anonyme dans une variable.

```
1. let moyenne = function (a, b) {
2.   return (a + b) / 2;
3. };
```

On peut aussi définir des fonctions fléchées. Une fonction fléchée est une fonction anonyme plus courte. On utilise le symbole `=>` pour définir une fonction fléchée.

```
1. let moyenne = (a, b) => {
2.   return (a + b) / 2;
3. };
```

Si a est un tableau, on peut utiliser la méthode `forEach` pour exécuter une fonction sur chaque élément du tableau.

```
1. a.forEach((element) => { element + 1; });
```

On peut aussi créer un nouveau tableau avec `Arrays.from()`.

```
1. Arrays.from([1, 2, 3], (x) => x + x); // [2, 4, 6]
```

Exercice. Écrivez programme Javascript qui traite une chaîne de caractères comprenant un document JSON contenant des numéros de téléphones et qui trouve le numéro de téléphone de John Galt. Votre programme doit débuter comme ceci :

```
1.  const data = `[
2.    {
3.      "name": "Alice",
4.      "phone": "+1 234 567 8901"
5.    },
6.    {
7.      "name": "Bob",
8.      "phone": "+1 345 678 9012"
9.    },
10.   {
11.     "name": "John Galt",
12.     "phone": "+1 456 789 0123"
13.   },
14.   {
15.     "name": "Eve",
16.     "phone": "+1 567 890 1234"
17.   }
18. ]`;
```

Solution. Notre solution est un programme JavaScript qui fait les choses suivantes. Il définit une variable data qui contient une chaîne de caractères au format. Cette chaîne représente une liste de numéros de téléphone, où chaque numéro est associé à un nom. Il utilise la fonction `JSON.parse(data)` pour convertir la chaîne JSON en un objet Javascript, qui est une structure de données qui peut contenir des valeurs de différents types. Cette fonction renvoie une liste d'objets, où chaque objet représente un numéro de téléphone avec deux propriétés: `name` et `phone`. Il assigne cette liste d'objets à une variable appelée `phoneList`. Il parcourt la liste de numéros de téléphone avec une boucle `for...of`, qui permet d'itérer sur les éléments d'un objet itérable (comme une liste). Il utilise la variable `item` pour accéder à chaque objet de la liste. Il vérifie si la valeur de la propriété `name` de l'objet item est égale à `John Galt` avec une condition `if`. Si c'est le cas, il affiche le numéro de téléphone de `John Galt` avec la fonction `console.log()`, en utilisant la valeur de la propriété `phone` de l'objet `item`. Il utilise l'instruction `break` pour arrêter la boucle `for` dès qu'il a trouvé le numéro de `John Galt`, sans parcourir le reste de la liste.

CodePython/chapitre14/Exercice5.js

```
const data = `[
  {
    "name": "Alice",
    "phone": "+1 234 567 8901"
  },
  {
    "name": "Bob",
    "phone": "+1 345 678 9012"
  },
  {
    "name": "John Galt",
    "phone": "+1 456 789 0123"
  },
  {
    "name": "Eve",
    "phone": "+1 567 890 1234"
  }
]`;
// Convertir les données JSON en un objet Javascript
const phoneList = JSON.parse(data);

// Parcourir la liste de numéros de téléphone avec une boucle for...of
for (const item of phoneList) {
  // Vérifier si le nom correspond à John Galt
  if (item.name === "John Galt") {
    // Afficher le numéro de téléphone de John Galt
    console.log("Le numéro de téléphone de John Galt est:", item.phone);
    // Arrêter la boucle
    break;
  }
}
```

13.2 Environnements virtuels

En installant Python, vous avez déjà plusieurs modules par défaut, mais il est facile d'ajouter de nouveaux modules et librairies spécialisés avec Python. C'est d'ailleurs quasiment essentiel si on souhaite développer des applications.

La command `pip` permet d'installer de nouveaux modules ou librairies. Alors que vous êtes en ligne de commande (mais pas au sein de l'interprète Python), tapez la commande qui suit qui va installer au sein de l'environnement Python global le module `fastrand` :

```
$ python -m pip install fastrand
```

Le module `fastrand` permet de générer rapidement des nombres aléatoires. Lancez ensuite l'interprète Python et entrez les commandes suivantes :

```
>>> import fastrand
>>> [fastrand.pcg32bounded(100) for i in range(10)]
[97, 11, 64, 39, 3, 71, 90, 74, 27, 82]
```

En tapant `python -m pip freeze` ou `python -m pip list`, vous pourrez voir tous les modules installés sur votre machine. Chaque module comprend un nom et aussi une de version. Les modules évoluent dans le temps et il est possible qu'un programme qui fonctionne bien avec une version d'un module, ne fonctionne plus si la version du module diffère.

La plupart du temps, une application Web s'exécute sur un serveur qui est distinct de l'ordinateur sur lequel le programmeur travaille. Le programmeur doit déployer l'application sur le serveur pour la tester. Cependant, cela peut être un processus fastidieux et prendre beaucoup de temps s'il faut manuellement réinstaller les modules, et s'assurer d'avoir la bonne version de chaque module. Pour cette raison, les programmeurs utilisent souvent un serveur de développement local pour tester leur application avant de la déployer sur un serveur distant. Un serveur de développement local est un serveur Web qui s'exécute sur l'ordinateur du programmeur et qui permet au programmeur de tester l'application Web sans avoir à la déployer sur un serveur distant.

Une application Web écrite en Python utilisera souvent des librairies. Il est possible pour un programmeur de développer plusieurs applications Web distinctes sur un même ordinateur, et chaque application Web peut dépendre de librairies différentes. Parfois, deux applications peuvent utiliser la même librairie, mais viser deux versions différentes de la librairie en question. Pour cette raison, il est important de pouvoir isoler les librairies utilisées par une application Web afin qu'elles ne puissent pas interférer avec les librairies utilisées par d'autres applications sur le même ordinateur. Les environnements virtuels permettent d'obtenir cette isolation. Un environnement virtuel est un environnement Python isolé qui permet aux développeurs de travailler sur des projets sans interférer avec les autres projets ou avec l'installation Python globale. Les environnements virtuels sont utiles pour les projets qui ont des dépendances spécifiques à une version de bibliothèque ou de module Python, car ils permettent aux développeurs de travailler avec différentes versions de Python et de bibliothèques sans conflit. Les environnements virtuels sont également utiles pour les projets qui doivent être exécutés sur différents systèmes, car ils permettent de nous aider à faire en sorte que toutes les dépendances sont installées correctement.

Les programmeurs utilisent parfois l'outil `venv` en Python pour créer des environnements virtuels. Il existe d'autres outils qui peuvent être utilisés pour créer des environnements virtuels. Dans ce cours, nous allons utiliser ce qui est disponible par défaut en Python (à compter de Python 3). Pour créer un environnement virtuel, il faut d'abord créer un répertoire pour le projet. Ensuite, un environnement virtuel peut être créé dans ce répertoire.

Pour créer un environnement virtuel, vous pouvez utiliser le module venv intégré à Python. Le module `venv` prend en charge la création d'environnements virtuels légers, chacun ayant son propre ensemble indépendant de packages Python installés dans leurs répertoires de site. Un

environnement virtuel est créé au-dessus d'une installation Python existante, connue sous le nom de Python de base de l'environnement virtuel, et peut éventuellement être isolé des packages dans l'environnement de base, de sorte que seuls ceux explicitement installés dans l'environnement virtuel sont disponibles. Lorsqu'ils sont utilisés à partir d'un environnement virtuel, des outils d'installation courants tels que `pip` installeront des packages Python dans un environnement virtuel sans avoir besoin d'être explicitement informés de le faire. Voici les étapes pour créer un environnement virtuel :

1. Ouvrez une invite de commande ou un terminal.

2. Accédez à un répertoire vierge. Ce répertoire vous servira de répertoire de travail pour votre projet.

3. Exécutez la commande `python -m venv mon_env` pour créer un nouvel environnement virtuel nommé `mon_env`. Remplacez `mon_env` par le nom que vous souhaitez donner à votre environnement. Cette commande va créer un répertoire `mon_env` qui contient un environnement virtuel Python. Au sein de ce répertoire, il y aura un répertoire appelé `bin` sur Linux ou Mac, ou `Scripts` sur Windows, qui contient un script `activate`. Ce script active l'environnement virtuel. On y trouve aussi d'autres fichiers dont le fichier `pyvenv.cfg` qui contient des informations sur l'environnement virtuel.

4. Activez l'environnement virtuel en exécutant la commande `source mon_env /bin/activate` sur Linux ou Mac, ou `mon_env\Scripts\activate.bat` sur Windows.

Vous pouvez maintenant installer les librairies Python dont vous avez besoin pour votre projet dans l'environnement virtuel.

Par exemple, pour créer un environnement virtuel appelé mon_env, exécutez la commande suivante :

```
python -m venv mon_env
```

Ensuite, pour activer l'environnement virtuel, vous pouvez exécuter la commande suivante sur Linux ou Mac :

```
source mon_env/bin/activate
```

Ou la commande suivante sur Windows:

```
mon_env\Scripts\activate.bat
```

Il est possible de quitter en tout temps l'environnement virtuel en exécutant la commande `deactivate`. Pour revenir ensuite dans l'environnement virtuel, exécutez la commande `source mon_env/bin/activate` sur Linux ou Mac, ou `mon_env\Scripts\activate.bat` sur Windows.

Au sein de votre éditeur de texte (Visual Studio Code), l'environnement virtuel apparaît comme un répertoire normal où vous pouvez créer des fichiers Python ou d'autres fichiers et répertoires. Néanmoins, il est recommandé de ne rien créer dans le répertoire de l'environnement virtuel. Placez plutôt vos fichiers et répertoires dans le répertoire-parent de l'environnement virtuel, celui qui vous servira de répertoire de travail pour votre projet.

Un fois au sein de l'environnement virtuel, vous pouvez ajouter des librairies Python en utilisant la commande `pip install nom_librairie`. Par exemple, si vous souhaitez installer la librairie `Flask`, vous pouvez exécuter la commande suivante :

```
python -m pip install flask
```

Pour vérifier que `flask` a été installé, lancez la console Python (tapez `python`) et exécutez l'instruction Python suivante :

```
>> import flask
```

Si vous ne recevez pas d'erreur, c'est que la librairie a été installée correctement.

Quittez maintenant la console Python en tapant `exit()`. Puis quittez l'environnement virtuel (`deactivate`). Si vous entrez à nouveau dans la console Python et que vous tapez à nouveau

```
>> import flask
```

Vous devriez recevoir une erreur puisque la librairie `flask` n'est pas installée en dehors de l'environnement virtuel. (Il est possible qu'il n'y ait pas d'erreur si vous aviez précédemment installé `flask` sur votre ordinateur.)

Retournez dans l'environnement virtuel (`source mon_env/bin/activate` sur Linux ou Mac, ou `mon_env\Scripts\activate.bat` sur Windows) et exécutez la commande `pip freeze`. Vous devriez voir la librairie `flask` dans la liste des librairies installées.

Alors que vous êtes dans votre répertoire de travail (en dehors du répertoire de l'environnement virtuel, `mon_env`), vous pouvez créer un fichier Python et l'exécuter. Par exemple, vous pouvez créer un fichier `projet.py` contenant le code suivant :

```
 1. import pysimdjson
 2. from roaringbitmap import RoaringBitmap
 3.
 4. # Créer un objet RoaringBitmap
 5. rb = RoaringBitmap()
 6.
 7. # Ajouter des éléments à l'objet RoaringBitmap
 8. rb.add(1)
 9. rb.add(2)
10. rb.add(3)
11.
12. # Convertir l'objet RoaringBitmap en JSON
13. json_data = rb.to_json()
14.
15. # Analyser le JSON avec pysimdjson
16. data = pysimdjson.loads(json_data)
17.
18. # Afficher les éléments de l'objet RoaringBitmap
19. for i in data:
20.     print(i)
```

Si vous exécutez le programme (`python projet.py`), vous devriez recevoir un message d'erreur puisque les librairies `roaringbitmap` et `pysimdjson` ne sont probablement pas présentes sur votre ordinateur. Pour installer ces librairies, vous pouvez exécuter la commande

```
python -m pip install roaringbitmap pysimdjson
```

Si vous exécutez maintenant le programme (`python projet.py`), vous devriez voir s'afficher les éléments 1, 2 et 3. Si vous exécutez la commande `pip freeze`, vous devriez voir que les librairies `pysimdjson` et `roaringbitmap` sont installées. Vous devriez voir quelque chose comme ceci :

```
Flask==3.0.0
pysimdjson==5.0.2
roaringbitmap==0.7.2
```

Si vous souhaitez installer toutes les librairies d'un projet, vous pouvez créer un fichier `requirements.txt` contenant la liste des librairies à installer. Pour générer un fichier `requirements.txt` à l'aide de `pip`, vous pouvez utiliser la commande `python -m pip freeze > requirements.txt`. Cette commande enregistre toutes les bibliothèques Python installées avec leur version actuelle dans un fichier requirements.txt. Vous pouvez ensuite

utiliser ce fichier pour installer les mêmes bibliothèques sur un autre système ou pour partager votre environnement de développement avec d'autres. Vous pouvez installer toutes les librairies en exécutant la commande `python -m pip install -r requirements.txt`.

13.3 Utilisation de Flask

Le module `Flask` utilise la variable `__name__`. La variable `__name__` est une variable spéciale en Python qui contient le nom du module courant. Cette variable est automatiquement définie par l'interprète Python et peut être utilisée pour déterminer si un module est exécuté en tant que programme principal ou s'il est importé en tant que module dans un autre programme. Lorsqu'un module est exécuté en tant que programme principal, la variable `__name__` est définie sur la chaîne de caractères `"__main__"`. Cependant, si le module est importé dans un autre programme, la variable `__name__` est définie sur le nom du module. La variable `__name__` est souvent utilisée pour encapsuler le code qui ne doit être exécuté que si le module est exécuté en tant que programme principal. Par exemple, si vous avez un fichier `mon_module.py` contenant des fonctions et du code, vous pouvez ajouter le code suivant à la fin du fichier pour exécuter une fonction uniquement si le fichier est exécuté en tant que programme principal :

```
1. if __name__ == "__main__":
2.     ma_fonction()
```

Cela garantit que la fonction `ma_fonction()` n'est exécutée que si le fichier `mon_module.py` est exécuté en tant que programme principal. Le module `Flask` utilise la variable `__name__` pour identifier les ressources telles que les modèles, les fichiers statiques et le dossier d'instance. Lorsque vous créez une instance de l'objet `Flask`, vous passez le nom du module courant à la variable `__name__`. `Flask` utilise ensuite cette variable pour identifier les ressources nécessaires à l'exécution de l'application.

Flask est une infrastructure logiciel (ou *framework*) Web flexible pour Python qui permet de créer rapidement des applications Web. Pour créer un serveur HTTP avec Flask, il suffit d'associer chaque route à une fonction qui renvoie le contenu à afficher. Vous pouvez également définir des paramètres pour chaque route, tels que les méthodes HTTP acceptées, les types de données acceptés et les types de données renvoyés. Voici quelques exemples de routes Flask :

- `/home`: associé à la fonction `home()` qui renvoie le contenu de la page d'accueil.

- `/login`: associé à la fonction `login()` qui gère la connexion de l'utilisateur.

- `/logout`: associé à la fonction `logout()` qui gère la déconnexion de l'utilisateur.

- `/user/<username>` : associé à la fonction `user(username)` qui renvoie le profil de l'utilisateur spécifié par `<username>`. Le nom d'utilisateur est un paramètre de chemin et peut être utilisé pour personnaliser le contenu renvoyé par la fonction associée à la route.

Si vous avez inclus le module `Flask` dans votre environnement (`python -m pip install flask`), vous pouvez dès maintenant créer une application Web. Il vous suffit de créer un fichier nommé `server.py` contenant le code suivant :

```
1. from flask import Flask
2. app = Flask(__name__)
3. @app.route('/')
4. def hello_world():
5.     return '<html><body>Hello World!</body></html>'
6.
7. app.run()
```

La première ligne `from flask import Flask` importe la classe `Flask` du module `flask`. La deuxième ligne crée une instance de la classe `Flask` et la stocke dans la variable `app`. Le paramètre __name__ est utilisé pour déterminer le nom de l'application.

La troisième ligne définit une route pour l'URL racine `/` en utilisant le décorateur `@app.route('/')`. Le décorateur est une fonction qui prend une autre fonction en argument et renvoie une nouvelle fonction. Dans ce cas, la nouvelle fonction est `hello_world()`.

La quatrième ligne définit la fonction `hello_world()`, qui renvoie une chaîne de caractères HTML contenant le texte `Hello World!`.

Enfin, la dernière ligne `app.run()` démarre l'application Web. Cela lance un serveur Web local qui écoute les connexions entrantes sur le port 5000 par défaut. L'application peut être accédée en ouvrant un navigateur Web et en visitant l'URL http://localhost:5000/.

Après avoir tapé python server.py, vous devriez avoir un message comme celui-ci :

```
* Serving Flask app 'server'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production
deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

Dans ce cas, Flask a choisi le port 5000. Vous pouvez aussi spécifier un port avec le paramètre port de la fonction run :

```
1. if __name__ == '__main__':
2.     app.run(port=5000)
```

Le message d'avertissement concernant les « WSGI servers » peut être ignoré.

Si vous ouvrez votre navigateur à l'adresse http://127.0.0.1:5000. Prenez soin de remplacer 5000 par un port différent si Flask utilise un port différent. Vous devriez voir une page HTML. Pour mettre fin à l'application Web de type Flask, vous pouvez presser les touches `CTRL-C`.

Pour renvoyer du XML à un client à l'aide de Flask, vous pouvez utiliser la classe `Response` de Flask pour créer une réponse HTTP personnalisée.

Voici un exemple de code qui renvoie une réponse XML à un client:

```
 1. from flask import Flask, Response
 2.
 3. app = Flask(__name__)
 4.
 5. @app.route('/xml')
 6. def xml_response():
 7.     xml_data = '<example>Hello, World!</example>'
 8.     return Response(xml_data, mimetype='text/xml')
 9.
10. if __name__ == '__main__':
11.     app.run()
```

Dans cet exemple, nous avons créé une route `/xml` qui renvoie une réponse XML. La réponse est créée en utilisant la classe `Response` de Flask et en spécifiant le contenu XML et le type MIME `text/xml`. Si vous enregistrer ce programme sous le nom serverxml.py et que vous lancez le serveur (après avoir mis fin au serveur précédent), vous pourrez avoir accès à la ressource http://127.0.0.1:5000/xml. Dans un navigateur, il est possible que vous puissiez voir le document XML.

Voici un exemple de serveur Flask qui renvoie une réponse JSON :

```
 1. from flask import Flask, json
 2.
 3. app = Flask(__name__)
 4.
 5. @app.route('/json')
 6. def example():
 7.     data = {'name': 'John', 'age': 30}
 8.     return json.dumps(data)
 9.
10. if __name__ == '__main__':
11.     app.run()
```

Dans cet exemple, nous avons créé une route `/json` qui renvoie une réponse JSON. Vous pouvez enregistrer ce code dans un fichier nommé `serverjson.py`. Après avoir mis fin aux autres serveurs, vous pouvez l'exécuter et avoir accès (dans votre navigateur) à l'adresse http://127.0.0.1:5000/json. Il est alors possible que vous puissiez voir le document JSON.

Vous pouvez combiner ces routes au sein d'une même application :

CodePython/chapitre14/virtuel/servercombine.py

```
 1. from flask import Flask, Response
 2.
 3. app = Flask(__name__)
 4.
 5. @app.route('/xml')
 6. def xml_response():
 7.     xml_data = '<example>Hello, World!</example>'
 8.     return Response(xml_data, mimetype='text/xml')
 9.
10. @app.route('/json')
11. def example():
12.     data = {'name': 'John', 'age': 30}
13.     return json.dumps(data)
14.
15. @app.route('/')
16. def hello_world():
17.     return '<html><body>Hello World!</body></html>'
18.
19. app.run()
```

L'exemple suivant illustre comment une route peut être dynamique, c'est-à-dire qu'elle peut correspondre à de multiples URL :

```
1. import Flask
2. app = Flask(__name__)
```

```
@app.route('/utilisateur/<nom>')
def show_user_profile(nom):
    # Affiche le profil de l'utilisateur spécifié par <nom>
    return '<html><body>Bonjour %s</body></html>' % nom

app.run()
```

Si vous lancez ce serveur et ouvrez la page `http://127.0.0.1:5000/utilisateur/daniel` dans votre navigateur, vous devriez voir un message `Bonjour Daniel`.

Il est parfois pratique de rediriger automatiquement les utilisateurs d'un URL à un autre. Il est possible d'obtenir ce résultat avec Flask en utilisant la fonction `redirect()` :

```
from flask import redirect
@app.route('/')
def index():
    return redirect('/upload')
```

Lorsqu'un utilisateur visite l'URL `/`, il sera automatiquement redirigé vers l'URL `/upload`.

Exercice. Écrivez programme Python avec Flask qui retourne la date et l'heure.

Solution. Note code est un programme Python qui utilise le module Flask. Il crée une instance de l'application Flask avec la commande `app = Flask(__name__)`, en utilisant le paramètre `__name__` qui représente le nom du module courant. Il définit la route principale de l'application web avec le décorateur `@app.route("/")`, qui associe la fonction index à l'URL racine de l'application. Il définit la fonction index, qui sera exécutée quand un utilisateur accède à la route principale. Il importe le module `datetime` de la bibliothèque standard Python, qui permet de manipuler des dates et des heures. Il importe le module `locale`, qui permet de gérer les paramètres régionaux, comme la langue, le format de la date, etc. Il définit le paramètre régional à `fr_CA` pour le français du Canada avec la commande `locale.setlocale(locale.LC_TIME, "fr_CA")`, en utilisant la constante `locale.LC_TIME` qui indique le paramètre de la date et de l'heure. Il obtient la date et l'heure actuelles avec la commande `now = datetime.now()`, qui renvoie un objet `datetime`. Il formate la date et l'heure en français avec la méthode `strftime()`, qui convertit un objet `datetime` en une chaîne de caractères selon un format spécifié. Il utilise les directives suivantes: `%A` pour le nom du jour de la semaine en toutes lettres, `%d` pour le jour du mois en chiffres, `%B` pour le nom du mois en toutes lettres, `%Y` pour l'année en quatre chiffres, `%H` pour l'heure en format

24 heures, `%M` pour les minutes, `%S` pour les secondes. Il assigne la date et l'heure formatées à deux variables : `date` et `heure`. Il retourne une page Web avec la date et l'heure avec la commande `return f"""..."""`, qui utilise une chaîne de caractères littérale étendue (ou *template literal* en anglais) pour insérer les variables `date` et `heure` dans le code HTML. Il lance l'application Flask avec la commande `app.run()`, qui démarre un serveur web local pour héberger l'application. Il utilise la condition `if __name__ == "__main__"` pour s'assurer que cette commande n'est exécutée que si le programme est lancé directement, et non pas importé comme un module.

CodePython/chapitre14/Exercice6.py

```
1.  # Importer le module flask
2.  from flask import Flask
3.
4.  # Créer une instance de l'application Flask
5.  app = Flask(__name__)
6.
7.  # Définir la route principale
8.  @app.route("/")
9.  def index():
10.     # Importer le module datetime
11.     from datetime import datetime
12.
13.     # Importer le module locale, qui permet de gérer les paramètres régionaux
14.     import locale
15.
16.     # Définir le paramètre régional à "fr_CA" pour le français du Canada
17.     locale.setlocale(locale.LC_TIME, "fr_CA")
18.
19.     # Obtenir la date et l'heure
20.     now = datetime.now()
21.
22.     # Formater la date et l'heure en français avec la méthode strftime
23.     date = now.strftime("%A %d %B %Y")
24.     heure = now.strftime("%H:%M:%S")
25.
26.     # Retourner une page web avec la date et l'heure
27.     return f"""
28.     <html>
29.         <head>
30.             <title>Date et heure</title>
31.         </head>
32.         <body>
33.             <h1>Bonjour, voici la date et l'heure:</h1>
34.             <p>Date: {date}</p>
35.             <p>Heure: {heure}</p>
36.         </body>
37.     </html>
38.     """
39.
40. # Lancer l'application Flask
41. if __name__ == "__main__":
42.     app.run()
```

Il peut être un peu lourd de toujours utiliser du code Python pour générer du HTML. Les développeurs inversent souvent le développement : au lieu d'écrire du HTML au sein de Python, on peut aussi mettre (un peu) de Python au sein du HTML. Ou tout simplement, le développeur peut utiliser du HTML pur au sein de son application écrite en Python. À cette fin, on utilise le répertoire `templates`. Le répertoire `templates` dans Flask est l'endroit où vous stockez les fichiers de modèle pour votre application

Web. Les fichiers de modèle sont des fichiers HTML qui contiennent des variables et des instructions de contrôle de flux qui sont remplacées par des valeurs dynamiques lorsqu'un utilisateur visite une page Web. Les fichiers de modèle doivent être stockés dans le répertoire `templates` pour que Flask puisse les trouver automatiquement. Vous pouvez aussi personnaliser l'emplacement du répertoire de modèles en utilisant le paramètre `template_folder` lors de la création d'une instance d'application Flask. Pour spécifier le répertoire de modèles dans Flask, vous pouvez utiliser le paramètre `template_folder` lors de la création d'une instance d'application Flask. Par exemple, pour définir le répertoire de modèles sur `/path/to/templates`, vous pouvez utiliser le code suivant :

```
1. app = Flask(__name__, template_folder='/path/to/templates')
```

Cela indique à Flask de chercher des fichiers de modèle dans le répertoire `/path/to/templates` plutôt que dans le répertoire par défaut `templates`. Vous pouvez également utiliser une variable pour stocker le chemin du répertoire de modèles et l'utiliser lors de la création de l'instance d'application Flask :

```
1. TEMPLATE_DIR = '/path/to/templates'
2. app = Flask(__name__, template_folder=TEMPLATE_DIR)
```

Nous allons développer une application Web qui accepte des fichiers d'images JPEG et qui retourne à l'utilisateur la position géographique où la photo a été prise. Cette information est généralement présente à même le fichier au format EXIF. EXIF (*Exchangeable Image File Format*) est un format de métadonnées standard pour les fichiers image numériques. Les données EXIF sont stockées dans les fichiers JPEG et contiennent des informations sur les paramètres de prise de vue, tels que la date et l'heure de la prise de vue, la marque et le modèle de l'appareil photo, les paramètres d'exposition, la distance focale, la vitesse d'obturation, l'ouverture et plus encore. Les données EXIF sont stockées dans un segment d'application défini par JPEG. Les données EXIF peuvent être lues et modifiées par des logiciels de traitement d'image tels que Photoshop ou Lightroom.

Le module `exifread` permet de lire des données EXIF. IL faut l'installer en utilisant `pip` avec la commande `python -m pip install exifread`. Une fois `exifread` installé, il peut être utilisé pour extraire les métadonnées EXIF des fichiers image numériques. Voici un exemple :

```
1. import exifread
2. # Ouvrir le fichier image pour la lecture (mode binaire)
3. f = open('img.jpg', 'rb')
4.
5. # Extraire les tags EXIF
6. tags = exifread.process_file(f)
```

```
 7.
 8. # Parcourir tous les tags et afficher les valeurs
 9. for tag in tags.keys():
10.     if tag not in ('JPEGThumbnail', 'TIFFThumbnail', 'Filename', 'EXIF MakerNote'):
11.         print("Key: %s, value %s" % (tag, tags[tag]))
```

Dans cet exemple, le fichier image `img.jpg` est ouvert en mode binaire et `exifread.process_file()` extrait les `tags` EXIF du fichier. Les `tags` sont parcourus pour en afficher leurs valeurs. La fonction `get_exif_data()` récupère toutes les métadonnées :

```
1. def get_exif_data(image_file):
2.     with open(image_file, 'rb') as f:
3.         exif_tags = exifread.process_file(f)
4.     return exif_tags
```

Toutes les valeurs ne sont pas pertinentes ou présentes. Pour éviter d'avoir à gérer des exceptions, il est pratique d'avoir une fonction qui retourne la valeur `None` lorsqu'une valeur est manquante :

```
1. def get_if_exist(data, key):
2.     if key in data:
3.         return data[key]
4.     return None
```

Les valeurs EXIF suivantes sont des informations de géolocalisation stockées dans les fichiers image numériques:

- `GPSLatitude` : la latitude de l'emplacement où la photo a été prise, stockée sous forme de tableau de trois nombres à virgule flottante. Par exemple, si la latitude est 37 degrés, 48,657 minutes nord, la valeur serait stockée comme (37.0, 48.657, 0.0).

- `GPSLatitudeRef` : un indicateur de la direction de la latitude. Il peut être soit `N` pour nord, soit `S` pour sud.

- `GPSLongitude` : la longitude de l'emplacement où la photo a été prise, stockée sous forme de tableau de trois nombres à virgule flottante. Par exemple, si la longitude est 122 degrés, 25,424 minutes ouest, la valeur serait stockée comme `(122.0, 25.424, 0.0)`.

- `GPSLongitudeRef` : un indicateur de la direction de la longitude. Il peut être soit `E` pour est, soit `W` pour ouest.

Nous souhaitons convertir les valeurs de ces paramètres en valeur de longitude et de latitude que nous pouvons passer à une application de

cartographie Web comme Google Map, Apple Plan, OpenStreetMap, etc. Nous pouvons y arriver avec le code suivant :

```
def convert_to_degrees(value):
    d = float(value.values[0].num) / float(value.values[0].den)
    m = float(value.values[1].num) / float(value.values[1].den)
    s = float(value.values[2].num) / float(value.values[2].den)
    return d + (m / 60.0) + (s / 3600.0)

def get_exif_location(exif_data):
    lat = None
    lon = None

    gps_latitude = get_if_exist(exif_data, 'GPS GPSLatitude')
    gps_latitude_ref = get_if_exist(exif_data, 'GPS GPSLatitudeRef')
    gps_longitude = get_if_exist(exif_data, 'GPS GPSLongitude')
    gps_longitude_ref = get_if_exist(exif_data, 'GPS GPSLongitudeRef')

    if gps_latitude and gps_latitude_ref and gps_longitude and gps_longitude_ref:
        lat = convert_to_degrees(gps_latitude)
        if gps_latitude_ref.values[0] != 'N':
            lat = 0 - lat

        lon = convert_to_degrees(gps_longitude)
        if gps_longitude_ref.values[0] != 'E':
            lon = 0 - lon

    return lat, lon
```

Ce code Python définit deux fonctions qui permettent d'extraire les informations de géolocalisation des fichiers image numériques. La fonction `convert_to_degrees(value)` convertit les coordonnées GPS stockées dans le format DMS (degrés, minutes, secondes) en degrés décimaux. La fonction `get_exif_location(exif_data)` extrait les informations de géolocalisation à partir des données EXIF d'un fichier image.

La fonction `convert_to_degrees(value)` prend un paramètre `value` qui est un tableau de trois nombres à virgule flottante représentant les coordonnées GPS stockées dans le format DMS. La fonction convertit ces coordonnées en degrés décimaux en utilisant la formule suivante :

```
degrees + minutes / 60 + seconds / 3600
```

La fonction `get_exif_location(exif_data)` prend un paramètre `exif_data` qui est un dictionnaire contenant les données EXIF d'un fichier image. La fonction extrait les informations de géolocalisation en recherchant les `tags` EXIF correspondants (`GPS GPSLatitude, GPS GPSLatitudeRef, GPS GPSLongitude, et GPS GPSLongitudeRef`) et en utilisant la fonction `convert_to_degrees()` pour convertir les coordonnées GPS en degrés décimaux.

La fonction renvoie une paire de valeurs `(lat, lon)` représentant la latitude et la longitude de l'emplacement où la photo a été prise. Si les informations de géolocalisation ne sont pas disponibles dans les données EXIF du fichier, la fonction renvoie `(None, None)`.

Créons maintenant un formulaire Web où l'utilisateur pourra déposer sa photo. Au lieu de faire un formulaire au HTML au sein de notre programme Python, un fichier HTML est employé le répertoire `templates` par défaut. Créez un répertoire `templates` au sein de votre répertoire de travail, et placez-y le fichier `formulaire.html` suivant :

```
<html>
   <body>
      <form action = "{{ url_for('upload_file') }}" method = "POST"
         enctype = "multipart/form-data">
         <input type = "file" name = "file" />
         <input type = "submit"/>
      </form>
      <p>{{ message }}</p>
   </body>
</html>
```

Il s'agit quasiment de HTML pur, à l'exception de la présence de séquences débutant par `{{` et se terminant par `}}`. Ces séquences doivent contenir du code qui serait traité par le serveur Flask.

Le code HTML définit un formulaire qui permet à l'utilisateur de télécharger un fichier sur un serveur Web. Le formulaire utilise la méthode HTTP POST pour envoyer les données et l'attribut `enctype` est défini sur `multipart/form-data` pour permettre l'envoi de fichiers binaires. Le formulaire contient un champ de fichier `<input type="file" name="file" />` qui permet à l'utilisateur de sélectionner le fichier à télécharger, ainsi qu'un bouton de soumission `<input type="submit" />` qui envoie le formulaire au serveur Web.

L'attribut action du formulaire spécifie l'URL du script côté serveur qui gère le téléchargement du fichier. Dans cet exemple, l'URL correspond à la fonction `upload_file()` et à sa route. La séquence `{{ message }}` permet d'inclure dans le rendu HTML le contenu de la variable `message`. En Flask, les doubles accolades `{{ }}` servent à afficher des variables dans un `template`. Par exemple, la variable `nom` peut être affichée dans un `template` avec `{{ nom }}`. Les doubles accolades sont utilisées pour les variables qui doivent être affichées à l'utilisateur. Les doubles accolades permettent aussid' afficher des expressions Python comme `{{ nom.upper() }}` pour afficher le contenu de la chaîne de caractères `nom` en majuscules ou `{{ nom + " suffixe" }}` pour afficher la chaîne de caractères suivie du mot `suffixe`.

Les accolades et le signe pourcentage `{% %}` servent pour les instructions de contrôle de flux dans un `template`. Par exemple, pour créer une boucle `for` dans un `template`, vous pouvez utiliser `{% for i in range(10) %}` pour commencer la boucle et `{% endfor %}` pour la terminer. On peut aussi utiliser des boucles `while` :

```
{% set counter = 0 %}{% while counter < 5 %} …{% endwhile %}
```

Il est aussi possible d'utiliser des clauses conditionnelles comme :

```
{% if i == 3 %} … {%endif%}
```

Les instructions de contrôle de flux sont utilisées pour les opérations qui ne doivent pas être affichées à l'utilisateur.

Pour permettre à Flask d'utiliser ce fichier HTML, il suffit de lui associer une route :

```
1. @app.route('/upload')
2. def upload_file_render():
3.     return render_template('upload.html', message = 'écrire un message ici')
```

Pour que l'application Web soit fonctionnelle, il faut aussi créer une route correspondant à une fonction `upload_file()` puisque c'est avec cette route que le formulaire communique. Cette route devra recevoir une image.

Il peut être pratique d'en faire un fichier. Le module `werkzeug.utils` est un module utilitaire fourni avec le framework Flask. Il contient plusieurs fonctions utiles pour la création d'applications web, notamment la fonction `secure_filename()`. Cette fonction prend un nom de fichier et renvoie une version sécurisée de ce nom de fichier qui peut être stockée en toute sécurité sur un système de fichiers régulier. La version sécurisée du nom de fichier est une chaîne ASCII qui ne contient que des caractères alphanumériques, des tirets et des points. Cette fonction est utile pour éviter les attaques d'injection de chemin d'accès lors du traitement des fichiers téléchargés par les utilisateurs. Pour utiliser la fonction `secure_filename()`, il faut l'importer à partir du module `werkzeug.utils`

```
from werkzeug.utils import secure_filename
```

Le code Flask suivant crée une route qui accepte les données d'une image et qui en fait un fichier :

```
1. @app.route('/uploader', methods = ['POST'])
2. def upload_file():
3.      f = request.files['file']
4.      sn = secure_filename(f.filename)
5.      f.save(sn)
```

Prenez note que nous utilisons une route de type POST. Contrairement aux routes de type GET, qui sont utilisées pour récupérer des données à partir d'un serveur web, les routes de type POST sont utilisées pour envoyer des données à un serveur web pour traitement. Les données

envoyées dans une requête POST sont généralement stockées dans le corps de la requête et peuvent être utilisées pour créer, mettre à jour ou supprimer des ressources sur le serveur web.

Les routes de type POST servent souvent pour les formulaires web, où les utilisateurs saisissent des informations qui doivent être envoyées au serveur Web pour traitement. Les données envoyées dans une requête POST peuvent être cryptées pour plus de sécurité, ce qui les rend plus appropriées pour l'envoi d'informations sensibles telles que des noms d'utilisateur et des mots de passe.

La fonction associée à notre route demeure incomplète et il n'est pas nécessaire d'enregistrer l'image. Nous pouvons rapidement traiter les données comme ceci :

```
@app.route('/uploader', methods = ['POST'])
def upload_file():
    print("upload_file")
    f = request.files['file']
    lat, long = get_exif_location(exifread.process_file(f))
    print(lat, long)
    if lat is None:
        return render_template('formulaire.html',
                               message = "Il n'y avait pas de données GPS dans l'image."
                                +" Choisissez une autre image.")
    link = "https://www.openstreetmap.org/?mlat="+str(lat)+"&mlon="+str(long)+"&zoom=15"
    return "<html><body><a href=\""+link+"\">carte</a></body></html>"

```

La fonction prend ensuite la fonction `get_exif_location()` pour extraire les informations de géolocalisation du fichier image. Si les informations de géolocalisation sont disponibles, la fonction crée un lien vers une carte OpenStreetMap qui affiche l'emplacement où la photo a été prise. La fonction renvoie ensuite une réponse HTML qui inclut le lien vers la carte. Si les informations de géolocalisation ne sont pas disponibles, la fonction retourne l'utilisateur au formulaire. Un message est affiché pour expliquer l'erreur.

Nous avons maintenant une application Web complète qui peut recevoir une image, en extraire les coordonnées géographiques et offrir un lien vers la carte géographique correspondante :

CodePython/chapitre14/virtuel/imageserver.py

```
import exifread
from flask import Flask, render_template, request, redirect

def get_if_exist(data, key):
    if key in data:
        return data[key]
    return None

def convert_to_degrees(value):
```

```
    d = float(value.values[0].num) / float(value.values[0].den)
    m = float(value.values[1].num) / float(value.values[1].den)
    s = float(value.values[2].num) / float(value.values[2].den)

    return d + (m / 60.0) + (s / 3600.0)

def get_exif_location(exif_data):
    lat = None
    lon = None

    gps_latitude = get_if_exist(exif_data, 'GPS GPSLatitude')
    gps_latitude_ref = get_if_exist(exif_data, 'GPS GPSLatitudeRef')
    gps_longitude = get_if_exist(exif_data, 'GPS GPSLongitude')
    gps_longitude_ref = get_if_exist(exif_data, 'GPS GPSLongitudeRef')

    if gps_latitude and gps_latitude_ref and gps_longitude and gps_longitude_ref:
        lat = convert_to_degrees(gps_latitude)
        if gps_latitude_ref.values[0] != 'N':
            lat = 0 - lat

        lon = convert_to_degrees(gps_longitude)
        if gps_longitude_ref.values[0] != 'E':
            lon = 0 - lon

    return lat, lon

def get_exif_data(image_file):
    with open(image_file, 'rb') as f:
        exif_tags = exifread.process_file(f)
    return exif_tags


app = Flask(__name__)

@app.route('/uploader', methods = ['POST'])
def upload_file():
    print("upload_file")
    f = request.files['file']
    lat, long = get_exif_location(exifread.process_file(f))
    print(lat, long)
    if lat is None:
        return render_template('formulaire.html',
                               message = "Il n'y avait pas de données GPS dans l'image."
                                +" Choisissez une autre image.")
    link = "https://www.openstreetmap.org/?mlat="+str(lat)+"&mlon="+str(long)+"&zoom=15"
    return "<html><body><a href=\""+link+"\">carte</a></body></html>"

@app.route('/')
def index():
    return redirect('/upload')

@app.route('/upload')
def upload_file_render():
   print("upload_file_render")
   return render_template('formulaire.html', message = "Choisissez une image JPEG.")

app.run()
```

13.4 Serveur sécurisé

Nos applications Flask utilisent le protocole HTTP et elles ne sont donc pas sécurisées. Pour utiliser HTTPS avec Flask, il faut d'abord générer un certificat SSL pour votre application. A cet effet, il faut installer OpenSSL sur votre système si ce n'est pas déjà le fait. Vous pouvez télécharger OpenSSL à partir du site officiel d'OpenSSL. Un certificat auto-signé peut être généré pour le développement en utilisant la commande suivante :

```
openssl req -x509 -newkey rsa:4096 -nodes -out cert.pem -keyout key.pem -days
365
```

Cette commande génère un certificat auto-signé qui est valide pendant 365 jours. La durée de validité est spécifiée par l'argument `-days`. Une fois que le certificat SSL généré, il peut être employé pour exécuter votre application Flask sur HTTPS. Pour ce faire, il faut ajouter les options `ssl_context` et `port` à l'appel `app.run()`. Voici un exemple de code Python qui utilise un certificat auto-signé pour exécuter une application Flask sur HTTPS :

```
 1. from flask import Flask
 2.
 3. app = Flask(__name__)
 4.
 5. @app.route('/')
 6. def index():
 7.     return 'Bonjour!'
 8.
 9. if __name__ == '__main__':
10.     app.run(ssl_context=('cert.pem', 'key.pem'))
11.
```

Dans cet exemple, l'application Flask est créée avec une route `/` qui renvoie `Bonjour!`. Ensuite `app.run()` exécute l'application sur le port 5000 avec le certificat SSL auto-signé généré précédemment.

13.5 Intégration d'une base de données SQL

Les bases de données relationnelles sont largement utilisées dans le développement web car elles permettent de stocker et d'organiser des données de manière efficace et structurée. Les bases de données relationnelles sont basées sur le modèle relationnel, qui utilise des tables pour stocker des données. Chaque table est composée de colonnes qui représentent les différents types de données que la table peut contenir, et chaque ligne représente un enregistrement unique dans la table.

Les bases de données relationnelles sont faciles à interroger et à mettre à jour, ce qui les rend idéales pour les applications web qui nécessitent un accès rapide aux données. Les bases de données relationnelles sont également extensibles, ce qui signifie que de nouvelles catégories de données peuvent être ajoutées après la création de la base de données originale sans avoir besoin de modifier toutes les applications existantes. Les développeurs et les analystes de données utilisent souvent des bases de données relationnelles pour lancer des requêtes analytiques.

La syntaxe SQL est utilisée pour interagir avec les bases de données relationnelles. Voici un résumé des commandes SQL les plus courantes:

- `CREATE TABLE` : Cette commande crée une nouvelle table dans une base de données. La syntaxe est la suivante: `CREATE TABLE table_name (column1 datatype, column2 datatype, column3 datatype, ...);`. Les paramètres `column1`, `column2`, etc. spécifient les noms des colonnes de la table, tandis que les paramètres `datatype` spécifient le type de données de chacune des colonnes.

- `INSERT INTO` : Cette commande est utilisée pour insérer de nouvelles données dans une table existante. La syntaxe est la suivante: `INSERT INTO table_name (column1, column2, column3, ...) VALUES (value1, value2, value3, ...);`. Les paramètres `column1`, `column2`, etc. spécifient les noms des colonnes dans lesquelles les données doivent être insérées, tandis que les paramètres `value1`, `value2`, etc. spécifient les valeurs à insérer.

- `SELECT` : Cette commande est utilisée pour récupérer des données à partir d'une ou plusieurs tables dans une base de données. La syntaxe est la suivante: `SELECT column1, column2, column3, ... FROM table_name WHERE condition;`. Les paramètres `column1`, `column2`, etc. spécifient les noms des colonnes à récupérer, tandis que le paramètre `table_name` spécifie la table à partir de laquelle récupérer les données. Le paramètre `condition` spécifie une condition qui doit être remplie pour que les données soient récupérées.

SQLite est une base de données relationnelle compatible avec SQL. Contrairement à d'autres systèmes basés sur SQL tels que MySQL et PostgreSQL, SQLite n'utilise pas d'architecture client-serveur. L'ensemble du programme est contenu dans une bibliothèque C, qui est intégrée dans les applications. La base de données devient une partie intégrante du programme, éliminant les processus autonomes gourmands en ressources. SQLite stocke ses données dans un seul fichier multiplateforme. Comme il n'y a pas de serveur dédié ou de système de fichiers spécialisé, le « déploiement » SQLite est aussi simple que de lier sa bibliothèque et de créer un nouveau fichier régulier.

On peut utiliser SQLite avec Python en installant le module `sqlite3` (`python -m pip install sqlite3`). Il faut toujours s'assurer que le fichier correspondant à la base de données est créé et accessible au

lancement du serveur. Il est possible de le faire avec un code semblable à celui-ci :

```
1. with sqlite3.connect('exemple.db') as conn:
2.     cursor = conn.cursor()
3.     cursor.execute('''CREATE TABLE IF NOT EXISTS users
4.                       (id INTEGER PRIMARY KEY AUTOINCREMENT,
5.                       name TEXT NOT NULL,
6.                       email TEXT NOT NULL);''')
```

Le code permet de créer le fichier contenant les données au besoin et d'y créer une table nommée `users` dans une base de données SQLite. Si le fichier existe déjà, il sera réutilisé : votre application peut donc garder ses données d'une exécution à l'autre. Voici comment cela fonctionne :

1. La première ligne crée une connexion à la base de données SQLite stockée dans le fichier `exemple.db`. La connexion est stockée dans la variable `conn`.
2. La deuxième ligne crée un objet curseur à partir de la connexion. Le curseur est stocké dans la variable `cursor`.
3. La troisième ligne exécute une instruction SQL pour créer une table nommée `users` si elle n'existe pas déjà. Cette table a trois colonnes: `id`, `name`, et `email`. La colonne `id` est définie comme clé primaire et est auto-incrémentée. Les colonnes `name` et `email` sont définies comme non nulles.
4. L'instruction SQL est exécutée en appelant la méthode `execute()` sur l'objet curseur.
5. Enfin, la clause `with` garantit que la connexion à la base de données est fermée correctement après l'exécution de l'instruction SQL. En Python, la clause `with` est utilisée pour les ressources non gérées (comme les flux de fichiers) et garantir que les ressources sont correctement nettoyées après leur utilisation. La clause `with` permet de simplifier la gestion des ressources courantes telles que les flux de fichiers. Par exemple, si un fichier est ouvert avec la clause `with`, le fichier sera automatiquement fermé à la fin du bloc `with`, même si une exception est levée. Voici un exemple de code qui utilise la clause `with` pour ouvrir un fichier : `with open('fichier.txt', 'r') as f: contenu = f.read()`. Dans cet exemple, `fichier.txt` est ouvert en mode lecture (`'r'`) avec la clause `with`. Le contenu du fichier est ensuite lu dans la variable `contenu`. À la fin du bloc `with`, le fichier est automatiquement fermé.

Le fichier nommé `maison.html` doit être placé dans le répertoire `templates` d'un nouveau projet Flask :

CodePython/chapitre14/bd/templates/maison.html

```
 1. <!DOCTYPE html>
 2. <html>
 3. <head>
 4.     <title>Page d'accueil</title>
 5. </head>
 6. <body>
 7.     {% if users | length != 0 %}
 8.     <h1>Utilisateurs</h1>
 9.     <table>
10.         <tr>
11.             <th>Nom</th>
12.             <th>Courriel</th>
13.         </tr>
14.         {% for user in users %}
15.         <tr>
16.             <td>{{ user[1] }}</td>
17.             <td>{{ user[2] }}</td>
18.         </tr>
19.         {% endfor %}
20.     </table>
21.     {% endif %}
22.
23.     <h1>Ajouter un utilisateur</h1>
24.     <form action="/add" method="post">
25.         <label for="name">Nom :</label>
26.         <input type="text" id="name" name="name"><br><br>
27.         <label for="email">Courriel :</label>
28.         <input type="text" id="email" name="email"><br><br>
29.         <input type="submit" value="Submit">
30.     </form>
31. </body>
32. </html>
```

La ligne `{% if users | length != 0 %}` est une instruction Jinja2 qui vérifie si la variable `users` contient des éléments. Si c'est le cas, le contenu entre `{% if users | length != 0 %}` et `{% endif %}` sera affiché. Sinon, il sera ignoré. Les balises `<tr>`, `<th>` et `<td>` définissent les lignes et les colonnes du tableau. La ligne `{% for user in users %}` est une instruction qui crée une boucle pour chaque élément dans la variable `users`. Le contenu entre `{% for user in users %}` et `{% endfor %}` sera répété pour chaque élément dans `users`. Les lignes `{{ user[1] }}` et `{{ user[2] }}` sont des instructions Jinja2 qui affichent les éléments 1 et 2 de chaque élément dans `users`.

La balise `<form>` définit un formulaire HTML pour ajouter un utilisateur à la base de données. Les balises `<label>` et `<input>` définissent les champs du formulaire. La ligne `action="/add"` définit l'URL à laquelle les données du formulaire seront envoyées lorsque l'utilisateur soumettra le

formulaire. La ligne `method="post"` définit la méthode HTTP utilisée pour envoyer les données du formulaire.

Il suffit ensuite de créer un fichier Python contenant l'application Flask au sein de votre répertoire de travail :

```
 1. from flask import Flask, render_template, request, redirect
 2. import sqlite3
 3.
 4. app = Flask(__name__)
 5.
 6. @app.route('/')
 7. def maison():
 8.     conn = sqlite3.connect('exemple.db')
 9.     c = conn.cursor()
10.     c.execute('SELECT * FROM users')
11.     users = c.fetchall()
12.     conn.close()
13.     return render_template('maison.html', users=users)
14.
15. @app.route('/add', methods=['POST'])
16. def ajoute():
17.     name = request.form['name']
18.     email = request.form['email']
19.
20.     conn = sqlite3.connect('exemple.db')
21.     c = conn.cursor()
22.     c.execute('INSERT INTO users (name, email) VALUES (?, ?)',
(name, email))
23.     conn.commit()
24.     conn.close()
25.     return redirect('/')
26.
27. with sqlite3.connect('exemple.db') as conn:
28.     cursor = conn.cursor()
29.     cursor.execute('''CREATE TABLE IF NOT EXISTS users
30.                       (id INTEGER PRIMARY KEY AUTOINCREMENT,
31.                       name TEXT NOT NULL,
32.                       email TEXT NOT NULL);''')
33. app.run()
```

Ce code définit une route pour la page d'accueil de l'application. Cette route se connecte à la base de données SQLite, récupère toutes les entrées de la table `users`, ferme la connexion à la base de données et renvoie le résultat à un template HTML appelé `maison.html`. IL y a aussi une route pour ajouter un nouvel utilisateur à la base de données. Cette route récupère les informations sur l'utilisateur à partir d'un formulaire HTML, se connecte à la base de données SQLite, insère les informations dans la table `users`, ferme la connexion à la base de données et redirige l'utilisateur vers la page d'accueil. IL y a une routine créant une table nommée `users` dans la base

de données SQLite si elle n'existe pas déjà. Il s'agit d'une application Web complète, qui affiche le contenu de la table et qui vous permet d'ajouter des utilisateurs au besoin.

Nous pouvons aussi ajouter une fonction à notre application géographique afin de garder en mémoire la longitude et la latitude des images soumises. La fonction suivante peut être utile :

```
1. def log(long,lat):
2.   with sqlite3.connect("img.db") as con:
3.     tables = [row[0] for row in con.execute("SELECT name FROM sqlite_master WHERE
type='table'")]
4.     if not "geo" in tables:
5.         con.execute("CREATE TABLE geo (date TEXT, long NUMERIC, lat NUMERIC)")
6.     dt = datetime.now()
7.     con.execute("INSERT INTO geo (date,long,lat) values (\""+str(dt)+"\","+str(long)+",
"+str(lat)+") ")
```

Voici comment cela fonctionne:

- La fonction `log()` prend deux arguments, `long` et `lat`, qui représentent la longitude et la latitude de la localisation.
- La clause `with` crée un contexte d'exécution pour un groupe d'instructions sous le contrôle d'un gestionnaire de contexte. Dans ce cas, le gestionnaire de contexte est un objet de connexion à la base de données SQLite.
- La ligne `tables = [row[0] for row in con.execute("SELECT name FROM sqlite_master WHERE type='table'")]` exécute une requête SQL pour récupérer les noms des tables dans la base de données. Les noms des tables sont stockés dans une liste appelée tables.
- La ligne `if not "geo" in tables:` vérifie si la table `geo` existe dans la base de données. Si elle n'existe pas, une nouvelle table nommée `geo` est créée avec trois colonnes : `date`, `long` et `lat`.
- La ligne `dt = datetime.now()` crée un objet `datetime` qui représente l'heure actuelle.
- La ligne `con.execute("INSERT INTO geo (date,long,lat) values (\""+str(dt)+"\","+str(long)+", "+str(lat)+") ")` insère les données de localisation dans la table `geo`. Les données sont stockées dans les colonnes `date`, `long` et `lat`.

Le programme Python suivant intègre à la fois l'extraction de longitude et de latitude, mais aussi son inclusion dans une base de données :

CodePython/chapitre14/bd/server.py

```
import exifread
from flask import Flask, render_template, request, redirect
from datetime import datetime
import sqlite3

def get_if_exist(data, key):
    if key in data:
        return data[key]
    return None

def convert_to_degrees(value):
    """
    Helper function to convert the GPS coordinates stored in the EXIF to degress in float format
    :param value:
    :type value: exifread.utils.Ratio
    :rtype: float
    """
    d = float(value.values[0].num) / float(value.values[0].den)
    m = float(value.values[1].num) / float(value.values[1].den)
    s = float(value.values[2].num) / float(value.values[2].den)

    return d + (m / 60.0) + (s / 3600.0)

def get_exif_location(exif_data):
    """
    Returns the latitude and longitude, if available, from the provided exif_data (obtained through get_exif_data above)
    """
    lat = None
    lon = None

    gps_latitude = get_if_exist(exif_data, 'GPS GPSLatitude')
    gps_latitude_ref = get_if_exist(exif_data, 'GPS GPSLatitudeRef')
    gps_longitude = get_if_exist(exif_data, 'GPS GPSLongitude')
    gps_longitude_ref = get_if_exist(exif_data, 'GPS GPSLongitudeRef')

    if gps_latitude and gps_latitude_ref and gps_longitude and gps_longitude_ref:
        lat = convert_to_degrees(gps_latitude)
        if gps_latitude_ref.values[0] != 'N':
            lat = 0 - lat

        lon = convert_to_degrees(gps_longitude)
        if gps_longitude_ref.values[0] != 'E':
            lon = 0 - lon

    return lat, lon

def get_exif_data(image_file):
    with open(image_file, 'rb') as f:
        exif_tags = exifread.process_file(f)
    return exif_tags

def log(long,lat):
  with sqlite3.connect("img.db") as con:
    tables = [row[0] for row in con.execute("SELECT name FROM sqlite_master WHERE type='table'")]
    if not "geo" in tables:
        con.execute("CREATE TABLE geo (date TEXT, long NUMERIC, lat NUMERIC)")
    dt = datetime.now()
    con.execute("INSERT INTO geo (date,long,lat) values (\""+str(dt)+"\","+str(long)+", "+str(lat)+") ")

app = Flask(__name__)

@app.route('/')
def index():
    return redirect('/upload')

@app.route('/upload')
def upload_file_render():
```

```
68.     return render_template('formulaire.html', message = "Choisissez une image JPEG.")
69.
70. @app.route('/uploader', methods = ['POST'])
71. def upload_file():
72.     f = request.files['file']
73.     f = request.files['file']
74.     lat, long = get_exif_location(exifread.process_file(f))
75.     if lat is None:
76.         return render_template('formulaire.html', "Il n'y avait pas de données GPS dans
l'image. Choisissez une autre image.")
77.     link = "https://www.openstreetmap.org/?mlat="+str(lat)+"&mlon="+str(long)+"&zoom=15"
78.     log(long,lat)
79.     return "<html><body><a href=\""+link+"\">carte</a></body></html>"
80.
81. app.run()
```

Exercice. Écrivez un programme Python avec Flask qui stocke dans une base de données les dates d'accès au serveur.

Solution. Notre programme définit le nom du fichier de la base de données avec la variable `db_file`, qui contient la chaîne de caractères `access.db`. Il définit la fonction `create_table()`, qui crée une table dans la base de données si elle n'existe pas déjà. Cette fonction fait les choses suivantes : elle se connecte à la base de données avec la commande `conn = sqlite3.connect(db_file)`, qui renvoie un objet connexion, elle crée un curseur avec la méthode `conn.cursor()`, qui permet d'exécuter des commandes SQL sur la base de données, elle exécute la commande SQL `CREATE TABLE IF NOT EXISTS access (id INTEGER PRIMARY KEY, date TEXT)` avec la méthode `cursor.execute()`, qui crée la table `access` avec deux colonnes: `id` (clé primaire) et `date` (la clause `IF NOT EXISTS` permet de vérifier si la table existe déjà avant de la créer), elle valide les changements avec la méthode `conn.commit()`, qui enregistre les modifications dans la base de données, elle ferme la connexion avec la méthode `conn.close()`, qui libère les ressources utilisées par la connexion. Le programme définit la fonction `insert_date()`, qui insère la date du dernier accès à la page web dans la table `access`. Cette fonction fait les choses suivantes : elle se connecte à la base de données de la même manière que la fonction `create_table()`, elle crée un curseur de la même manière que la fonction `create_table()`, elle exécute la commande SQL `INSERT INTO access (date) VALUES (?)` avec la méthode `cursor.execute()`, qui insère la date passée en paramètre dans la colonne `date` de la table `access`. Le caractère `?` est un paramètre qui sera remplacé par la valeur fournie en argument, sous forme de tuple, l'`id` sera généré automatiquement par le système de gestion de base de données, elle valide les changements de la même manière que la fonction `create_table()`, elle ferme la connexion de la même manière que la fonction `create_table()`. Le programme définit la fonction `get_table_content()`, qui récupère le contenu complet de la table `access`. Cette fonction fait les choses suivantes : elle se connecte à la base de données de la même manière que la fonction `create_table()`, elle crée

un curseur de la même manière que la fonction `create_table()`, elle exécute la commande `SQL SELECT * FROM access` avec la méthode `cursor.execute()`, qui sélectionne toutes les lignes de la table `access`, elle récupère le résultat avec la méthode `cursor.fetchall()`, qui renvoie une liste de tuples, où chaque tuple représente une ligne de la table, avec les valeurs de l'`id` et de la `date`, elle ferme la connexion, elle retourne le résultat avec la commande `return result`. Le programme définit la route principale de l'application web avec le décorateur `@app.route("/")`, qui associe la fonction index à l'URL racine de l'application. Il définit la fonction `index()`, qui sera exécutée quand un utilisateur accède à la route principale. Elle importe le module `datetime` pour obtenir la date et l'heure actuelles. Elle configure les paramètres régionaux pour afficher la date et l'heure en français du Canada. Elle crée une table dans la base de données si elle n'existe pas déjà. Elle insère la date courante dans la base de données. Elle récupère le contenu de la base de données et l'affiche sous la forme d'un document HTML.

CodePython/chapitre14/Exercice7.py

```
1.  # Importer le module flask
2.  from flask import Flask
3.
4.  # Importer le module sqlite3, qui permet de manipuler des bases de données sqlite
5.  import sqlite3
6.
7.  # Créer une instance de l'application Flask
8.  app = Flask(__name__)
9.
10. # Définir le nom du fichier de la base de données
11. db_file = "access.db"
12.
13. # Définir la fonction qui crée la table de la base de données si elle n'existe pas
14. def create_table():
15.     # Se connecter à la base de données
16.     conn = sqlite3.connect(db_file)
17.     # Créer un curseur pour exécuter des commandes SQL
18.     cursor = conn.cursor()
19.     # Créer la table access si elle n'existe pas, avec deux colonnes: id et date
20.     cursor.execute("CREATE TABLE IF NOT EXISTS access (id INTEGER PRIMARY KEY, date
TEXT)")
21.     # Valider les changements
22.     conn.commit()
23.     # Fermer la connexion
24.     conn.close()
25.
26. # Définir la fonction qui insère la date du dernier accès dans la base de données
27. def insert_date(date):
28.     # Se connecter à la base de données
29.     conn = sqlite3.connect(db_file)
30.     # Créer un curseur pour exécuter des commandes SQL
31.     cursor = conn.cursor()
32.     # Insérer la date dans la table access, en laissant l'id se générer automatiquement
33.     cursor.execute("INSERT INTO access (date) VALUES (?)", (date,))
34.     # Valider les changements
35.     conn.commit()
36.     # Fermer la connexion
37.     conn.close()
38.
39. # Définir la fonction qui récupère le contenu de la table access
40. def get_table_content():
41.     # Se connecter à la base de données
```

```
    conn = sqlite3.connect(db_file)
    # Créer un curseur pour exécuter des commandes SQL
    cursor = conn.cursor()
    # Sélectionner toutes les lignes de la table access
    cursor.execute("SELECT * FROM access")
    # Récupérer le résultat sous forme de liste de tuples
    result = cursor.fetchall()
    # Fermer la connexion
    conn.close()
    # Retourner le résultat
    return result

# Définir la route principale
@app.route("/")
def index():
    # Importer le module datetime
    from datetime import datetime

    # Importer le module locale, qui permet de gérer les paramètres régionaux
    import locale

    # Définir le paramètre régional à "fr_CA" pour le français du Canada
    locale.setlocale(locale.LC_TIME, "fr_CA")

    # Obtenir la date et l'heure
    now = datetime.now()

    # Formater la date et l'heure en français avec la méthode strftime
    date = now.strftime("%A %d %B %Y")
    heure = now.strftime("%H:%M:%S")

    # Créer la table de la base de données si elle n'existe pas
    create_table()

    # Insérer la date du dernier accès dans la base de données
    insert_date(date)

    # Récupérer le contenu de la table access
    content = get_table_content()

    # Créer une variable pour stocker le code HTML du tableau
    table = ""

    # Parcourir le contenu de la table
    for row in content:
        # Ajouter une ligne au tableau avec les valeurs de l'id et de la date
        table += f"<tr><td>{row[0]}</td><td>{row[1]}</td></tr>"

    # Retourner une page web avec la date et l'heure et le tableau
    return f"""
    <html>
        <head>
            <title>Date et heure</title>
        </head>
        <body>
            <h1>Bonjour, voici la date et l'heure:</h1>
            <p>Date: {date}</p>
            <p>Heure: {heure}</p>
            <h2>Voici le contenu de la table access:</h2>
            <table border="1">
                <tr><th>Id</th><th>Date</th></tr>
                {table}
            </table>
        </body>
    </html>
    """

# Lancer l'application Flask
if __name__ == "__main__":
    app.run()
```

14 Développement d'applications WebSocket asynchrones

Les applications Web conventionnelles utilisent le protocole HTTP. Le protocole HTTP est essentiellement asymétrique : une application-client telle qu'un navigateur émet des requêtes et le serveur (par exemple Flask) répond. Il n'est pas possible pour le serveur d'initier une communication vers le client. Certains types d'applications sont donc plus difficiles à concevoir. Par exemple, si nous souhaitons concevoir un jeu vidéo multijoueur avec le protocole http, tel qu'un jeu d'échec, nous pourrions avoir un serveur, et deux navigateurs branchés au serveur. Quand un des joueurs déplace une pièce au sein de son navigateur, le navigateur peut en informer le serveur par l'entremise d'une requête http. Mais comment en informer le second navigateur ? Une solution consiste à faire en sorte que les navigateurs font des requêtes à intervalles réguliers au serveur. Une meilleure solution consiste en l'utilisation d'un autre protocole, le protocole WebSocket.

14.1 WebSocket

WebSocket est un protocole réseau qui permet de créer des canaux de communication bidirectionnels entre les navigateurs et les serveurs web. La plupart des navigateurs supportent WebSocket, bien que la norme soit relativement récente (2011). Il permet la notification au client d'un changement d'état du serveur, sans que ce dernier ait à effectuer une requête.

La norme RFC 6455 décrit le protocole WebSocket, qui permet une communication bidirectionnelle entre un client et un serveur en utilisant une seule connexion TCP. Le protocole consiste en une poignée de main d'ouverture, suivie d'un encodage de message de base, qui est transmis sur la connexion TCP. La poignée de main d'ouverture est utilisée pour négocier les paramètres de la connexion, tels que les sous-protocoles, les extensions et les en-têtes personnalisés. Les messages sont encodés en binaire ou en texte, et peuvent être divisés en fragments pour une transmission plus efficace. La norme définit également les codes d'état de fermeture, qui sont utilisés pour indiquer la raison de la fermeture de la connexion.

WebSocket s'appuie sur TCP ce qui signifie qu'il requiert une notion de connexion. Cette exigence introduit une certaine complexité au sein du protocole, et peut limiter la performance par rapport à une approche sur

UDP. En revanche, en utilisant TCP, le protocole WebSocket simplifie le travail du programmeur puisque les données arrivent en séquence et il n'y a normalement pas de pertes de données.

14.2 Programmation asynchrone

La programmation d'un serveur WebSocket est plus aisée en programmation asynchone. La programmation asynchrone est une technique de programmation qui permet à un programme de démarrer une tâche à l'exécution potentiellement longue et de continuer à réagir aux autres événements pendant l'exécution de cette tâche. Par exemple, on peut vouloir charger un fichier sur le disque, mais aussi continuer à faire d'autres opérations. Dans le cas de WebSocket, on souhaite gérer des connexions multiples et des demandes simultanées.

Les mots-clés `async/await` permettent de simplifier la programmation asynchrone en Python. Le mot-clé `async` est utilisé pour créer une coroutine Python (aussi appelée fonction asynchrone), tandis que le mot-clé `await` suspend l'exécution d'une coroutine jusqu'à ce qu'elle soit terminée et renvoie les données de résultat. Le mot-clé `await` ne peut être utilisé que dans une fonction asynchrone. Une coroutine ne peut pas être appelée directement comme une fonction Python conventionnelle. On peut soit l'appeler en la précédant du mot-clé `await`, soit avec la méthode `asyncio.run()`, comme dans cet exemple :

```
1. import asyncio
2.
3. async def my_coroutine():
4.     print("Hello, world!")
5.
6. async def main():
7.     await my_coroutine()
8.
9. asyncio.run(main())
```

Dans cet exemple, la fonction `my_coroutine()` est une coroutine qui affiche `Hello, world!`. La fonction `main()` appelle la coroutine en utilisant les mots-clés `async/await`. Enfin, la fonction `asyncio.run()` est utilisée pour exécuter la coroutine.

On peut aussi exécuter les coroutine en les transformant en tâches. La méthode `asyncio.create_task()` permet d'exécuter une coroutine de manière asynchrone. Elle crée une tâche `asyncio` qui s'exécute en arrière-plan et retourne un objet `Task` qui peut être utilisé pour surveiller l'état de la tâche ou pour l'annuler si nécessaire. Voici un exemple de code qui

utilise `asyncio.create_task()` pour exécuter deux coroutines en parallèle :

```
 1. import asyncio
 2.
 3. async def coroutine1():
 4.     print("Coroutine 1 débutée")
 5.     await asyncio.sleep(2)
 6.     print("Coroutine 1 terminée")
 7.
 8. async def coroutine2():
 9.     print("Coroutine 2 débutée")
10.     await asyncio.sleep(1)
11.     print("Coroutine 2 terminée")
12.
13. async def main():
14.     task1 = asyncio.create_task(coroutine1())
15.     task2 = asyncio.create_task(coroutine2())
16.
17.     await task1
18.     await task2
19.
20. asyncio.run(main())
21.
```

La méthode `asyncio.gather()` offre une autre approche pour exécuter plusieurs coroutines en parallèle. Elle prend en entrée une liste de coroutines et les exécute simultanément. Voici comment vous pouvez réécrire l'exemple précédent en utilisant `asyncio.gather()` :

```
 1. import asyncio
 2.
 3. async def coroutine1():
 4.     print("Coroutine 1 débutée")
 5.     await asyncio.sleep(2)
 6.     print("Coroutine 1 terminée")
 7.
 8. async def coroutine2():
 9.     print("Coroutine 2 débutée")
10.     await asyncio.sleep(1)
11.     print("Coroutine 2 terminée")
12.
13. async def main():
14.     await asyncio.gather(coroutine1(), coroutine2())
15.
16. asyncio.run(main())
```

Pour illustrer l'utilité de ce concept, supposons que vous souhaitiez charger deux pages Web en même temps. Il est possible de cette manière d'aller plus vite. Le code Python suivant utilise les mots-clés `async`/`await` et le module `aiohttp` pour charger deux pages Web en même temps. La

fonction charge le module `aiohttp` pour récupérer le contenu d'une page Web donnée. La fonction main crée deux tâches en utilisant la fonction `asyncio.create_task()` pour charger les pages Web de Google et de l'Université du Québec à Montréal (UQAM) en parallèle. La fonction `asyncio.gather()` est utilisée pour exécuter les deux tâches en parallèle et renvoyer les résultats. Enfin, la fonction main affiche la longueur du contenu de chaque page Web. Lorsque vous exécutez ce code, vous verrez que les deux pages Web sont chargées en même temps, ce qui permet d'accélérer le temps de chargement total.

```
 1. import aiohttp
 2. import asyncio
 3.
 4. async def charge(url):
 5.   print("charge ",url)
 6.   async with aiohttp.ClientSession() as session:
 7.     async with session.get(url) as response:
 8.         r = await response.text()
 9.   print("chargement de l'URL ",url," terminé")
10.   return r
11.
12. async def main():
13.     task1 = asyncio.create_task(
14.         charge("https://www.google.com/"))
15.
16.     task2 = asyncio.create_task(
17.         charge("https://www.uqam.ca/"))
18.     await asyncio.gather(task1, task2)
19.     print(len(task1.result()))
20.     print(len(task2.result()))
21.
22. if __name__ == '__main__':
23.    asyncio.run(main())
```

Exercice. Écrivez programme Python qui charge deux fichiers (`fichier1.txt` et `fichier2.txt`) en même temps.

Solution. Notre programme définit une fonction asynchrone `lire_fichier()`, qui prend en paramètre le nom d'un fichier et qui l'ouvre en mode lecture, lit son contenu et l'affiche. Il définit ensuite une fonction principale `main()`, qui crée deux tâches asynchrones pour lire deux fichiers différents, `fichier1.txt` et `fichier2.txt`, et qui attend que les deux tâches soient terminées. Enfin, il exécute la fonction principale avec la fonction `asyncio.run()`, qui lance la boucle d'événements d'`asyncio`, qui gère l'exécution des tâches asynchrones.

CodePython/chapitre15/Exercice1.py

```
1.  # Importer le module asyncio, qui permet de gérer la
    programmation asynchrone
2.  import asyncio
3.
4.  # Définir une fonction asynchrone qui lit un fichier et affiche
    son contenu
5.  async def lire_fichier(nom):
6.      # Ouvrir le fichier en mode lecture
7.      with open(nom, "r") as fichier:
8.          # Lire le contenu du fichier
9.          contenu = fichier.read()
10.         # Afficher le nom et le contenu du fichier
11.         print(f"Le fichier {nom} contient:\n{contenu}")
12.
13. # Définir une fonction principale qui lance les tâches
    asynchrones
14. async def main():
15.     # Créer deux tâches asynchrones pour lire deux fichiers
    différents
16.     tache1 = asyncio.create_task(lire_fichier("fichier1.txt"))
17.     tache2 = asyncio.create_task(lire_fichier("fichier2.txt"))
18.     # Attendre que les deux tâches soient terminées
19.     await tache1
20.     await tache2
21.
22. # Exécuter la fonction principale avec la boucle d'événements
    d'asyncio
23. asyncio.run(main())
```

14.3 Ping-Pong WebSocket

Il est relativement facile de créer un client WebSocket en Python. Nous pouvons créer une fonction `client()` définie comme une fonction asynchrone (coroutine) qui se connecte à un serveur WebSocket en utilisant l'adresse `ws://localhost:8080`. Une fois la connexion établie, la fonction attend une réponse. Si une réponse est reçue, elle est stockée dans la variable `response`. Dès qu'une réponse est obtenue, elle retourne au serveur le message `allo!`. Le client attend de nouveau réponse. Le code continue à envoyer le message `allo!` au serveur et à attendre une réponse tant que la boucle `while` est vraie.

```
1. import websockets
2. async def client():
3.     async with websockets.connect('ws://localhost:8080') as websocket:
4.         while True:
5.             response = await websocket.recv()
6.             await websocket.send('allo!')
```

Une des applications les plus simples de WebSocket permet la communication entre deux clients en passant par un serveur. Il y a un seul serveur. Le serveur prend tous les messages qu'il reçoit d'un client et les

retransmet à un autre client. Nous avons deux clients. Les deux clients établissent une connexion avec le serveur, de sorte que nous avons deux connexions. Les clients s'engagent alors dans un échange continu :

1. Le client 1 envoie un message au serveur.
2. Le serveur reçoit le message et le diffuse au deuxième client.
3. Le client 2 reçoit le message du serveur.
4. Le client 2 répond au serveur.
5. Le client 1 reçoit le message.
6. Et ainsi de suite…

Nous avons déjà illustré comment programmer un client qui répond toujours dès qu'il reçoit un message. Ce client forme une partie du ping-pong. Nous avons donc besoin d'un client qui commence par envoyer un message dès qu'une connexion est établie. Nous souhaitons aussi mesurer le temps écoulé et afficher la vitesse des transactions.

CodePython/chapitre15/benchmark/client.py

```
 1. import websockets
 2. import time
 3.
 4. async def client1():
 5.     async with websockets.connect('ws://localhost:8080') as websocket:
 6.         round_trips = 0
 7.         start = time.time_ns()
 8.         await websocket.send('allo')
 9.         while True:
10.             response = await websocket.recv()
11.             round_trips += 1
12.             if(round_trips % 65536 == 0):
13.                 end = time.time_ns()
14.                 duration = (end - start)/1000000000
15.                 print("rate: ",round_trips/duration," round trips per second")
16.             await websocket.send('allo!')
17.
18. async def client2():
19.     async with websockets.connect('ws://localhost:8080') as websocket:
20.         while True:
21.             response = await websocket.recv()
22.             await websocket.send('allo!')
```

La fonction `client1()` est définie comme une fonction asynchrone qui se connecte à un serveur WebSocket en utilisant l'adresse `ws://localhost:8080`. Une fois la connexion établie, la fonction envoie le message `allo!` au serveur et attend une réponse. Si une réponse est reçue, elle est stockée dans la variable `response`. Le code continue à envoyer le message `allo!` au serveur et à attendre une réponse tant que la boucle `while` est vraie. La variable `round_trips` est utilisée pour compter le nombre de tours de boucle effectués. Si `round_trips` est un multiple de 65536, la durée de la boucle est calculée et la vitesse de la boucle est affichée en utilisant la fonction `print()`.

La fonction `client2()` est également définie comme une fonction asynchrone qui se connecte à un serveur WebSocket en utilisant l'adresse `ws://localhost:8080`. Une fois la connexion établie, la fonction attend une réponse du serveur et envoie le message `allo!` au serveur.

En résumé, ce code établit une connexion WebSocket avec un serveur et envoie des messages au serveur en utilisant la bibliothèque `websockets`. La fonction `client1()` envoie des messages `allo!` au serveur et affiche la vitesse de la boucle toutes les 65536 boucles. La fonction `client2()` attend simplement une réponse du serveur et envoie le message `allo!` au serveur. Il reste maintenant à exécuter les coroutines. Nous pouvons y arriver en ajoutant une nouvelle fonction :

```
1. import asyncio
2. async def main():
3.     task1 = asyncio.create_task(client1())
4.     task2 = asyncio.create_task(client2())
5.     await asyncio.gather(task1, task2)
6.
7. asyncio.run(main())
```

La fonction `main()` est définie comme une fonction asynchrone qui crée deux tâches asynchrones en utilisant la fonction `asyncio.create_task()`. La première tâche est créée en appelant la fonction `client1()`, et la deuxième tâche est créée en appelant la fonction `client2()`. Les deux tâches sont ensuite exécutées simultanément en utilisant la fonction `asyncio.gather()`. La fonction `asyncio.run()` est utilisée pour exécuter la fonction `main()`. La fonction `main()` peut être combinée avec les fonctions `client1()` et `client2()` dans un seul fichier nommé `client.py`.

Il reste à écrire le serveur qui exploite le module `sanic` à cette fin. Pour l'installer avec pip :

```
pip install sanic
```

Le serveur prend ensuite la forme suivante. Le code est dans le fichier `server.py`.

CodePython/chapitre15/benchmark/server.py

```
1. import sanic
2.
3. app = sanic.Sanic(__name__)
4.
```

```
 5. connected = set()
 6.
 7. @app.websocket('/')
 8. async def sendToOthers(request, ws):
 9.     connected.add(ws)
10.     try:
11.         while True:
12.             message = await ws.recv()
13.             for client in connected:
14.                 if client is not ws:
15.                     await client.send(message)
16.     finally:
17.         connected.remove(ws)
18.
19. if __name__ == '__main__':
20.     app.run(host='0.0.0.0', port=8080)
21.
```

La fonction `sendToOthers()` est définie comme une fonction asynchrone qui est appelée chaque fois qu'un client se connecte au serveur WebSocket. La fonction ajoute le client à l'ensemble `connected` et commence à écouter les messages entrants. Si un message est reçu, la fonction envoie le message à tous les clients connectés, sauf à celui qui a envoyé le message. La fonction `app.run()` est utilisée pour exécuter le serveur WebSocket sur l'adresse IP 0.0.0.0 et le port 8080.

Pour tester ce code, vous devez lancer le script Python `server.py`. Puis, sans l'interrompre, vous pouvez lancer le script `client.py`. Naturellement, il faut que le port 8080 soit disponible sur votre machine. Vous pouvez changer, dans le script du client et dans le script du serveur le port 8080 pour un autre port comme par exemple 8087. Après un long moment, le temps nécessaire pour faire 65536 boucles, le client devrait afficher une indication de la vitesse. Vous pourriez, par exemple, voir ce message qui indique que votre machine peut effectuer plus de 3000 boucles complètes du client 1 au client 2 :

```
rate:  3087.5183011639106  round trips per second
rate:  3093.7954992488803  round trips per second
```

Vous pouvez mettre fin au serveur et au client en tapant `ctrl-c` alors que vous êtes dans la console.

Exercice. Écrivez un serveur Web qui affiche une page Web changeant de couleur régulièrement selon les indications du serveur. La nouvelle couleur doit être communiquée par le serveur.

Solution. Les lignes 1 à 3 importent les modules nécessaires pour le programme : `sanic` pour le framework web, `random` pour la génération

aléatoire de couleurs, et `asyncio` pour la programmation asynchrone. La ligne 11 crée une instance de l'application `sanic`, en lui passant le nom du module courant (`__name__`) comme argument. La ligne 14 définit une liste de couleurs possibles, sous forme de chaînes de caractères représentant les noms des couleurs en anglais. Les lignes 17 à 21 définissent une fonction asynchrone `envoyer_couleur()`, qui prend en paramètre un objet `websocket` (`ws`) et qui envoie une couleur aléatoire au client via ce `websocket`. Pour cela, elle utilise la fonction `random.choice()` pour choisir une couleur dans la liste couleurs, et la méthode `ws.send()` pour envoyer la couleur au client. Le mot-clé `await` indique qu'il faut attendre que l'envoi soit terminé avant de continuer. Les lignes 24 à 50 définissent une route qui renvoie une page HTML au client, en utilisant le décorateur `@app.route("/")` pour associer la fonction `index()` à l'URL racine de l'application web. La fonction index prend en paramètre un objet `request` qui représente la requête HTTP du client, et renvoie un objet `sanic.response.html` qui contient le code HTML de la page web. Ce code HTML contient un titre, un en-tête, et un script JavaScript qui ouvre un `websocket` avec le serveur, définit une fonction pour changer la couleur de fond de la page, et écoute les messages envoyés par le serveur via le `websocket`. Quand le client reçoit un message du serveur, il récupère la couleur envoyée et appelle la fonction pour changer la couleur de la page. Les lignes 52 à 60 définissent une route `websocket` qui communique avec le client, en utilisant le décorateur `@app.websocket("/ws")` pour associer la fonction `websocket()` à l'URL `/ws` de l'application web. La fonction `websocket()` prend en paramètre un objet `request` qui représente la requête HTTP du client, et un objet `ws` qui représente le `websocket` ouvert avec le client. La fonction utilise une boucle `while True` pour répéter indéfiniment les actions suivantes : appeler la fonction `envoyer_couleur()` pour envoyer une couleur aléatoire au client, et attendre une seconde avec la fonction `asyncio.sleep()`. Le mot-clé `await` indique qu'il faut attendre que ces actions soient terminées avant de continuer. Les lignes 63 à 64 exécutent l'application `sanic`, en utilisant la méthode `app.run()` avec les arguments `host="localhost"` et `port=8000` pour indiquer l'adresse et le port du serveur. Notre programme définit une fonction asynchrone `lire_fichier()`, qui prend en paramètre le nom d'un fichier et qui l'ouvre en mode lecture, lit son contenu et l'affiche. Il définit ensuite une fonction principale `main()`, qui crée deux tâches asynchrones pour lire deux fichiers différents, `fichier1.txt` et `fichier2.txt`, et qui attend que les deux tâches soient terminées. Enfin, il exécute la fonction principale avec la fonction `asyncio.run()`, qui lance la boucle d'événements d'`asyncio`, qui gère l'exécution des tâches asynchrones.

CodePython/chapitre15/Exercice2.py

```python
# Importer le module sanic, qui est un framework web rapide et asynchrone
import sanic

# Importer le module random, qui permet de générer des nombres aléatoires
import random

# Importer le module asyncio, qui nous permet de faire des pauses
import asyncio

# Créer une instance de l'application sanic
app = sanic.Sanic(__name__)

# Définir une liste de couleurs possibles
couleurs = ["red", "green", "blue", "yellow", "pink", "orange"]

# Définir une fonction asynchrone qui envoie une couleur aléatoire au client via websocket
async def envoyer_couleur(ws):
    # Choisir une couleur aléatoire dans la liste
    couleur = random.choice(couleurs)
    # Envoyer la couleur au client
    await ws.send(couleur)

# Définir une route qui renvoie une page HTML avec un script qui ouvre un websocket
@app.route("/")
async def index(request):
    return sanic.response.html("""
    <html>
        <head>
            <title>Changer la couleur de la page</title>
        </head>
        <body>
            <h1>Changer la couleur de la page</h1>
            <script>
                // Créer un objet websocket qui se connecte au serveur
                var ws = new WebSocket("ws://localhost:8000/ws");
                // Définir une fonction qui change la couleur de fond de la page
                var changer_couleur = function(couleur) {
                    document.body.style.backgroundColor = couleur;
                }
                // Quand le websocket reçoit un message du serveur
                ws.onmessage = function(event) {
                    // Récupérer la couleur envoyée par le serveur
                    var couleur = event.data;
                    // Appeler la fonction qui change la couleur de la page
                    changer_couleur(couleur);
                }
            </script>
        </body>
    </html>
    """)

# Définir une route websocket qui communique avec le client
@app.websocket("/ws")
async def websocket(request, ws):
    # Tant que le websocket est ouvert
    while True:
        # Appeler la fonction qui envoie une couleur aléatoire au client
        await envoyer_couleur(ws)
        # Attendre une seconde
        await asyncio.sleep(1)

# Lancer l'application sanic
if __name__ == "__main__":
    app.run(host="localhost", port=8000)
```

14.4 Exemple d'application

Pour illustrer WebSocket, nous pourrions utiliser plusieurs types d'applications :

1. Un outil de gestion de projet qui utilise WebSocket pour fournir des mises à jour en temps réel aux utilisateurs. Dès qu'un utilisateur ajoute un élément à un projet, tous les utilisateurs reçoivent une notification.
2. Un outil de clavardage en ligne qui permet aux utilisateurs de discuter, d'échanger des messages, etc.
3. Un outil d'édition collaborative de documents. Dès qu'un des utilisateurs a modifié un texte, tous les utilisateurs voient le texte modifié.

Nous allons plutôt concevoir une application de dessin collaboratif. Nous voulons créer un tableau blanc commun et permettre à tous les utilisateurs de le modifier, en temps réel. Afin de garder l'application simple, une seule couleur (bleu) sera permise et une seule grille de 8 par 8. La Figure 16 donne un exemple du résultat souhaité.

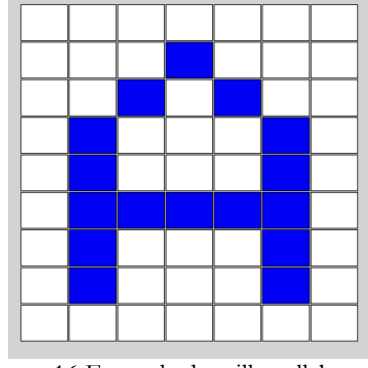


Figure 16 Exemple de grille collaborative

Nous allons utiliser le module `sanic` de nouveau. Tout comme Flask, `Sanic` permet de créer des serveurs Web conventionnels, en plus d'offrir les fonctions `WebSocket` nécessaires. Si nous disposons d'un fichier HTML nommé `index.html`, nous pouvons offrir le fichier HTML aux visiteurs du site avec ce script Python :

```
 1. from sanic import Sanic
 2. from sanic.response import html, file
 3.
 4. app = Sanic(__name__)
 5.
 6. @app.route('/')
 7. async def index(request):
 8.     return await file('index.html')
 9. if __name__ == '__main__':
10.     app.run()
11.
```

Ce code utilise le framework Sanic pour créer un serveur web. Il définit une application Sanic nommée `app` et crée une route pour l'URL racine `/`. Lorsque l'URL racine est demandée, la fonction `index()` est appelée. Cette fonction renvoie le contenu du fichier `index.html`. La dernière ligne `app.run()` lance l'application `Sanic`.

Nous souhaitons ajouter à notre serveur une route WebSocket. Le serveur devrait garder une copie de l'état de l'image et la mettre à jour lorsqu'un client lui transmet une nouvelle description. Le serveur doit aussi annoncer aux autres clients que l'image a été modifiée. Pour décrire l'image, nous n'avons besoin que de 64 charactères, un par élément de la grille. Par convention, la lettre `w` désigne le blanc (*white*) et `b` le bleu (*blue*). L'état de l'image est représenté avec une seule chaîne de caractères qui comprend 64 charactères `w/b` séparés par des virgules. Les huit premiers correspondent à la première rangée, et ainsi de suite. Au départ, le tableau est toujours blanc. En Python, l'état est un tableau : `state = ['w']*64`. Ce tableau esy converti en chaîne de caractère avec l'expression : `",".join(state)`. Finalement, on peut convertir une chaîne de caractères message en tableau avec cette expression : `message.split(',')`.

```
 1. from sanic import Sanic
 2.
 3. app = Sanic(__name__)
 4.
 5. connected = set()
 6. state = ['w']*64
 7.
 8. @app.websocket('/ws')
 9. async def sendToOthers(request, ws):
10.     connected.add(ws)
11.     global state
12.     await ws.send(",".join(state))
13.     try:
14.         while True:
15.             message = await ws.recv()
16.             state = message.split(',')
17.             for client in connected:
18.                 if client is not ws:
19.                     await client.send(message)
20.     finally:
21.         connected.remove(ws)
22.
23. if __name__ == '__main__':
24.     app.run()
```

Ce code définit une application Sanic nommée `app` et crée une route pour la connexion WebSocket à l'URL `/ws`. Lorsqu'une connexion WebSocket est établie, la fonction `sendToOthers()` est appelée. Cette fonction ajoute

la connexion WebSocket à un ensemble de connexions actives, envoie l'état actuel du serveur à la connexion WebSocket et attend des messages entrants. Lorsqu'un message est reçu, la fonction met à jour l'état du serveur et envoie le message à toutes les connexions actives, sauf celle qui a envoyé le message. La dernière ligne `app.run()` lance l'application Sanic.

Prenez en note que la variable globale `state` peut être réassignée au sein de la fonction `sendToOthers()`. Il peut y avoir plusieurs instances de la fonction `sendToOthers()` en cours d'exécution, mais il ne peut y avoir qu'une seule variable `state`.

Il ne reste plus qu'à combiner nos deux serveurs dans un seul script Python. Heureusement, ce n'est pas difficile :

CodePython/chapitre15/boxes/server.py

```
 1. from sanic import Sanic
 2. from sanic.response import html, file
 3.
 4. app = Sanic(__name__)
 5.
 6. @app.route('/')
 7. async def index(request):
 8.     return await file('index.html')
 9.
10. connected = set()
11. state = ['w']*64
12.
13. @app.websocket('/ws')
14. async def sendToOthers(request, ws):
15.     connected.add(ws)
16.     global state
17.     await ws.send(",".join(state))
18.     try:
19.         while True:
20.             message = await ws.recv()
21.             state = message.split(',')
22.             for client in connected:
23.                 if client is not ws:
24.                     await client.send(message)
25.     finally:
26.         connected.remove(ws)
27.
28. if __name__ == '__main__':
29.     app.run()
```

Concernant le fichier HTML, son corps comporte 64 éléments correspondant aux éléments de la grille (de classe `box`), ainsi que d'un

élément utilisé pour indiquer l'état de la connexion (ayant comme l'identifiant `message`). Le fichier HTML aura donc cette forme :

```
    <body>
        <div class="wrapper">
            <div class="box"></div>
            <div class="box"></div>
             …
            <div class="box"></div>
            <div class="box"></div>
        </div>
        <div id="message"></div>
    </body>
```

Au lieu de créer autant de HTML, un script JavaScript pourrait remplir le corps du document, mais nous voulons garder l'exemple aussi simple que possible.

CSS est exploité pour obtenir l'apparence voulue. Mettons d'abord en forme la page elle-même :

```
            body {
                background: #d3d3d3;
            }
```

Cet extrait de code définit la couleur de fond de l'élément `body` en gris clair. La propriété `background` définit la couleur de fond d'un élément HTML. La valeur de la propriété `background` peut être spécifiée de différentes manières, telles que l'utilisation d'un nom de couleur, d'une valeur hexadécimale, d'une valeur RGB ou d'une valeur HSL 1. Nous avons ici choisi le gris clair.

Nous pouvons maintenant mettre en forme l'espace réservé aux messages avec ce code CSS :

```
            #message {
                margin-left: auto;
                margin-right: auto;
                width: 30vw;
                padding: 2em;
            }
```

Ce code CSS définit les propriétés de style pour un élément HTML ayant l'identifiant `message`. Voici ce que chaque propriété signifie :

- `margin-left: auto;` et `margin-right: auto;` : Ces propriétés définissent la marge gauche et droite de l'élément `message` en tant que `auto`. Cela signifie que la marge gauche et droite de l'élément seront

automatiquement ajustées pour centrer l'élément horizontalement dans son conteneur parent.

- `width: 30vw;` : Cette propriété définit la largeur de l'élément `message` à 30% de la largeur de son conteneur parent. La valeur `vw` signifie `viewport width`, ce qui signifie que la largeur de l'élément sera calculée en fonction de la largeur de l'écran de l'utilisateur.
- `padding: 2em;` : Cette propriété définit l'espacement intérieur de l'élément `message` à 2 fois la taille de la police actuelle. Cela signifie que le texte à l'intérieur de l'élément sera espacé de 2 fois la taille de la police.

Le CSS suivant met en forme les éléments de la grille (`box`),:

```
1. .box {
2.  flex-grow: 1;
3.  width: 12.5%;
4.  border: 1px solid black;
5.  background-color: white;
6. }
```

Ce code CSS définit les propriétés de style pour les éléments HTML de classe `box`. Voici ce que chaque propriété signifie:

- `flex-grow: 1;` : Cette propriété définit le facteur d'expansion de l'élément `box` selon sa dimension principale. Elle indique la quantité d'espace restant que l'élément devrait consommer dans un conteneur flexible relativement à la taille des autres éléments du même conteneur. La dimension principale correspond à la hauteur ou à la largeur de l'élément selon la valeur de `flex-direction`. Si tous les éléments voisins possèdent le même facteur d'expansion, ils recevront tous la même part d'espace. La valeur `1` signifie que l'élément `box` peut s'étendre pour remplir l'espace disponible.
- `width: 12.5%;` : Cette propriété définit la largeur de l'élément `box` à 12,5% (ou 1/8) de la largeur de son conteneur parent. La valeur `12.5%` signifie que la largeur de l'élément sera calculée en fonction de la largeur de l'écran de l'utilisateur.
- `border: 1px solid black;` : Cette propriété définit la bordure de l'élément `box` à 1 pixel d'épaisseur et de couleur noire.
- `background-color: white;` : Cette propriété définit la couleur de fond de l'élément `box` à blanc.

Il reste maintenant à mettre en forme l'élément contenant la grille (classe `wrapper`) :

```
.wrapper {
    margin-left: auto;
    margin-right: auto;
    display: flex;
    width: 30vw;
    height: 30vw;
    gap: 1px;
    flex-wrap: wrap;
}
```

Voici ce que chaque propriété signifie:

- `margin-left: auto;` et `margin-right: auto;` : Ces propriétés définissent la marge gauche et droite de l'élément `wrapper` en tant que `auto`. Cela signifie que la marge gauche et droite de l'élément seront automatiquement ajustées pour centrer l'élément horizontalement dans son conteneur parent.
- `display: flex;` : Cette propriété définit l'élément `wrapper` comme un conteneur flexible. Les éléments enfants de `wrapper` peuvent être disposés horizontalement ou verticalement, selon la valeur de la propriété `flex-direction`.
- `width: 30vw;` : Cette propriété définit la largeur de l'élément `wrapper` à 30% de la largeur de son conteneur parent. La valeur `vw` signifie `viewport width`, ce qui signifie que la largeur de l'élément sera calculée en fonction de la largeur de l'écran de l'utilisateur.
- `height: 30vw;` : Cette propriété définit la hauteur de l'élément `wrapper` à 30% de la hauteur de son conteneur parent. La valeur `vw` signifie `viewport width`, ce qui signifie que la hauteur de l'élément sera calculée en fonction de la hauteur de l'écran de l'utilisateur.
- `gap: 1px;` : Cette propriété définit l'espace entre les éléments enfants de `wrapper` à 1 pixel.
- `flex-wrap: wrap;` : Cette propriété définit si les éléments enfants de `wrapper` doivent être disposés sur une seule ligne ou sur plusieurs lignes. La valeur `wrap` signifie que les éléments enfants peuvent être disposés sur plusieurs lignes si nécessaire.

Finalement, il ne nous reste plus qu'à écrire notre code JavaScript. Il faut commencer par créer une connexion WebSocket avec le serveur. L'expression est la suivante :

```
let socket = new WebSocket("ws://"+window.location.host+"/ws");
```

Ce code JavaScript crée un objet WebSocket nommé `socket` qui se connecte à un serveur WebSocket en utilisant l'URL `"ws://"+window.location.host+"/ws"`. Voici ce que chaque partie de l'URL signifie:

- `ws://` : Cela indique que la connexion est établie en utilisant le protocole WebSocket.
- `window.location.host` : Cela renvoie l'hôte et le numéro de port de la page actuelle. Par exemple, si la page actuelle est `http://example.com:8080/index.html`, alors `window.location.host` renverra `example.com:8080`.
- `/ws` : Cela est ajouté à la fin de l'URL pour spécifier le chemin du point de terminaison WebSocket sur le serveur.

Il est nécessaire de capturer l'état actuel de la grille. Nous pouvons faire le tour des éléments et vérifier leur couleur, pour combiner le tout en chaîne de caractères :

```
1. function getstatus() {
2.   let boxes = document.querySelectorAll(".box");
3.   let data = Array.from(boxes, function (box) {
4.     return box.style.background[0];
5.   }).toString();
6.   return data;
7. }
```

Le code JavaScript que vous avez fourni définit une fonction nommée `getstatus()` qui renvoie une chaîne de caractères représentant les couleurs de fond des éléments HTML ayant la classe `box`. Voici ce que chaque partie de la fonction signifie :

- `let boxes = document.querySelectorAll(".box");` : Cette ligne de code sélectionne tous les éléments HTML ayant la classe "box" et les stocke dans une variable nommée "boxes".
- `let data = Array.from(boxes, function (box) { return box.style.background[0]; }).toString();` : Cette ligne de code crée un tableau à partir des éléments stockés dans la variable `boxes`. Pour chaque élément, la couleur de fond est extraite et stockée dans le tableau. Ensuite, le tableau est converti en une chaîne de caractères à l'aide de la méthode `toString()` et stocké dans la variable `data`.
- `return data;` : Cette ligne de code renvoie la chaîne de caractères représentant les couleurs de fond des éléments ayant la classe `box`.

Dès que la connexion WebSocket est ouverte, nous souhaitons reconnaître les événements de clic de souris sur chaque élément de la grille. Lorsqu'un

clic est observé, nous devons changer la couleur de l'élément de la grille et envoyer au serveur une description du nouvel état. Le serveur communiquera ensuite aux clients le nouvel état.

```
socket.addEventListener("open", (event) => {
  document.getElementById("message").innerHTML = "ok";
  let boxes = document.querySelectorAll(".box");

  Array.from(boxes, function (box) {
    box.addEventListener("click", function (event) {
      if (event.target.style.background == "blue") event.target.style.background = "white";
      else event.target.style.background = "blue";
      let data = getstatus();
      socket.send(data);
      document.getElementById("message").innerHTML = "transmis";

    });
  });
});
```

Ce code JavaScript ajoute un gestionnaire d'événements à l'objet WebSocket nommé `socket`. Le gestionnaire d'événements est déclenché lorsque la connexion WebSocket est établie avec succès. Voici ce que chaque partie du code signifie:

- `socket.addEventListener("open", (event) => { … });` : Cette ligne de code ajoute un gestionnaire d'événements à l'objet WebSocket nommé `socket`. Le gestionnaire d'événements est déclenché lorsque la connexion WebSocket est établie avec succès. La fonction fléchée qui suit la chaîne `open` est appelée lorsque l'événement `open` est déclenché.
- `document.getElementById("message").innerHTML="ok";` : Cette ligne de code assigne le contenu HTML de l'élément ayant l'ID `message` le texte `ok`.
- `let boxes = document.querySelectorAll(".box");` : Cette ligne de code sélectionne tous les éléments HTML ayant la classe `box` et les stocke dans une variable nommée `boxes`.
- `Array.from(boxes, function (box) { … });` : Cette ligne de code crée un tableau à partir des éléments stockés dans la variable `boxes`. Pour chaque élément, un gestionnaire d'événements `click` est ajouté. Lorsque l'événement `click` est déclenché, la couleur de fond de l'élément est modifiée en alternance entre bleu et blanc. Ensuite, la fonction `getstatus()` est appelée pour récupérer les couleurs de fond de tous les éléments ayant la classe `box`. Les couleurs sont ensuite envoyées au serveur WebSocket à l'aide de la méthode `send()`. Enfin, le contenu HTML de l'élément ayant l'ID `message` devient `transmis`.

Il ne nous reste plus qu'à gérer la réception des messages du serveur. Dans de tels cas, il faut rapidement mettre à jour les couleurs de la grille.

```
socket.addEventListener("message", (event) => {
  document.getElementById("message").innerHTML = "reçu";
  let boxes = document.querySelectorAll(".box");
  let data = event.data.split(",");
  Array.from(boxes, function (box, index) {
  if (data[index] == "b") box.style.background = "blue";
    else box.style.background = "white";
  });
});
```

Ce code JavaScript utilise l'API WebSocket pour écouter les messages envoyés par un serveur. Lorsqu'un message est reçu, la fonction fléchée passée en tant que deuxième argument de `addEventListener()` est appelé. Cette fonction met à jour le contenu de l'élément HTML avec l'ID `message` en utilisant le texte `reçu`. Ensuite, elle récupère tous les éléments HTML avec la classe `box` et les stocke dans la variable `boxes`. Le message reçu est ensuite divisé en un tableau de chaînes en utilisant la virgule comme séparateur et stocké dans la variable `data`. Enfin, la méthode `Array.from()` est utilisée pour itérer sur chaque élément de boxes et mettre à jour la couleur de fond de chaque élément en fonction de la valeur correspondante dans le tableau data. Si la valeur est `b`, la couleur de fond est définie sur bleu, sinon elle est définie sur blanc.

Nous avons finalement le document HTML suivant :

CodePython/chapitre15/boxes/index.html

```
<html>
  <head>
    <meta charset="utf-8">
    <script>
      let socket = new WebSocket("ws://"+window.location.host+"/ws");

      function getstatus() {
        let boxes = document.querySelectorAll(".box");
        let data = Array.from(boxes, function (box) {
          return box.style.background[0];
        }).toString();
        return data;
      }

      socket.addEventListener("open", (event) => {
        document.getElementById("message").innerHTML = "ok";
        let boxes = document.querySelectorAll(".box");

        Array.from(boxes, function (box) {
          box.addEventListener("click", function (event) {
            if (event.target.style.background == "blue") event.target.style.background =
"white";
            else event.target.style.background = "blue";
            let data = getstatus();
            socket.send(data);
            document.getElementById("message").innerHTML = "transmis";

          });
```

```
        });
      });

      socket.addEventListener("message", (event) => {
        document.getElementById("message").innerHTML = "reçu";
        let boxes = document.querySelectorAll(".box");
        let data = event.data.split(",");
        Array.from(boxes, function (box, index) {
          if (data[index] == "b") box.style.background = "blue";
          else box.style.background = "white";
        });
      });

      socket.addEventListener("error", (event) => {
        document.getElementById("message").innerHTML = "error" + event.data;
      });

      socket.addEventListener("close", (event) => {
        document.getElementById("message").innerHTML = "fermé" + event.data;
      });

    </script>
    <style>
            body {
                background: #d3d3d3;
            }
            #message {
                margin-left: auto;
                margin-right: auto;
                width: 30vw;
                padding: 2em;
            }

            .wrapper {
                margin-left: auto;
                margin-right: auto;
                display: flex;
                width: 30vw;
                height: 30vw;
                gap: 1px;
                flex-wrap: wrap;
            }

            .box {
                flex-grow: 1;
                width: 12.5%;
                border: 1px solid black;
                background-color: white;
            }
        </style>
    </head>
    <body>
        <div class="wrapper">
            <div class="box"></div>
            <div class="box"></div>
…
            <div class="box"></div>
            <div class="box"></div>
            <div class="box"></div>
            <div class="box"></div>
        </div>
        <div id="message"></div>
    </body>
</html>
```

Si vous placer le fichier HTML en question, nommé `index.html`, dans le même répertoire que le script Python, vous devriez pouvoir lancer le script Python pour démarrer le serveur. Prenez soin de vous assurer qu'il n'y a pas d'autre serveur utilisant le port 8000. Par la suite, ouvrez la page `http://127.0.0.1:8000` dans un navigateur plus d'une fois. Vous pouvez utiliser un navigateur, mais l'application a surtout un intérêt si vous

l'ouvrez dans deux fenêtres en même temps. Vous devriez ensuite être capable de dessiner dans l'une ou l'autre des fenêtres tout en voyant le même résultat dans toutes les fenêtres.

Il peut être intéressant de réfléchir à ce qui se passe si vous cliquez en même temps sur la grille dans deux fenêtres différentes. Dans un tel cas, le résultat n'est pas bien défini : il y aura concurrence entre les deux fenêtres.

14.5 Conclusion

La technologie WebSocket vous permet de concevoir des applications web facilitant la collaboration en temps réel entre les utilisateurs. Comme vous avez pu le constater, il n'est pas beaucoup plus difficile de concevoir une application WebSocket qu'une application Web conventionnelle en Python.