Chapitre 0

Introduction

Python

Le contexte

De nos jours, dans une société où la digitalisation est devenue incontournable, le **calcul scientifique** est présent (dans une mesure plus ou moins importante et d'une manière plus ou moins transparente) dans la plupart des activités professionnelles (l'ingénierie, les sciences fondamentales, la médecine, les sciences humaines et sociales, les beaux-arts, le divertissement et les jeux, etc.), voire non professionnelles.

Le **calcul scientifique** utilise de plus en plus l'informatique qui intervient de manière essentielle dans la résolution de problèmes par des méthodes numériques, la modélisation et la simulation numérique, l'analyse et le traitement des données, etc.

Il convient de souligner le fait que l'**informatique** :

- ne se réduit pas seulement à l'algorithmique ou à la programmation ;
- représente un domaine extrêmement vaste où la science est difficilement séparable de la technologie et la théorie est étroitement liée à la pratique.

Le cours d'Informatique et calcul scientifique (ICS)

Les principaux **objectifs du cours d'ICS** :

- comprendre les fondements de la programmation en **langage Python** ;
- acquérir une **pensée algorithmique** ;
- comprendre et utiliser des **méthodes numériques élémentaires**.

A la fin de l'année, l'étudiant sera à même de concevoir et d'analyser des programmes qui font intervenir des algorithmes et des méthodes numériques codés en langage Python.

Le **contenu du cours d'ICS** est organisé en quatre parties :

1. **Programmation Python**
2. Algorithmique
3. Bibliothèques scientifiques et graphiques Python
4. Méthodes numériques

La **bibliographie** (facultative et minimale) pour la **première partie** :

- Documentation for Python 3 (https://docs.python.org/3/) ;
- G. Swinnen, Apprendre à programmer avec Python 3, Eyrolles, 2012, ISBN : 9782212134346 ;
- J. Guttag, Introduction to Computation and Programming Using Python : With Application to Understanding Data, Second Edition, MIT Press, 2016, ISBN : 9780262529624 ;
- J. P. Forestier, Programming with Python 3, OSYX, 2017.

Chapitre 1

Notions générales et caractéristiques du langage Python

Python

Langage de programmation

Remarque : le contenu de ce premier chapitre sera de mieux en mieux compris au fur et à mesure de l'avancement du cours théorique et, surtout, des travaux pratiques associés.

Un **langage de programmation** est :

- un langage formel qui représente un code de communication entre un humain et un ordinateur ;
- destiné à écrire des programmes ;
- composé, comme les langages naturels, d'un **alphabet**, d'un **vocabulaire** ainsi que des règles de **grammaire** (définissant surtout la **syntaxe**) et d'une **sémantique** associée.

Les langages dits de **bas niveau** sont proches du processeur et au plus bas niveau on trouve le **langage machine** qui contient des instructions qui sont directement exécutables par un processeur spécifique (central ou graphique).

Ces langages sont très efficaces mais ils sont écrits pour et utilisés par des **ordinateurs spécifiques**.

Langage de programmation (suite)

Les langages dits de **haut niveau** utilisent une abstraction élevée et ignorent les détails spécifiques à un processeur, voire un ordinateur, en particulier.

Ces langages sont :

- plus proches de la logique humaine (voire du langage naturel) ;
- ne dépendent pas d'un certain type de processeur ou d'ordinateur.

En revanche, ils sont moins efficaces, mais il y a des techniques qui peuvent pallier cet inconvénient (*abstraction penality* en anglais).

Exemples de langages de programmation de **haut niveau** : **Python**, Java, Scala, C++, C#, etc.

En 1997, on comptait déjà environ 2'350 langages de programmation.

Programme - code - logiciel

Un **programme informatique** :

- est un ensemble d'instructions qui sont destinées à être exécutées par un ordinateur afin de réaliser une certaine tâche ;
- est écrit normalement (par un programmeur humain ou par un autre programme) dans un langage de programmation ;
- correspond le plus souvent à l'**implémentation** d'un certain **algorithme**.

Si le **programme** est :

- lisible par un humain (compétent), il s'appelle **code source** ;
- exécutable directement par l'ordinateur, il s'appelle **code machine**.

Un **logiciel** (*software* en anglais) est un **ensemble** de programmes (ou un seul programme) destiné à fournir un service informatique.

En général, un **programme** est écrit par un programmeur, il est de taille relativement réduite et destiné à une utilisation restreinte.

En revanche, un **logiciel** est écrit par une équipe de développeurs, a une taille consistante, est destiné à de nombreux utilisateurs et dispose d'une documentation adéquate.

Logiciels

Il y a des **logiciels** :

- **gratuits** ou payants ;
- **ouverts** (utilisant des standards et/ou formats ouverts) ou propriétaires ;
- **libres** (qui permet d'accéder au code source, d'étudier et d'adapter le fonctionnement, de redistribuer des copies et d'améliorer le logiciel et de publier les améliorations) ou pas.

Quelques exemples de **logiciels** :

- un système d'exploitation ;
- un navigateur (comme Firefox, Chrome, Edge, Safari, etc.) ;
- un antivirus (comme McAfee, Norton, Bitdefender, Panda, etc.) ;
- un logiciel de communication (comme Discord, Telegram, WhatsApp, Skype, Zoom, etc.) ;
- une suite bureautique (MS Office, LibreOffice, Google Documents, iWork, etc.) ;
- un logiciel photo (Photoshop, Gimp, Paint, Lightroom, etc.) ;
- une base de données (MySQL, SAP, Access, Oracle, PostgreSQL, MS SQL Server, etc.) ;
- un jeu vidéo.

Compilateur vs interpréteur

Le **compilateur** (est lui-même un programme qui) traduit (ou transforme) le **code source** écrit dans un certain langage de programmation (de haut niveau) en un code dit **code compilé** (ou code objet) écrit :

- soit en **langage machine** et, donc, exécutable directement par l'ordinateur ;
- soit en **code intermédiaire** (ou bytecode ou code objet) qui est destiné à un autre programme (nommé parfois machine virtuelle **VM** - *virtual machine* en anglais) qui a un interpréteur intégré.

L'opération de traduction faite par le **compilateur** et décrite ci-dessus s'appelle **compilation** (voir l'image à la page suivante) et elle :

- à lieu à un moment appelé couramment **compile time** ;
- réalise l'analyse syntaxique et l'analyse sémantique du code source et les éventuelles erreurs trouvées s'appellent "**erreurs à la compilation**" ;
- génère le **code compilé**.

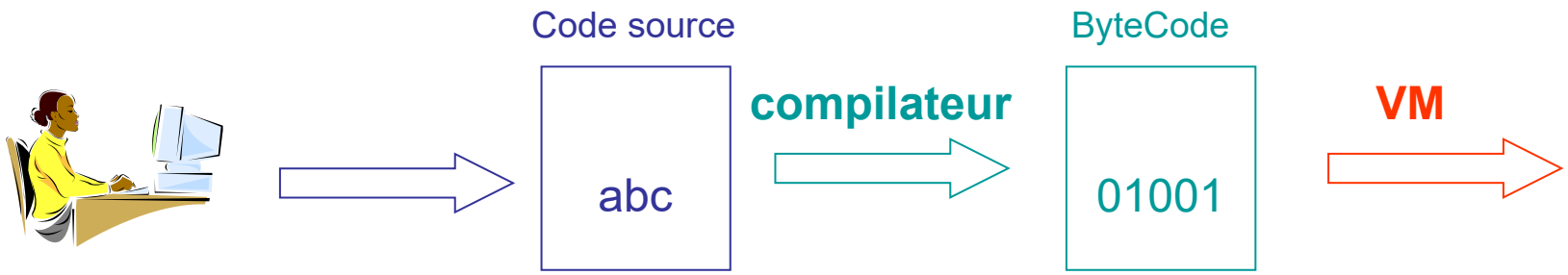
Un **interpréteur** :

- parcourt progressivement le **code source** ou le **code intermédiaire** ;
- fait une transformation similaire au compilateur (et produit surtout du code machine exécutable) mais exécute les instructions "à la volée" ;
- permet que l'exécution du code se réalise au fur et à mesure.

Exécution - runtime

Un **programme** réalise la tâche pour laquelle il a été conçu au moment où il est **exécuté** (directement ou après compilation et/ou interprétation) par le(s) processeur(s) de l'ordinateur et ce moment s'appelle couramment **runtime**.

Les éventuelles erreurs produites à l'exécution s'appellent "**erreurs à l'exécution**" et celles qui peuvent être traitées s'appellent couramment "**exceptions**".



IDE

Un **environnement de programmation intégré** (*Integrated Development Environment – IDE* en anglais) est un ensemble d'outils qui facilitent le travail des **programmeurs** (aka **développeurs** ou **codeurs**) afin d'augmenter la productivité dans la création des logiciels.

Plus précisément, un **IDE** prévoit au moins :

- un éditeur de texte (le plus souvent avec la coloration syntaxique – *syntax highlighting* en anglais, la complétion automatique - *code completion* en anglais, etc.) ;
- une façon simple (par exemple un clic sur un bouton) pour compiler, éditer de liens et/ou exécuter un programme ;
- un débuggeur en ligne ;
- d'autres outils (pour la création des interfaces graphiques, pour des tests automatiques, pour l'analyse du code, pour le contrôle des versions, etc.).

Certains **IDE** sont dédiés à un langage de programmation en particulier tandis que d'autres peuvent être utilisés avec de nombreux langages différents.

Langage Python

Guido van **Rossum** est le principal auteur du langage **Python**.

Il est assisté par une équipe de développeurs du noyau (*core developers* en anglais) qui ont un accès en écriture au dépôt de **CPython** (voir plus bas).

La **fondation Python Software Foundation (PSF)** :

- est une société à but non lucratif qui détient les droits de propriété intellectuelle du langage de programmation Python ;
- a pour mission de "promouvoir, protéger et faire progresser le langage de programmation **Python**, et de soutenir et de faciliter la croissance d'une communauté diversifiée et internationale de programmeurs **Python**".

La version de référence du **Python** est écrite en **langage C** et s'appelle **CPython**.

Implémentations et distribution Python

Il y a plusieurs implémentations et distributions **Python** :

- **CPython** est l'implémentation de référence de Python écrite en **langage C**, produite par le groupe central responsable de toutes les décisions de haut niveau concernant le langage Python et disponible à l'adresse **python.org** ;
- **Anaconda** est une distribution gratuite et open-source des langages de programmation **Python** et **R** pour le calcul scientifique produite par Anaconda Inc. et qui vise à simplifier la gestion et le déploiement des packages ;
- **PyPy** est un interpréteur Python écrit en Python, qui remplace l'interpréteur CPython et utilise une compilation juste-à-temps (**JIT**) pour accélérer l'exécution des programmes Python (avec des gains de performances significatifs) ;
- **Jython** est une implémentation de Python pour la **JVM** (Java Virtual Machine) et qui peut accéder à l'environnement de développement **Java** ;
- **IronPython** est une implémentation de Python pour la **CLR** (Common Language Runtime) et qui peut accéder à l'environnement de développement **.Net**.

IDE libres qui supportent Python

- **Jupyter Notebook** (Windows, Linux, MacOS) (intégré ou pas dans Anaconda) :
  - est destiné à la fois aux débutants et aux professionnels qui travaillent surtout dans la science des données (*Data Science*) ;
  - supporte plus de 40 langages de programmation (notamment les langages **Julia**, **Python** et **R** qui ont donné le nom de cette application web) ;
  - à part l'éditeur, est muni aussi d'outils éducationnels et de présentation ;
- **PyCharm** (Windows, Linux, MacOS X) contient une **API** (Application Programming Interface) qui peut être utilisée par les développeurs pour écrire leurs propres plugins Python pour qu'ils puissent étendre les fonctionnalités de base ;
- **Spyder** (Windows, Linux, MacOS) est écrit en Python pour Python et disponible via Anaconda ;
- **PyDev** est un plug-in pour Eclipse (Windows/Linux/MacOS) qui peut être utilisé avec Python, Jython et IronPython ;
- **Visual Studio Code** (Windows, Linux, MacOS) est un environnement open-source développé par Microsoft et qui supporte plusieurs langages de programmation, y compris Python ;
- **Xcode (**Mac OS X 10.5 ou plus récent) est un environnement gratuit développé par Apple et qui supporte une dizaine de langages de programmation, y compris Python ;
- **IDLE** (Windows/Linux/MacOS) est un éditeur par défaut fourni avec Python et dont le nom est une abréviation pour *Integrated Development and Learning Environment*.

Historique et caractéristiques du langage Python

Versions Python – Historique :

- début de l'implémentation en décembre 1989 ;
- version 1.0 - janvier 1994 ;
- version 2.0 - octobre 2000 ;
- version 3.0 - décembre 2008 ;
- version 3.9 - octobre 2020 ;
- version 3.10 - octobre 2021 ;
- version 3.10.7 - le 6 septembre 2022 (dernière version).

Python est un langage :

- **généraliste** (mais largement préféré aujourd'hui à d'autres langages dans certains domaines mentionnés à la fin de cette liste) ;
- de **haut niveau** ;
- **interprété** :
  - le code peut être (directement) interprété à l'exécution ;
  - cependant, le code peut être compilé (transformé en bytecode) avant l'exécution (ce qui améliore les performances).

Caractéristiques du langage Python (suite)

Python est un langage :

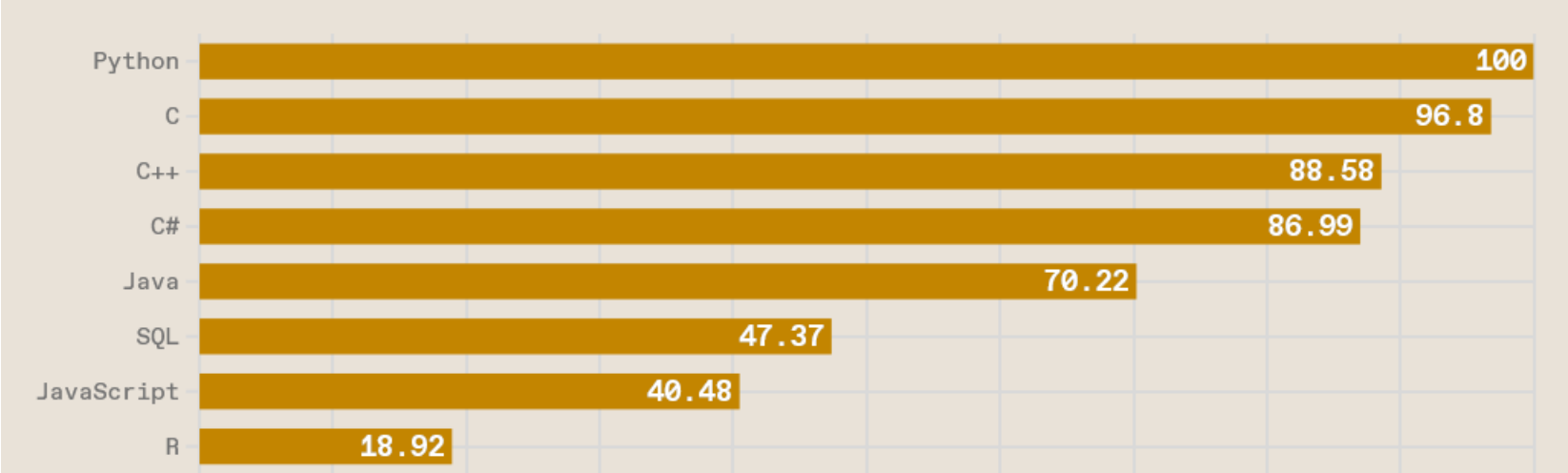
- qui utilise un **typage dynamique fort** :
  - pas de vérification de type à la compilation (donc pas de typage statique) ;
  - les types sont vérifiés à l'exécution ;
- **orienté objets** :
  - cependant, on peut écrire de petits scripts sans créer ni instancier de classes ;
  - permet de programmer aussi en style fonctionnel ;
- **indépendant de plateforme** (**cross-platform** en anglais) : Python fonctionne sur les principales plateforme hardware et avec les principaux systèmes d'exploitation ;
- utilisation des indentations pour structurer le code ;
- existence d'une multitude de bibliothèques (*libraries* en anglais) standard (écrites ou, au moins, accessibles en) Python et utilisables dans les plus divers domaines et notamment en :
  - science des données (*Data Science* en anglais) ;
  - apprentissage automatique (*Machine Learning* en anglais) ;
  - intelligence artificielle (*Artificial Intelligence* en anglais) ;
  - techniques d'exploration de données massives (*Data Mining* en anglais).

Python aujourd'hui

L'application **IEEE Spectrum Top Programming Languages** synthétise 11 métriques provenant de huit sources pour arriver (en pondérant et en combinant ces métriques) à un classement général de la **popularité des langages de programmation**.

Les sources couvrent des contextes qui incluent les échanges sur les médias sociaux, la production de code open source et les offres d'emploi.

Pour **l'année 2022**, **Python** est en tête de liste (devant les langages **C** et **C++**).



Python aujourd'hui (suite)

L'**index TIOBE** est calculé et mis à jour une fois par mois par l'entreprise **TIOBE Software BV** (basée aux Pays-Bas) et il mesure aussi la **popularité des langages de programmation**.

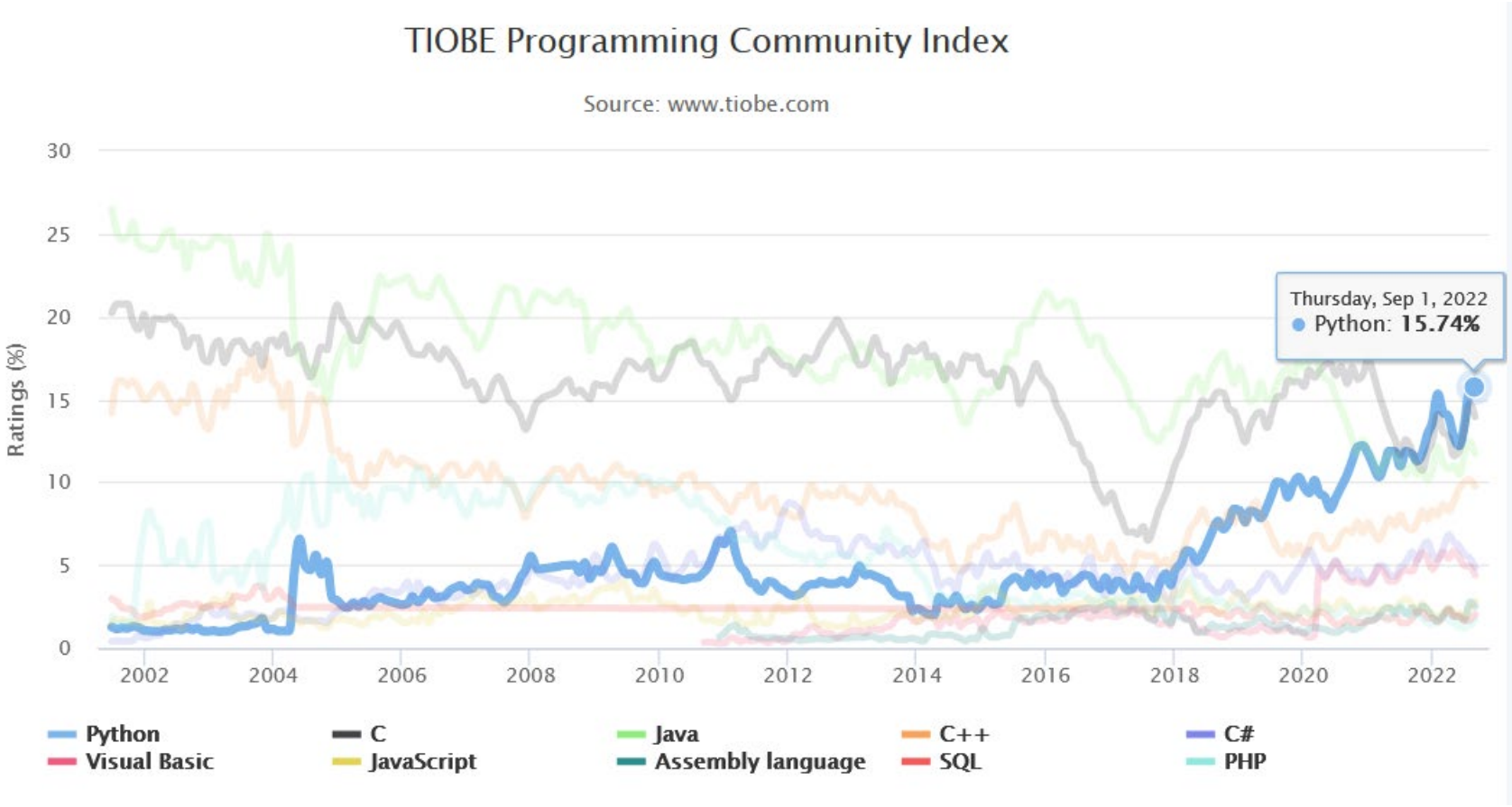
Le calcul est basé sur le nombre de pages web retournées par les principaux moteurs de recherche quand on recherche un certain langage de programmation et le classement prend en compte le nombre d'ingénieurs qualifiés dans le monde, de cours et de fournisseurs tiers.

Au mois de septembre **2022**, **Python** se place en première position (devant les langages **C** et **Java**).

Sep 2022	Sep 2021	Change	Programming Language	Ratings	Change
1	2	▲	Python	15.74%	+4.07%
2	1	▼	C	13.96%	+2.13%
3	3		Java	11.72%	+0.60%
4	4		C++	9.76%	+2.63%
5	5		C#	4.88%	-0.89%
6	6		Visual Basic	4.39%	-0.22%
7	7		JavaScript	2.82%	+0.27%
8	8		Assembly language	2.49%	+0.07%

Python aujourd'hui (suite)

Dans la figure ci-dessous, on peut voir l'historique de l'évolution de **la popularité des langages de programmation** et, en particulier, du langage **Python** selon l'**index TIOBE**.



Compléments

Motto : La programmation est la branche la plus difficile des mathématiques appliquées. (Edsger W. Dijkstra)

Remarque : on présente ci-dessous quelques éléments de vocabulaires utiles par la suite.

Informatique

L'**informatique** est définie dans le Larousse en ligne comme la " science du traitement automatique et rationnel de l'information considérée comme le support des connaissances et des communications" ou, encore, l'ensemble "des applications de cette science, mettant en œuvre des matériels (ordinateurs) et des logiciels".

Le terme **informatique** :

- a été utilisé pour la première fois en 1962 par Philippe Dreyfus, ancien directeur du Centre national de calcul électronique de Bull ;
- peut être vu comme une combinaison des termes "information" et "automatique" afin de désigner le "traitement automatique de l'information" ;
- a plusieurs équivalents anglais comme *informatics* ou *computer science*.

Informatique et ordinateur

Concernant le syntagme largement utilisé *computer science*, le professeur Hal Abelson du MIT mentionnait : " *Computer science is no more about computers than astronomy is about telescopes*".

Dans cet esprit, selon Wikipédia, l'**informatique** "est un domaine d'activité scientifique, technique, et industrielle concernant le traitement automatique de l'information par l'exécution de programmes informatiques par des machines …".

Ordinateur

L'**ordinateur** est défini dans le Larousse en ligne comme une " machine automatique de traitement de l'information, obéissant à des programmes formés par des suites d'opérations arithmétiques et logiques."

Le terme **ordinateur** :

- a été utilisé pour la première fois en 1955 par IBM France (suite à la suggestion du professeur de lettres Jacques Perret) ;
- peut être vu comme dérivant du mot "ordonnateur" (c'est-à-dire celui qui "met en ordre") ;
- a comme équivalent anglais *computer* (qui ne doit pas donc être traduit dans le sens mentionné ci-dessus par *calculateur* ou *calculatrice*).

Hardware et software

Hardware

Le **hardware** est un anglicisme qui désigne le matériel informatique dans son ensemble.

Le **hardware** correspond aux parties physiques (électroniques, électriques, magnétiques, mécaniques, etc.) de l'ordinateur comme : le boitier, la carte mère, le(s) processeur(s), d'autres cartes (son, vidéo, réseau, etc.), le clavier, l'écran, les unités de stockage de données (comme les disques HHD ou SSD, les barrettes de mémoires RAM, les clés USB), etc.

Software - Logiciels

Le **software** est un anglicisme qui désigne en français les logiciels dans leur ensemble.

Le **software** est une collection d'instructions et d'informations qui gèrent le travail d'un ordinateur, comme : des programmes, des bibliothèques (*libraries* en anglais), de la documentation en ligne, des données numériques, etc.

Système d'exploitation

Système d'exploitation

Un **système d'exploitation** (*operating system* en anglais) est un **logiciel** dit **système** qui gère le hardware et les ressources software de l'ordinateur, tout en fournissant les services et l'accès aux ressources nécessaires à d'autres programmes pour qu'ils puissent être exécutés par l'ordinateur.

Le **système d'exploitation**, appelé aussi **plateforme** (d'exploitation), offre l'interface de communication (graphique ou non) entre l'utilisateur et l'ordinateur.

Exemples de systèmes d'exploitation : **Windows**, **Mac OS**, **Linux**, Android, Unix, etc.

Pour les postes clients, on fait parfois la distinction entre un ordinateur **Mac** (qui est produit par la compagnie Apple Inc. et qui utilise MAC OS comme système d'exploitation) et tout autre ordinateur qui est nommé génériquement **PC** - *Personal Computer* en anglais – et qui utilise le plus souvent **Windows** ou **Linux** comme système d'exploitation.

Chapitre 2

Introduction à la programmation orientée objets

Python

Objets

Vu le modèle de données qu'il utilise, Python est un **langage orienté objets**.

En Python tout est **objet** dans le sens où les briques fondamentales de la conception de tout programme Python sont des objets.

Le langage Python réalise l'**abstraction de données** par l'intermédiaire des **objets** et tout programme représente les données par des **objets** ou par des **relations entre les objets**.

Un **objet** est une **entité** qui :

- peut être clairement identifiée, reconnue et manipulée ;
- correspond à une notion concrète ou abstraite ;
- a une certaine valeur ;
- a un certain **type**.

Exemples d'**objets** :

- un nombre entier, un nombre réel ou un nombre complexe ;
- une fonction mathématique ;
- un message, une photo ou un fichier.

Objets (suite)

Exemples d'**objets** (suite) :

- l'année de fabrication, la couleur ou le nombre de places d'une voiture ;
- le moteur, la boîte à vitesse ou la carrosserie d'une voiture ;
- une voiture (vue dans son ensemble) ;
- la capacité d'une voiture d'accélérer, de freiner ou de reculer ;
- le plan d'ingénieur utilisé pour construire un certain type de voiture ;
- un parc, une collection ou une liste de voitures.

Il y a des **objets** :

- **simples** (voire très simples) comme la hauteur ou la largeur d'une fenêtre affichée à l'écran durant un jeu vidéo ;
- **complexes** (voire très complexes) comme un jeu vidéo dans son ensemble - avec tout son design graphique et toutes ses interactions (par l'intermédiaire de la souris, du clavier, du joystick, de la voix, des gestes, du mouvement des yeux, etc.) avec le joueur local ou les joueurs en ligne.

Chaque **objet** doit avoir un certain **type** qui est lui-même un **objet**.

Classes et instances

Plus précisément, à un **même objet** peuvent correspondre (éventuellement) **plusieurs types** (si on prend en compte un mécanisme appelé **polymorphisme**) et, dans un tel cas, on dit que l'**objet** est **polymorphe**.

Prenons un exemple : une **maison** construite à partir d'un certain **plan d'architecte** est un **objet** mais aussi le plan utilisé pour la construction de la maison est lui-même un **objet**.

Dans un tel cas, on dit que le **plan d'architecte** est un **objet classe** qui est le type de l'**objet instance maison**.

Le plus souvent, de manière simplifiée mais moins précise, on se contente de dire que l'**objet** (qui correspond à la) **maison** est une **instance** de la **classe** (qui correspond au) **plan d'architecte**.

Dans cette optique, la **classe** est une sorte de moule à partir duquel on fabrique des **objets** (similaires mais pas forcément identiques) qui sont des **instances** de la **classe**.

La **construction** (ou la création) d'un objet selon le plan prévu par une **classe** est appelée **instanciation** de la **classe** et l'**objet** construit (ou créé) à partir de la **classe** est appelé une **instance** de la **classe**.

Attributs

Tout **objet** (soit objet **instance** soit objet **classe**) doit avoir une **identité unique** qui :

- permet de reconnaitre l'objet sans ambiguïté tout au long de son existence ;
- est précisée à la création de l'objet (et correspond à l'adresse où l'objet est stocké dans la mémoire de l'ordinateur) ;
- ne change plus après la création de l'objet.

De plus, assez souvent, un **objet** (surtout un objet **instance** d'une **classe**) est caractérisé par des **attributs** qui précisent :

- son **état** défini à l'aide des **attributs données** ou, tout simplement, des **données** (plus précisément, des "objets données" ou *data objects* en anglais) qui sont appelées (dans ce contexte) des **variables d'instance** ou des **champs** ;
- son **comportement** défini à l'aide des **attributs fonctions** ou, tout simplement, des **fonctions** (plus précisément, des "objets fonctions" ou *function objects* en anglais) qui sont appelées (dans ce contexte) des **méthodes d'instance**.

Objets - exemples

Exemples d'objets :

- **ma voiture** qui a
  - un **nom** ("myCar") ;
  - un certain **état** (type "Renault", année fabrication "2022", couleur "rouge", nbVitesses "6", etc.) ;
  - un certain **comportement** (accélérer, décélérer, freiner, changer de vitesses, etc.).

*Remarque : l'**identité** de **ma voiture** est donnée par la combinaison **unique** du numéro de châssis et du numéro de série du moteur.*

- **mon compte en banque** qui a :
  - une **référence** ("monComptePostal") ;
  - un certain **état** (type "personnel", montant "2000000", dernier dépôt "septembre2022", etc.) ;
  - un certain **comportement** (déposer, soustraire, bloquer, etc.).

*Remarque : l'**identité** **de mon compte en banque** est donnée par son numéro **unique** dans les registres de la banque.*

- une **variable** du programme qui correspond à un **certain** **étudiant du CMS** qui a :
  - un **nom** ("toto") ;
  - certains **champs** (adresse "Lausanne", section "CMS", note_ICS "5.5", etc.) ;
  - certaines **méthodes** (changer_adresse, afficher_section, lire_note_ICS, modifier_note_ICS, etc.).

*Remarque : l'**identité** de l'**étudiant** correspond à l'adresse **unique** où les informations qui le concernent sont stockées dans la mémoire de l'ordinateur.*

Modules et packages

Les **objets** construits à partir d'une **même classe** sont **similaires** mais **pas identiques**.

Attention : afin d'utiliser (en lecture et/ou en écriture) les **attributs** d'un objet (soit variables d'instance soit méthodes) en dehors de la classe où il est défini, il faut préfixer le **nom** d'un tel **attribut** par le **séparateur point** précédé par (le nom de) l'**objet** qui est concerné par cet **attribut**.

L'organisation du code Python est basée sur des **objets** nommés :

- **modules** qui regroupent des objets (le plus souvent des fonctions et des classes) afin de les rendre disponibles dans d'autres programmes (y compris dans l'interpréteur Python) ;
- **packages** qui sont des modules spéciaux qui regroupent d'autres modules (voire des sous-packages).

Les **modules** et les **packages** sont des **espaces de noms** et un programme qui en a besoin peut **importer** tout un tel espace (avec tout son contenu) ou seulement certains de ces éléments.

Exemples d'objets **modules** (ou simplement de modules) et d'objets **packages** (ou simplement de packages) :

- **sys** – module qui permet l'accès à certaines variables et fonctions liées étroitement à l'interpréteur ;
- **math** – module qui permet l'accès à des fonctions mathématiques (définies en langage C standard) ;
- **datetime** – module qui fournit des classes qui permettent la manipulation des dates et du "temps" ;
- **dateutil** - package qui est une extension du module standard **datetime**.

Fonctions et méthodes

Afin de simplifier le travail du programmeur, le langage Python met à sa disposition :

- un certain nombre d'**objets prédéfinis natifs** (*built-in* en anglais) qui sont disponibles (directement) dans tout programme Python ;
- un certain nombre d'**objets prédéfinis** dans des **modules** ou des **packages** et qui sont disponibles (seulement) après avoir été importés dans les programmes qui veulent les utiliser.

Il faut bien faire la distinction entre les **fonctions** et les **méthodes** :

- une **fonction** qui n'est pas une méthode est définie en dehors de toute classe et peut être appelée ensuite **directement** par son nom (à condition qu'il soit connu à l'endroit de l'appel) ;
- une **méthode** est une fonction "spéciale" qui est définie à l'intérieur (i.e. dans le corps) d'une classe et doit être ensuite appelée (en dehors de cette classe) en faisant **préfixer** son nom par le **séparateur point** précédé par :
  - le **nom** d'un **objet** dont le type est cette classe, pour une méthode dite **d'instance** ;
  - le **nom** de cette **classe** ou d'un **objet** dont le type est cette classe, pour une méthode dite **statique** ;
  - le **nom** de cette **classe** ou d'un **objet** dont le type est cette classe, pour une méthode dite **de classe**.

Classes et fonctions - exemples

Exemples de **classes** prédéfinies :

- **object** - classe racine ancêtre de toutes les classes (voir plus loin) ;
- **int** - classe dont les instances sont des objets qui correspondent à des nombres entiers ;
- **float** - classe dont les instances sont des objets qui correspondent à des nombres réels ;
- **complex** - classe dont les instances sont des objets qui correspondent à des nombres complexes ;
- **str** - classe dont les instances sont des objets qui correspondent à des chaînes de caractères.

Exemples de **fonctions** prédéfinies :

- **type()** - appelée avec la référence d'un objet comme argument, cette fonction retourne le **type** de l'objet argument ;
- **id()** - appelée avec la référence d'un objet comme argument, cette fonction retourne un entier représentant l'**identifiant** (unique et constant) de l'objet argument ;
- **pow()** – appelée avec deux argument (numériques), cette fonction retourne la valeur du premier argument élevé à la puissance précisée par le deuxième argument ;
- **int()** - appelée avec un argument (nombre ou chaîne de caractères), cette fonction (constructeur) crée et retourne un objet de type **int** ; appelée sans argument, elle retourne l'objet nombre entier 0 (zéro).

Héritage

En tant que langage orienté objets, Python assure la réutilisation et l'extension du code déjà écrit (et qui fonctionne correctement) par l'intermédiaire d'un mécanisme fondamental appelé **héritage**.

De plus, l'**héritage** simplifie et rend robustes la maintenance et la modification des programmes.

En fait, à la création d'une nouvelle classe appelée **classe dérivée**, on peut indiquer que le point de départ est une classe (voire plusieurs classes) déjà existante(s) appelée(s) **classe(s) de base**.

Une **classe dérivée** est appelée aussi **classe fille** ou **sous-classe**, tandis qu'une **classe de base** est appelée aussi **classe mère** ou **superclasse**.

Si une classe dérive de plusieurs **classes de base** on parle d'**héritage multiple**.

Grâce à l'héritage, la **classe dérivée** :

- récupère l'état et le comportement décrits dans la (ou dans chaque) **classe de base** par l'intermédiaire de ses attributs ;
- peut modifier certains attributs hérités ;
- peut ajouter ses propres attributs (données ou méthodes).

Classe object et type type

L'**héritage** est **relatif** dans le sens où une classe peut être à la fois **classe de base** pour une certaine classe et **classe dérivée** par rapport à une autre classe.

Une classe qui n'indique pas explicitement sa **classe de base** hérite d'office (i.e. automatiquement) d'une **classe prédéfinie** (ou **native**) nommé **object** et *dont la classe de base est elle-même*.

En termes imagés, on peut dire que la classe prédéfinie **object** est la classe racine ou ancêtre de toutes les classes (soit natives soit définies par le programmeur).

Vu qu'une **classe** est à la fois un **type** (pour les objets créés par son instanciation) et un **objet**, on peut se demander quel est le type d'un objet classe.

En fait, le type d'une **classe** est appelé **méta-type** (ou méta-classe).

La plupart des classes ont comme type un **objet prédéfini** nommé **type** et *dont le type est lui-même*.

En termes imagés, on peut dire que le type natif **type** est le **méta-type** (ou la méta-classe) d'origine ou racine.

D'ailleurs, la classe prédéfinie **object** a **type** comme **méta-type**.

Entrée standard et sortie standard

Les données reçues par un programme depuis "l'extérieur" sont couramment appelées les **entrées** ou, simplement, l'**entrée** (*input* en anglais) du programme.

Par exemple, les données d'**entrée** peuvent être :

- introduites par l'utilisateur final (*end user* en anglais) au clavier ;
- lues par le programme à partir d'un fichier ou d'une base de données.

Les résultats calculés et transmis par un programme vers "l'extérieur" sont couramment appelés les **sorties**, ou, simplement, la **sortie** (*output* en anglais) du programme.

Par exemple, les résultats d'un programme vus comme **sorties** peuvent être :

- affichés à l'écran ;
- écrits dans un fichier ou dans une base de données.

Entrée standard et sortie standard (suite)

L'**entrée standard** (*standard input* en anglais) pour un programme est, par défaut, le **clavier** qui correspond à un objet de type **file** nommé **stdin** et qui est un **attribut** du module **sys**.

La **sortie standard** (*standard output* en anglais) pour un programme est, par défaut, l'**écran** (ou une fenêtre console affichée à l'écran) qui correspond à un objet de type **file** nommé **stdout** et qui est un **attribut** du module **sys**.

Afin de lire et de récupérer les **données d'entrée** introduites par l'utilisateur au **clavier**, on peut utiliser la **fonction prédéfinie** (*built-in*) **input()** sans argument ou avec un argument optionnel.

```
input()
input(prompt = '')
```

Entrée standard et sortie standard (suite)

Concrètement, la fonction **input()** :

- écrit dans la sortie standard l'objet argument (s'il y en a un) qui est normalement un texte (de type **str**) ;
- lit une ligne de texte à l'entrée standard (sans le(s) caractère(s) de fin de ligne) ;
- retourne la ligne lue **comme texte** (objet string de type **str**).

La fonction **input()** est une fonction dite **bloquante** dans le sens où elle arrête l'exécution du programme et donne la possibilité à l'utilisateur d'introduire une ligne de texte à l'entrée standard.

En mode interactif, si l'utilisateur introduit une expression (à l'aide du clavier), elle est immédiatement évaluée et le résultat apparaît dans la fenêtre console.

En revanche, en mode script, afin d'afficher des informations dans la **sortie standard**, le programme utilise (le plus souvent) la **fonction prédéfinie** (*built-in*) **print()** sans argument ou avec un ou plusieurs **arguments positionnels** (précisés par leurs valeurs) et/ou **nommés** (précisés par des noms établis et suivis par des signes "égal" et des valeurs).

Entrée standard et sortie standard (suite)

La fonction **print()** peut être appelée :

- sans argument => on passe à la ligne suivante ;
- avec un ou plusieurs **arguments positionnels** (séparés par des virgules) => les objets correspondants sont (transformés en string et) affichés (avec, par défaut, **une seule espace** entre eux) ;
- avec aussi des **arguments nommés**, notamment :
  - **sep** pour indiquer (sous la forme d'une string) le séparateur entre les objets affichés (et le séparateur par défaut est **l'espace**) ;
  - **end** pour préciser (sous la forme d''une string) ce qui se passe à la fin de l'affichage (car sinon, par défaut, **on passe** à la ligne suivante).

Remarque : la transformation d'un objet en string (i.e. en chaîne de caractères) se fait grâce aux fonctions prédéfinies **str()** ou **repr()**.

Entrée standard et sortie standard (suite)

Exemple : soient les instructions ci-dessous.

```
nom = input('Introduisez votre nom : ')
print('Bonjour', nom, '!')
print()
print('Je vous', 'salue', end = ' : ')
print('chaque fois', 'souvent', 'seulement parfois.', sep = ' ou ')
```

A l'exécution, ces instructions affichent :

```
Introduisez votre nom : Toto
Bonjour Toto !

Je vous salue : chaque fois ou souvent ou seulement parfois.
```

Chapitre 3

Structure du code, instructions, expressions, affectations

Python

Exécuter du code Python

Il convient de mentionner que le langage **Python** est un langage de programmation à la fois **interprété** et **compilé**.

Il y a deux possibilités/méthodes pour traiter du code Python :

- en mode interactif ;
- en mode script.

Le **mode interactif** :

- est connu aussi comme REPL – *read-eval-print loop* en anglais ;
- se présente comme un terminal (le shell Python) avec un prompteur spécifique (qui peut être obtenu avec la commande **python** qui lance l'interpréteur interactif Python) ;
- chaque entrée de l'utilisateur est lue et évaluée/exécutée (par l'interpréteur Python), et le résultat est retourné/affiché dans le terminal ;
- le programme est interprété/exécuté de **manière séquentielle**, bout après bout, durant la session interactive.

Exécuter du code Python (suite)

Le **mode script** :

- un programme Python, appelé couramment **script**, peut être encapsulé dans un fichier texte avec l'extension recommandée (mais pas toujours obligatoire) **.py** ;
- un tel **script** peut être utilisé autant de fois qu'on veut ;
- le **script** peut être exécuté à partir d'un terminal avec la commande **python** suivie par le nom du fichier correspondant (avec l'extension **.py**) ;
- plus précisément, dans le cas ci-dessus, le code source est d'abord **compilé** et transformé en code intermédiaire (ou bytecode) qui est ensuite **exécuté** par la machine virtuelle Python **PVM** (***Python Virtual Machine*** en anglais) ;
- le script peut être aussi seulement compilé à partir d'un terminal, grâce à la fonction **compile()** du module **py_compile** qui, appelée avec un argument de type chaîne de caractères **str** correspondant au nom du script, produit un fichier bytecode avec le même nom que le script mais avec l'extension **.pyc** (ou **.pyo**) ;
- le bytecode obtenu par la compilation d'un script, peut être exécuté à partir d'un terminal grâce à la commande **python** suivie par le nom du fichier correspondant (avec l'extension **.pyc**).

Structure d'un programme Python

Toutes les opérations mentionnées pour le **mode script** et effectuées dans un terminal peuvent être aussi réalisées à partir d'un **environnement de développement intégré** (IDE) (qui a été éventuellement utilisé pour l'édition du **script**) par certaines actions au niveau de l'interface graphique de l'IDE ou grâce à certaines combinaisons de touches du clavier.

Un **programme** Python (ou un **code source** Python) est formé d'une suite de **blocs de code** (et peut être stocké dans un fichier texte avec une extension dédiée comme **.py**).

Un **bloc de code** est une partie d'un programme Python qui est exécutée comme (une seule) unité dans un contexte (*frame* en anglais) d'exécution (qui contient des données nécessaires au débogage et qui décide comment l'exécution du programme continue à la fin de l'exécution du bloc).

Exemples de **blocs de code** :

- la définition (le corps) d'une fonction ;
- la définition (le corps) d'une classe ;
- un module ;
- une commande écrite dans l'interpréteur interactif Python.

Bloc de code

Un **bloc de code** est identifiable grâce à une même **indentation** de base (qui est formée, le plus souvent, par quatre espaces ou par une tabulation).

La première ligne d'un programme ne doit pas être indentée.

Un **bloc de code** est une succession de **lignes logiques** (voir plus loin aussi la notion d'instruction simple).

Le caractère # (qui ne fait pas partie d'une constante chaîne de caractères) marque le début d'un **commentaire** qui tient jusqu'à la fin de la ligne physique correspondante.

Les **commentaires** sont ignorés par l'interpréteur mais ils sont utiles, par exemple :

- afin de noter une remarque ou une explication destinée à un programmeur (humain) qui lira le code source ;
- afin de cacher une partie du code vis-à-vis de l'interpréteur durant la conception du programme :
  - soit parce qu'il s'agit d'une partie qui fonctionne correctement et il n'y a pas besoin qu'elle soit analysée (et exécutée ensuite) ;
  - soit parce qu'il s'agit d'une partie qui contient un problème qui sera résolu plus tard.

Ligne logique vs ligne physique

L'interpréteur ignore aussi toute **ligne vide** (*blank line* en anglais), c'est-à-dire une ligne qui contient **seulement** des espaces et des tabulations (ou, plus généralement, des "espaces blancs") ou un éventuel commentaire.

Souvent, une **ligne physique** contient **une seule** **ligne logique** et la fin de la **ligne physique** est aussi la fin de la **ligne logique**.

Cependant, une **ligne logique** peut s'étaler sur une ou plusieurs **lignes physiques**.

Deux **lignes physiques** font partie de la même **ligne logique** si la première ligne physique ne finit pas par un commentaire et son dernier caractère est le backslash **** (qui, dans ce contexte, joue le rôle spécial de "caractère de continuation").

En outre, une **ligne logique** continue après la fin d'une **ligne physique** si une parenthèse **(** ou un crochet **[** ou une accolade **{** a été ouvert et pas encore fermé.

A son tour, une **ligne physique** peut contenir **une ou plusieurs** **lignes logiques** séparées par des caractères points-virgules **;**.

Ligne logique vs ligne physique (suite)

Exemples

```
x = 2 * 3
+ 3 * 4
print(x)                        #affiche 6
y = 2 * 3 \
  + 3 * 4
print(y)                        #affiche 18
a = 1; b = 2; c = 3
print(a, b, c)                  #affiche 1 2 3
```

Mots clés

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

Ligne physique

Chaque **ligne physique** est composée de suites de caractères nommées **tokens** qui peuvent être :

- des "**mots**", à savoir :
  - des **mots clés** (*keywords* en anglais) réservés par le langage Python (voir le tableau ci-dessus) ;
  - des **identificateurs** choisis par le programmeur (exemples : *rayon*, *une_variable*, *MaClasse*, *VIP*, *nom_utilisateur*, *faire_totaux,* etc.) ;
  - des **constantes littérales** i.e. des valeurs (des nombres, des chaînes de caractères, des valeurs logiques, etc.) non modifiables (comme : *10* , *-45.97* , *2.3e-6* , *'Un message string !'*, etc.) ;
- des **délimiteurs**, à savoir :
  - des séparateurs (comme : **espaces**, **tabulations**, **commentaires**, etc.) ;
  - des délimiteurs de structures (comme : **;** , **(** , **[** , **{** , **}** , **]** , **)** , etc.) ;
  - des opérateurs (comme : **+** , **%** , **and** , **<=** , etc.).

Afin de savoir si une chaîne de caractères est un mot clé, on peut utiliser la fonction **iskeyword()** du module **keyword** qui, appelée avec un argument de type **str**, retourne **True** si l'argument est un mot clé ou **False** dans le cas contraire.

Identificateurs

Un **identificateur** (ou **identifiant**) est un nom symbolique (une suite de caractères) utilisé(e) pour **référencer** (ou **pointer**) un objet.

Un **identificateur** ne peut **pas** être un **mot clé** et ne peut **pas** contenir d'**espaces** (ou de tabulations ou de fins de ligne).

Un **identificateur** **doit** commencer par **une lettre** (majuscule ou minuscule) ou **un tiret bas** (*underscore* en anglais) _ suivi par aucun, un ou plusieurs **lettres**, **tirets bas** et **chiffres** (et **ne doit** donc contenir **aucun caractère spécial** comme : **!**, **@**, **#**, **$**, **%**, **^**, **&**, *****, **(**, **)**, **~**, etc.).

Python est **sensible à la casse** (*case sensitive* en anglais) car il fait la distinction entre les majuscules et les minuscules.

Par **convention**, on utilise des **identificateurs** qui commencent :

- par une **majuscule** pour les (objets) **classes** et par une **minuscule** pour les autres objets (par exemple pour les objets instances, les fonctions, les modules, etc.) ;
- par un ou deux tirets bas et finissent par au plus un tiret bas pour des objets attributs privés (*private* en anglais).

Identificateurs

En outre, il y a aussi des identificateurs qui commencent par deux tirets bas et finissent par deux tirets bas et ils sont utilisés pour des objets spéciaux (nommés aussi ***dunders*** ou ***magics*** en anglais) qui sont normalement prédéfinis et ont une signification spéciale pour l'interpréteur.

Afin de ne pas employer d'espaces, si un **identificateur** est formé de plusieurs "mots", on utilise :

- pour les classes, la notation dite **camel case** selon laquelle chaque "mot" distinct de l'identificateur commence par une majuscule ;
- pour les autres objets, la notation dite **snake case** selon laquelle les "mots" distincts de l'identificateur sont séparés entre eux par des tirets bas.

En outre, les **identificateurs** des **constantes** s'écrivent en utilisant comme lettres seulement des majuscules ainsi que la notation **snake case**.

Exemples d'identificateurs :

- valides : nb, temp, student3, var_1, mot_de_passe, Personne, NomClasse, afficher, calculer_moyenne, get_val, EPS, MAX_VALUE, _nom, _repondre ;
- invalides : 3eme_etudiant, $var, est_vrai?, for, aujourd'hui, £are, lambda.

Variables

Afin de traiter des données, un programme Python a besoin de stocker (au moins provisoirement) des informations qui peuvent être ensuite retrouvées et référencées en vue de leur manipulation.

Pour cela, Python utilise des **variables** dont les **noms** *(names* en anglais) sont des **références (**ou **pointeurs) vers des objets**.

Les **noms** sont introduits (ou créés ou définis) par des **opérations de liaisons de noms** (*name binding operations* en anglais) comme celles données comme exemples ci-dessous :

- la définition d'une fonction ;
- la définition d'une classe ;
- l'identifiant qui apparaît comme la cible d'une **affectation** (voir plus loin) ;
- l'en-tête d'une boucle *for* ;
- une instruction *import*.

La liaison entre le **nom** et l'objet référencé peut être détruite, par exemple par :

- une nouvelle **affectation** ;
- une instruction **del** (qui est un mot clé).

Variables (suite)

Un **objet** a une **identité unique et constante** durant son cycle de vie, exprimée par un nombre entier.

Un appel de la fonction (prédéfinie) native (*built-in*) **id()** avec un **nom** de **variable** comme argument retourne l'**identité** de l'objet référencé par cette variable.

De plus, une **variable** n'a **pas de type intrinsèque** et l'appel de la fonction (prédéfinie) native (*built-in*) **type()** avec un **nom** de **variable** comme argument retourne le **type** de l'objet référencé par cette variable.

La valeur, le type et l'identité associés à un **nom** de **variable** peuvent changer (ou *varier*) durant l'exécution d'un programme (d'où le terme de *variable*).

A un **nom** correspond une **variable** qui peut être :

- *locale* à un bloc si le nom est lié dans ce bloc (et s'il n'a pas été déclaré explicitement comme *nonlocal* ou *global*) ;
- *globale* si le nom est lié au niveau d'un module ;
- *libre* si elle est utilisée dans un bloc de code sans y être définie (et la résolution de son nom intervient à l'exécution et non pas à la compilation).

Variables - exemples

```
x = 2.5                 #affectation
print(x)                #affiche 2.5
print(type(x))          #affiche <class 'float'>
print(id(x))            #affiche 2252330474064

x = -7.8                #réaffectation
print(x)                #affiche -7.8
print(type(x))          #affiche <class 'float'>
print(id(x))            #affiche 2252299890832

x = 'Hello !'           #réaffectation
print(x)                #affiche Hello !
print(type(x))          #affiche <class 'str'>
print(id(x))            #affiche 2252331161520

y = x                   #affectation
print(id(y))            #affiche 2252331161520
```

Instructions et expressions

Par la suite, on utilisera plutôt le syntagme **nom** de **variable** ou, tout simplement, **variable** (car **nom** seul a un sens commun qui peut provoquer des confusions).

Il convient de souligner le fait qu'une **variable** n'est pas un objet mais la **référence** (ou un **pointeur**) **vers un objet** et cette distinction est essentielle surtout pour les objets dits **mutables** (ou muables ou modifiables).

On présente maintenant deux notions qui sont apparentées et qui ne peuvent pas toujours être distinguées de manière absolument précise, à savoir les **instructions** et les **expressions**.

Une **instruction** est une portion de code qui est interprétée/exécutée et produit un effet en réalisant une **seule action** ou une **seule commande** (par exemple initialiser/modifier la valeur d'une variable, appeler une fonction ou afficher un message à l'écran).

Une **instruction simple** correspond à une seule **ligne logique** de code et elle ne contient aucune autre instruction.

Une **instruction** qui n'est pas simple est dite **structurée** (ou **composée**) et elle regroupe plusieurs instructions et contrôle leur exécution.

Instructions et expressions (suite)

Une **instruction structurée** peut être considérée et traitée comme une seule instruction.

D'habitude, une **instruction structurée** est formée :

- d'un **en-tête** qui :
  - s'étale le plus souvent sur une **ligne physique** ;
  - démarre avec un **mot clé** ;
  - finit par le **caractère deux-points** ;
- d'un **corps** sous la forme d'un **bloc de code** qui :
  - peut être formé d'une ou de plusieurs instructions (simples ou structurées) ;
  - est **indenté** (vers la droite et de même manière) par rapport à l'en-tête (***indexation significative***).

Le but de cette syntaxe et de l'***indexation significative*** est d'assurer une bonne lisibilité du code Python.

Une **expression** est une portion de code qui peut être **évaluée** (par l'interpréteur) afin d'obtenir une valeur (qui peut être un objet Python quelconque).

Une **expression** peut faire intervenir des valeurs littérales, des variables, des opérateurs, des appels de fonctions, etc. (et elle peut être combinée *horizontalement*).

Instructions et expressions (suite)

Toute **expression** peut être vue comme une **instruction** (dont l'action est d'**évaluer** l'expression et de **retourner** sa valeur).

Les **expressions** sont utilisées, le plus souvent, en tant qu'**instructions** afin de calculer et d'écrire des valeurs ou d'appeler des fonctions.

En revanche, il y a beaucoup d'**instructions** qui ne peuvent pas être utilisées comme **expressions**.

Exemples

```
2 + 5
x = print(2 + 5)      #affiche 7
print(x)              #affiche None
print(type(x))        #affiche <class 'NoneType'>
#l'instruction ci-dessous produit une erreur TypeError qui arrête l'exécution et
#affiche le message "unsupported operand type(s) for +: 'NoneType' and 'int'"
y = print(2 + 5) + 10
```

Affectations

En Python, il n'y a **pas de déclaration** de variable.

En revanche, l'**affectation** (*assignment* en anglais) est utilisée pour **lier** (*bind* en anglais) ou **relier** (*rebind* en anglais) des **noms** de **variables** à des **valeurs** (ou **objets**).

Après l'**affectation**, une **variable** peut être utilisée dans des **expressions**.

Une **affectation** (simple) ne modifie pas l'objet lié ou relié.

Cependant, si à la suite d'une nouvelle **affectation** un objet n'est plus du tout référencé dans un programme, il devient candidat au **ramasse-miettes** (*garbage collector* en anglais) qui est un programme qui tourne en background et réalise le nettoyage **automatique** de la mémoire.

Une **affectation** peut être :

- **simple** si on utilise l'opérateur binaire d'affectation (simple) = ;
- **composée** si on utilise un opérateur binaire d'affectation (composée) parmi les opérateurs suivants : +=, -=, *=, /=, //=, %=, **= (ou parmi les opérateurs d'affectation composée pour la manipulation des bits).

Affectations (suite)

La priorité faible des opérateurs d'**affectation** assure que des opérations arithmétiques et de comparaison s'effectuent avant l'**affectation**.

La syntaxe de l'**affectation simple**

```
nom = expression
```

où :

- **nom** est le nom d'une variable qui représente l'opérande de gauche ;
- **expression** est une expression qui peut être évaluée et qui représente l'opérande de droite.

Le fonctionnement de l'**affectation simple**

Grâce à l'opérateur d'affectation =, la valeur de l'opérande de droite est d'abord évaluée et ensuite affectée (assignée, liée ou reliée) à l'opérande de gauche.

On peut affecter (assigner) une **même** valeur à plusieurs **noms** de **variables** en écrivant, par exemple :

```
nom_1 = nom_2  = nom_3  = expression
```

Affectations (suite)

L'**affectation** est une **instruction** car elle produit un effet mais elle n'est **pas** une **expression** car elle n'a pas de valeur (elle ne retourne aucune valeur).

Par conséquent, une **affectation** ne peut pas être utilisée dans une expression.

Par exemple, l'**instruction** `z = (x = 5) + 3` produit une erreur **SyntaxError** et le message "*invalid syntax*" est affiché.

Normalement, toute **affectation** **composée** peut être remplacée par une **affectation** **simple équivalente**, comme dans les exemples ci-dessous :

- x += 10 remplacée par x = x + 10 ;
- x -= 10 remplacée par x = x - 10 ;
- x *= 10 remplacée par x = x * 10 ;
- x /= 10 remplacée par x = x / 10 ;
- x //= 10 remplacée par x = x // 10 ;
- x % = 10 remplacée par x = x % 10 ;
- x ** = 10 remplacée par x = x ** 10.

Affectations (suite)

Cependant, une **affectation composée** ne peut que **relier** une **valeur** à un **nom** de **variable** qui était **déjà lié** à une précédente valeur (donc, dans tous les exemples ci-dessus, la variable x doit déjà avoir une valeur associée).

Par exemple, si la variable `var` est utilisée dans un programme pour la première fois dans l'instruction suivante : `var += 10` alors une erreur **NameError** se produit et le message "*name 'var' is not defined*" est affiché.

Exemples

```
x = 5 ; x = x + 1
print(x)                              #affiche 6
y = x = 7
print('x =', x, 'et y =', y)     #affiche x = 7 et y = 7
u, v = 3, 5
print('u =', u, 'et v =', v)     #affiche u = 3 et v = 5
#l'instruction ci-dessous produit une erreur SyntaxError qui arrête l'exécution
#et affiche le message "invalid syntax"
z = (x = 5) + 3
```

Chapitre 4

Types natifs simples et opérateurs associés

Python

Types natifs

La notion de **type** est intimement liée à la façon dont l'ordinateur représente, stocke et traite les informations.

Python est un langage de programmation **fortement typé** dans le sens où toute information (tout objet) doit avoir un **type** bien précis.

De plus, Python est **dynamiquement typé** dans le sens où il n'utilise pas de déclarations de **type** explicites (et les **types** sont vérifiés à l'exécution).

Les **types** de données (prédéfinis) **natifs** (*built-in types* en anglais) sont des **types** connus d'office par l'interpréteur (car intégrés à l'interpréteur), comme les **types** :

- **numériques** (ou, simplement, nombres) ;
- séquences ;
- mappings (qui correspondent aux dictionnaires) ;
- classes ;
- instances ;
- exceptions.

Constantes natives

En Python, il y a très peu de **constantes natives** (*built-in constants* en anglais).

On donne ci-dessous trois exemples de **vraies** constantes (immutables) et dont les noms sont des **mots clés** :

- **None** dont le type est **NoneType** ;
- **True** et **False** dont le type est le type booléen **bool**.

La constante **None** :

- est en fait un **singleton** (dans le sens où, dans un programme, il n'y a qu'une seule et unique instance de la classe **NoneType**) ;
- est utilisée normalement pour indiquer l'absence d'une valeur (comme lorsqu'un argument par défaut n'est pas précisé dans l'appel d'une fonction).

Exemple

```
print(type(None))      #affiche <class 'NoneType'>
print(type(True))      #affiche <class 'bool'>
print(type(False))     #affiche <class 'bool'>
var_bool = True
```

Type bool

Le type **bool** est un type prédéfini **natif** (*built-in*) simple (qui est en fait un sous-type du type **int**).

Une **variable** de type **bool** correspond finalement à une des deux constantes natives **True** et **False**.

Exemple

```
var_bool = False
print(type(var_bool))     #affiche <class 'bool'>
print(var_bool)           #affiche False
print(id(var_bool))       #affiche 140713949178224

var_bool_bis = False
print(id(var_bool_bis))   #affiche 140713949178224
print(id(False))          #affiche 140713949178224
print(id(True))           #affiche 140713949178192

var_bool = True
print(id(var_bool))       #affiche 140713949178192
```

Opérateurs booléens

Des **expressions** dont les valeurs sont booléennes peuvent être **composées** grâce à trois **opérateurs booléens** (ou logiques) présentés ci-dessous en ordre croissant de leurs priorités et dont les noms sont des **mots clés** :

- **or** qui correspond à l'opérateur logique binaire **ou** ;
- **and** qui correspond à l'opérateur logique binaire **et** ;
- **not** qui correspond à l'opérateur logique unaire de **négation**.

Les opérateurs binaires **or** et **and** travaillent avec court-circuit dans le sens où le deuxième opérande est évalué seulement si c'est nécessaire pour obtenir la valeur booléenne de la composition, i.e. seulement si :

- la valeur du premier opérande est **False** pour l'opérateur **or** ;
- la valeur du premier opérande est **True** pour l'opérateur **and**.

Exemples

```
t1 = t2 = True ; f1 = f2 = False
print(t1 and t2)      #affiche True
print(t1 and f1)      #affiche False
print(t1 or t2)       #affiche True
```

```
print(t1 or f1)           #affiche True
print(not t1)             #affiche False
print(not f1)             #affiche True
print(not f1 and f2)      #affiche False
```

Opérateurs relationnels

En Python, il y a huit **opérateurs binaires de comparaison** avec la même priorité (qui est plus élevée que celles des opérateurs booléens), à savoir :

- **<** i.e. strictement inférieur ;
- **<=** i.e. inférieur ou égal ;
- **>** i.e. strictement supérieur ;
- **>=** i.e. supérieur ou égal ;
- **==** i.e. **égal** (à ne pas le confondre avec l'opérateur **d'affectation =**) ;
- **!=** i.e. non égal ou différent ;
- **is** i.e. identité d'objet (et **is** est un mot clé) ;
- **is not** i.e. non (ou contraire de l') identité d'objet.

Les opérateurs **is** et **is not** sont appelés aussi des **opérateurs d'identité** (*identity operators* en anglais).

Il y a aussi deux **opérateurs** dits **d'appartenance** (*membership operators* en anglais), à savoir **in** et **not in** qui s'utilisent pour les objets de type **séquence** (et **in** est un mot clé).

L'utilisation de tous ces **opérateurs relationnels** retourne une **valeur booléenne**.

Tester la valeur True pour un objet

Il convient de mentionner que tout objet peut être comparé avec la valeur booléenne **True** (ou **False**).

A part les nombres qui correspondent à **zéro**, les séquences (et les collections) **vides** et les constantes natives **False** et **None** qui sont interprétés comme **False**, les autres objets sont (normalement) considérés comme **True**.

Les opérateurs et les fonctions natives qui ont comme résultat une **valeur booléenne** retournent (normalement) **False** ou **0** pour **faux** et **True** ou **1** pour **vrai**.

Exemples

```
x = 1
print(x == True)      #affiche True
x = 0
print(x == False)     #affiche True
x = 33
print(x == True)      #affiche False
print(x == False)     #affiche False
print(x and True)     #affiche True
```

```
print(0 and False)    #affiche 0
print(False and 0)    #affiche False
print(0 or False)     #affiche False
print(False or 0)     #affiche 0
print(1 or False)     #affiche 1
print(False or 1)     #affiche 1
print(1 and True)     #affiche True
print(True and 1)     #affiche 1
```

Les types numériques

Il y a trois **types numériques** distincts :

- **int** pour des nombres entiers ;
- **float** pour des nombres réels ;
- **complex** pour des nombres complexes avec une partie réelle et une partie imaginaire de type **float** (et la racine carrée de -1 est noté **j** ou **J**).

Les nombres (considérés par défaut en base 10) peuvent être :

- précisés par des constantes numériques ;
- obtenus suite aux calculs faisant intervenir des opérateurs et des fonctions.

Exemples de **constantes numériques** (en base 10) :

- entières : **79** , **+574** , **-397** , **10_000** ;
- réelles en notation décimale (et le point décimal est indispensable) : **28.5** , **3.14_15** , **+0.75** , **-79.0** , **79.** , **.23** , **-.68** ;
- réelles en notation scientifique (la mantisse est un nombre décimal ou entier, avec ou sans signe, et l'exposant est un nombre entier, avec ou sans signe) : **-5.26e6** , **-5.26E6** , **-5.26E+6** , **77.21E-5** , **72e45**.

Les types numériques (suite)

Les nombres entiers de type **int** n'ont pas de limitation (autre que la limitation de mémoire de la machine) pour la précision de leur représentation.

Les nombres réels de type **float** sont représentés selon la norme **IEEE-754** en double précision et certains calculs avec ces nombres ne sont pas rigoureusement exacts, comme dans les exemples ci-dessous (mais ces **erreurs** dites **d'arrondi** sont tout à fait négligeables dans la plupart des calculs d'ingénieur).

```
print(1.1 + 2.2)                  #affiche 3.3000000000000003
print(0.1 + 0.1 + 0.1 - 0.3)      #affiche 5.551115123125783e-17
```

Afin d'améliorer la précision de tels calculs, Python a prévu, par exemple, le module standard **decimal**, pour les nombres réels avec parties décimales, et le module standard **fractions**, pour les nombres rationnels.

En outre, il y a de nombreuses bibliothèques (*libraries* en anglais) Python dédiées aux calculs mathématiques, numériques, scientifiques, techniques et d'ingénieur comme **NumPy** ou **SciPy**.

Les types numériques (suite)

Les constructeurs **int()**, **float()** et **complex()** (qui sont aussi des **fonctions natives** ou *built-in*) peuvent être utilisés afin de créer des nombres ayant un type numérique spécifique.

Chaque tel constructeur peut être appelé :

- sans argument, cas où il crée un objet qui correspond au nombre (entier, réel ou complexe) zéro ;
- avec un seul argument :
  - soit d'un type **numérique** et qui représente un nombre à convertir en un nouveau nombre ;
  - soit de type **str** et qui représente une chaîne de caractères à convertir en nombre.

Cependant, ces dernières conversions ne sont pas toujours possibles (et, dans de tels cas, une erreur/exception est produite et, si elle n'est pas traitée, le programme s'arrête et un message d'erreur approprié est affiché).

Le constructeur **float()** accepte aussi comme argument de type **str** valide la chaîne de caractères '**nan**' (qui n'est pas sensible à la casse) pour des indéterminations (et l'abréviation provient de *Not a Number* en anglais) et une des chaînes de caractères '**inf**' et '**infinity**' (qui ne sont pas sensibles à la casse), précédée éventuellement par le signe '**+**' ou '**-**', pour plus infini et moins infini.

Les types numériques (suite)

Le constructeur **complex()** peut être appelé aussi avec deux arguments numériques de type **int** ou **float** qui deviennent la partie réelle et, respectivement, la partie imaginaire du nouveau nombre complexe créé pour l'occasion.

Pour un objet (nombre complexe) de type **complex**, on peut utiliser :

- l'attribut d'instance nommé **real** afin d'obtenir sa partie réelle ;
- l'attribut d'instance nommé **imag** afin d'obtenir sa partie imaginaire ;
- la méthode d'instance nommée **conjugate()** afin de calculer son conjugué.

Python prévoit aussi le module standard **cmath** qui fournit des fonctions mathématiques qui travaillent avec des nombres complexes.

Python dispose de **fonctions natives** (built-in) pour convertir des nombres entiers écrits en base décimale 10 (qui est la base par défaut) en nombres écrits en base binaire 2 ou base octale 8 ou base hexadécimale 16, à savoir les fonctions : **bin()**, **oct()** et **hex()**.

Les types numériques – exemples

```
print(int())          #affiche 0
print(int(3))         #affiche 3
print(int('10'))      #affiche 10
print(int(5.2))       #affiche 5
print(int(5.9))       #affiche 5
print(int(-5.2))      #affiche -5
print(int(-5.9))      #affiche -5

print(float(7.8))          #affiche 7.8
print(float(10))           #affiche 10.0
print(float('-3.2'))       #affiche -3.2
print(float('  -3.2  ')) #affiche -3.2

u = float('NaN')
print(u)              #affiche nan
print(type(u))        #affiche
                      #<class 'float'>
```

```
w2 = float('-inf')
print(w2)                       #affiche -inf
print(5/w2)                     #affiche -0.0
print(5 + w2)                   #affiche -inf
print(-1 * w2)                  #affiche inf

print(complex(2.1, 3.8))    #affiche
                            #(2.1+3.8j)
print(complex(5.5))         #affiche (5.5+0j)

z = complex('3.2+5j')
print(z)                        #affiche (3.2+5j)
print(z.real)                   #affiche 3.2
print(z.imag)                   #affiche 5.0
print(z.conjugate())            #affiche (3.2-5j)

print(bin(20))                  #affiche 0b10100
print(oct(20))                  #affiche 0o24
print(hex(20))                  #affiche 0x14
```

Opérateurs pour les types numériques

On considère deux variables **x** et **y** dont les types sont **int** ou **float**, et on présente ci-dessous des **opérateurs** (dits **arithmétiques**) disponibles pour ces types d'opérandes :

- **x + y** donne la somme de **x** et **y** ;
- **x – y** donne la différence de **x** et **y** ;
- **x * y** donne le produit de **x** et **y** ;
- **x / y** donne la division ("exacte") ou le quotient ("exact") de **x** et **y** ;
- **x // y** donne la division entière ou le quotient entier de **x** et **y** ;
- **x % y** **x** modulo **y** (donne le reste de la division entière de **x** par **y**) ;
- **-x** donne le négatif (ou l'opposé) de **x** ;
- **+x** laisse **x** inchangé ;
- **x ** y** donne **x** à la puissance **y.**

A part la division entière et le modulo, les autres opérations mentionnées ci-dessus sont valables aussi pour des opérandes de type **complex**.

Opérateurs pour les types numériques (suites)

Si une **expression arithmétique** est **mixte**, par exemple les deux opérandes d'un même opérateur binaire n'ont pas le même type, Python fait implicitement une **conversion d'ajustement de type** dite **non dégradante** (i.e. qui garde l'intégralité des données), selon la hiérarchie suivante :

int → float → complex

Il convient de souligner que l'évaluation d'une **expression arithmétique** se fait en tenant compte de la priorité (*precedence* en anglais) et de l'associativité (*associativity* en anglais) des opérateurs ainsi que de la présence éventuelle des parenthèses.

En outre, il y a aussi des **fonctions natives** (*built-in*) disponibles comme :

- **abs(x)** qui retourne la valeur absolue de **x ;**
- **pow(x, y)** qui retourne **x** à la puissance **y.**

Opérateurs pour les types numériques – exemples

```
print(11 / 4)                  #affiche 2.75
print(11 // 4)                 #affiche 2
print(11 % 4)                  #affiche 3
print(9 % 2.5)                 #affiche 1.5
print(-9 % 2.5)                #affiche 1.0
print(-9 % -2.5)               #affiche -1.5
print(9 % -2.5)                #affiche -1.0

x = 2
print(type(x))                 #affiche <class 'int'>
y = 3.8
print(type(y))                 #affiche <class 'float'>
somme = x + y
print(type(somme))             #affiche <class 'float'>
print(somme)                   #affiche 5.8

z = complex(2.1, -5.8)
print(type(z))                 #affiche <class 'complex'>
print(z)                       #affiche (2.1-5.8j)

w = somme + z
print(type(w))                 #affiche <class 'complex'>
print(w)                       #affiche (7.9-5.8j)

print(pow(3, 4))               #affiche 81
print(abs(-7.8))               #affiche 7.8
```

Chapitre 5

Structures de contrôle de flux

Python

Instructions de contrôle de flux

Les **instructions de contrôle de flux** permettent l'interruption du **déroulement séquentiel** d'un programme, afin d'effectuer :

- des **choix**, grâce aux **instructions structurées de sélection** qui utilisent les mots clés **if** et, éventuellement, **else** et **elif** ;
- des **boucles** (répétitions), grâce aux **instructions structurées de répétition** qui utilisent les mots clés **while** et **for**.

De plus, le langage Python dispose d'**instructions de branchement inconditionnel** qui utilisent les mots clés **break** et **continue** et qui peuvent modifier le déroulement "normal" d'une boucle, ainsi que l'instruction qui correspond au mot clé **pass**.

Par la suite, le terme d'**instruction** sera utilisé dans un sens large pour désigner n'importe quelle **instruction Python** :

- **instruction simple** ;
- **instruction structurée ;**
- **bloc d'instructions**.

Structure de sélection

Syntaxe générale (trois cas possibles)

if condition_booleenne: **bloc_if**	**if condition_booleenne:** **bloc_if** **elif condition_booleenne_1:** **bloc_elif_1** **…** **elif condition_booleenne_n:** **bloc_elif_n** **else:** **bloc_else**
if condition_booleenne: **bloc_if** **else:** **bloc_else**	

où : entre le mot clé **if** ou **elif** et le séparateur **:** se trouve une **expression booléenne** quelconque (d'habitude, une **condition booléenne** à vérifier) ;

un **bloc** qui vient après un séparateur **:** peut être soit une instruction simple ou structurée, soit un groupe d'instructions ;

les trois points **…** indiquent que le nombre de blocs **elif** peut être quelconque.

Structure de sélection (suite)

Il convient de remarquer que :

- les parties **if**, **else** et **elif** sont des en-têtes et finissent par le caractère deux points **:** ;
- les blocs qui viennent ensuite sont indentés par rapport aux en-têtes ;
- chaque **condition booléenne** mentionnée ci-dessus peut être une expression composée formée de diverses expressions booléennes plus simples faisant intervenir, par exemple, des opérateurs de comparaison et reliées avec des opérateurs logiques (**and**, **or** ou **not**).

Fonctionnement

- si la **condition_booleenne** vaut **True**, le **bloc_if** est exécuté, les éventuelles parties **elif** et **else** qui viennent ensuite sont ignorées (et le programme continue avec l'instruction qui se trouve après elles) ;
- si la **condition_booleenne** vaut **False** et :
  - si les parties **else** et **elif** sont absentes, le **bloc_if** est ignoré (et le programme continue avec l'instruction qui suit immédiatement le **bloc_if**) ;
  - si seule la partie **else** est présente et suit immédiatement le **bloc_if**, le **bloc_else** est exécuté (et le programme continue ensuite avec l'instruction qui vient immédiatement après le **bloc_else**).

Structure de sélection (suite)

De plus, si la **condition_booleenne** de la partie **if** vaut **False** et une ou plusieurs parties **elif** viennent après le **bloc_if**, la première partie **elif** se comporte comme la partie **if** et ainsi de suite.

En outre, un choix **if … else** qui fait intervenir des **expressions** peut être écrit en tant qu'**expression conditionnelle** comme dans l'exemple ci-dessous où on calcule la valeur absolue d'un nombre réel.

```
if nb <= 0:
    abs_nb = -nb
else:
    abs_nb = nb
```

Le code de gauche est équivalent au code de droite.

```
abs_nb = -nb if nb <= 0 else nb
```

Exemple de code

A la page suivante, on présente le contenu d'une cellule Jupyter Notebook qui, à l'exécution, demande à l'utilisateur son Indice de Masse Corporelle (IMC) et lui donne un conseil en fonction de la valeur IMC.

Il convient de remarquer l'utilisation de la fonction native **float()** afin de transformer en nombre réel la réponse de l'utilisateur (qui est retournée par la fonction **input()** en tant que chaîne de caractères de type **str**).

Structure de sélection - exemple

```
imc = float(input('Bonjour ! Votre IMC, s.v.p. : '))
if imc < 18.5:
    print('Vous devez manger plus !')
elif imc > 25:
    print('Vous devez bouger plus !')
else:
    print("C'est bien ! Continuez comme ça !")
print('Au revoir !')
```

On donne ci-dessous trois exemples d'affichages en fonction de la réponse de l'utilisateur :

```
Bonjour ! Votre IMC, s.v.p. : 17
Vous devez manger plus !
Au revoir !
```

```
Bonjour ! Votre IMC, s.v.p. : 23
C'est bien ! Continuez comme ça !
Au revoir !
```

```
Votre IMC, s.v.p. : 28
Vous devez bouger plus !
Au revoir !
```

Structure de répétition (conditionnelle) while

La boucle **while** permet d'exécuter un **bloc de code** tant qu'une **expression booléenne** est évaluée à **True** (et cette condition est donc la **condition de continuation** de la boucle).

Syntaxe

while condition_de_continuation:
  bloc_while
	while condition_de_continuation:
	  bloc_while
	else:
	  bloc_else

où la **condition_de_continuation** est une expression booléenne qui est évaluée successivement.

Il convient de mentionner que si la première évaluation de la **condition_de_continuation** donne **False**, alors le **bloc_while** (qui représente le "tour de boucle") n'est jamais exécuté et :

- si la partie (ou la **clause**) **else** est absente, le programme continue avec la première instruction qui vient après le **bloc_while** ;
- si la clause **else** est présente, le **bloc_else** est exécuté et le programme continue ensuite avec la première instruction qui vient après ce bloc.

Structure de répétition (conditionnelle) while (suite)

Fonctionnement général

Si la valeur de la **condition_de_continuation** est :

- **True**, alors :
  - le **bloc_while** est d'abord exécuté ;
  - si aucune instruction **break** n'a été rencontrée et exécutée (voir plus loin), la boucle continue ensuite avec une nouvelle évaluation de la **condition_de_continuation** et ainsi de suite ;
- **False**, alors :
  - s'il n'y a pas de clause **else**, on passe à la suite du programme (qui vient après le **bloc_while**) ;
  - s'il y a une clause **else**, on exécute d'abord le bloc **bloc_else** et on passe ensuite à la suite du programme (qui vient après le **bloc_else**).

Donc, la clause **else** ajoute le bloc supplémentaire **bloc_else** qui est exécuté **seulement** si la boucle a été quittée à la suite de l'évaluation de la **condition_de_continuation** qui correspondait à **False**.

Par conséquent, le **bloc_else** n'est pas exécuté si la boucle **while** est quittée prématurément à la suite d'une instruction **break** (voir plus loin).

Structure de répétition (conditionnelle) while (suite)

La boucle **while** peut être utilisée avec trois instructions supplémentaires : **break**, **continue** et **pass**.

L'instruction **break** peut être utilisée dans le corps d'une boucle **while** (le plus souvent dans le cadre d'un choix **if**) afin de quitter prématurément la boucle (et ceci sans l'exécution d'une éventuelle clause **else**).

L'instruction **continue** peut être utilisée dans le corps d'une boucle **while** (le plus souvent dans le cadre d'un choix **if**) afin d'arrêter le tour de boucle courant et revenir directement à l'évaluation de la **condition_de_continuation** de l'en-tête de la boucle.

L'instruction **pass** ne correspond à aucune opération car il n'y a rien qui se passe quand cette instruction est exécutée.

En fait, **pass** est un substitut (ou remplaçant ou *placeholder* en anglais) qui est utilisé quand la syntaxe demande une instruction mais aucun code n'est en réalité nécessaire.

Structure de répétition (conditionnelle) while – exemple

Rappel : la suite de Fibonacci est une suite récurrente définie ainsi : $\mathbf{f_0 = 0}$, $\mathbf{f_1 = 1}$ et $\mathbf{f_n = f_{n-2} + f_{n-1}}$ pour $\mathbf{n>1}$.

Le code ci-dessous, calcule (de manière itérative) les termes de la suite de Fibonacci qui sont plus petits qu'une valeur donnée (dans cet exemple **20**) et les affiche en ordre croissant :

```
f_0=0 f_1=1 f_2=1 f_3=2 f_4=3 f_5=5 f_6=8 f_7=13
```

Le compteur `n` n'est pas essentiel pour la solution mais il permet d'afficher les indices des termes de la suite.

```
lim = 20
old = 0
new = 1
n = 0
while old < lim:
    print('f_', n, '=', old, sep='', end=' ')
    n += 1
    temp = new
    new = old + new
    old = temp
```

La classe range

La **classe** (ou le **type**) **range** :

- correspond à une **séquence** (immutable ou non modifiable) de nombres ;
- est utilisée souvent afin de préciser combien de fois une boucle **for** doit être exécutée (voir plus loin).

Un **objet range** est une instance de la **classe range** qui a deux constructeurs :

range(stop)	**range(start, stop [, step])**

où :

- les arguments sont des **nombres entiers** ;
- l'argument **stop** indique la fin de la séquence (et sa valeur n'est pas incluse dans la séquence) ;
- l'argument **start** indique le début de la séquence (et sa valeur est incluse dans la séquence) ;
- si l'argument **start** est omis, sa valeur par défaut est **0** ;
- les crochets indiquent une partie facultative ;
- l'argument **step** indique le **pas** de la séquence et, s'il est omis, sa valeur par défaut est **1** ;
- si les valeurs des arguments rendent la séquence **impossible**, alors l'objet créé est une séquence **vide**.

La classe range (suite)

Remarques :

- les arguments du constructeur (y compris l'argument ***step***) peuvent avoir des valeurs positives, négatives ou nulles ;
- les valeurs négatives des indices sont considérées comme de l'indexation à partir de la fin de la séquence ;
- un **objet** de type **range** correspond à une suite arithmétique de nombres entiers ;
- un **objet** de type **range** peut être une séquence **vide** (c'est-à-dire sans aucun élément) ;
- afin d'obtenir un **objet** de type **list** (qui est affiché comme une suite d'éléments entourée par des **crochets**) à partir d'un **objet** de type **range**, on passe ce dernier objet comme argument au constructeur de la classe **list** ;
- les **objets** de type **range** occupent moins de mémoire que des **objets** similaires de type **list** ou **tuple** ;
- les **objets** de type **range** supportent la plupart des opérations spécifiques aux **séquences** (à part la concaténation et la répétition).

La classe range – exemples

```
print(type(range))                  #affiche <class 'type'>
print(range(5))                     #affiche range(0, 5)
print(list(range(5)))               #affiche [0, 1, 2, 3, 4]
print(list(range(2,5)))             #affiche [2, 3, 4]
print(list(range(1, 10, 2)))        #affiche [1, 3, 5, 7, 9]
print(list(range(2, -10, -3)))      #affiche [2, -1, -4, -7]
print(list(range(0)))               #affiche []
print(list(range(10,5)))            #affiche []
print(list(range(0,5,-1)))          #affiche []

#l'instruction structurée ci-dessous affiche :
# i = 0 , i = 1 , i = 2 , i = 3 , i = 4
for i in range(5):
    if i < 4:
        print('i =', i, end=' , ')
    else:
        print('i =', i)

print('Fin !')                      #affiche Fin !
```

Structure de répétition (inconditionnelle) for

Normalement, la boucle **for** est utilisée pour :

- parcourir, élément par élément, une **collection d'objets** qui correspond :
  - soit à un **objet container** d'un type natif (*built-in*) adéquat (par exemple une liste de type **list** ou une chaîne de caractères de type **str**) ;
  - soit, de manière plus générale, à un **objet** dit **itérable** ;
- exécuter un **bloc d'instructions** (le "tour de boucle") un certain nombre de fois, cas où un objet de type **range** est souvent utilisé.

Syntaxe

for element in collection:
  bloc_for
	for element in collection:
	  bloc_for
	else:
	  bloc_else

Structure de répétition (inconditionnelle) for (suite)

où :

- **collection** est un objet container ou itérable ;
- **element** est une variable qui, à tour de rôle, correspond à chaque élément de la collection ;
- **in** est un mot clé qui fait partie (en tant qu'**opérateur d'appartenance**) de la syntaxe de l'en-tête de la boucle **for**.

Fonctionnement

- si le **bloc_for** ne contient pas d'instruction **break**, il est exécuté autant de fois que le nombre d'éléments de l'objet container (ou itérable) **collection** ;
- à chaque exécution d'un tour de boucle, la variable **element** correspond à l'élément courant de l'objet **collection** ;
- s'il n'y a pas de clause **else** après le **bloc_for**, après le dernier tour de boucle, l'exécution du programme continue avec l'instruction qui vient juste après le **bloc_for** ;
- si une clause **else** suit le **bloc_for** et si la boucle n'a pas été quittée à la suite d'une instruction **break** (voir plus loin), le **bloc_else** est exécuté après le dernier tour de boucle et l'exécution du programme continue ensuite avec l'instruction qui vient après le **bloc_else**.

Structure de répétition (inconditionnelle) for (suite)

Comme la boucle **while**, la boucle **for** peut être utilisée avec les trois instructions supplémentaires **break**, **continue** et **pass** et elles ont les mêmes effets que pour la boucle **while**.

En particulier, le **bloc_else** n'est pas exécuté si la boucle **for** est quittée prématurément à la suite d'une instruction **break**.

Remarques finales :

- grâce aux instructions structurées présentées dans ce chapitre, on peut contrôler le flux **à l'exécution** en fonction des informations obtenues au **Runtime** ;
- dans certains cas, on a besoin de créer une **boucle infinie** (qui sera éventuellement arrêtée avec une instruction **break** et) qui peut être une boucle **while** avec une **condition de continuation** toujours **True**, comme dans l'en-tête **while 1:** ;
- afin d'éviter certains effets de bord quand on parcourt une collection avec une boucle **for**, on recommande soit de ne pas changer la collection durant un tel parcours soit d'itérer sur une copie ad hoc de la collection.

Structure de répétition (inconditionnelle) for - exemple

Le code ci-dessous calcule de manière itérative la somme des nombres naturels strictement plus petits qu'une valeur (entière et strictement positive) **n** donnée à l'exécution par l'utilisateur final.

Rappel : la somme mentionnée peut être calculée aussi directement avec la formule **n*(n-1)/2**.

```
n = int(input('Bonjour ! Une valeur entière et strictement positive, s.v.p. : '))
somme = 0
for i in range(1, n):
    somme += i
print(somme)
#Vérification
print(n*(n-1)//2)
```

A l'exécution, ce code affiche (par exemple) :

```
Bonjour ! Une valeur entière et strictement positive, s.v.p. : 11
55
55
```

Chapitre 6

Fonctions

Python

Fonctions

Dans ce chapitre, on introduit les mots clés : **def**, **return**, **global** et **nonlocal**.

Une **fonction** permet de regrouper du code (des instructions) afin de l'exécuter (les exécuter) **ultérieurement** et, le plus souvent, **plusieurs fois**.

En plus, grâce aux fonctions on peut :

- améliorer la compréhension et la lisibilité du code ;
- simplifier le développement et la maintenance du code.

Il y a des **fonctions** :

- **prédéfinies natives** - utilisables directement et partout grâce à leur nom ;
- **prédéfinies** dans des **modules standards** appartenant à la **bibliothèque** (*library*) **standard** Python - utilisables grâce à leur nom, mais après avoir été importées ou grâce à leur nom qualifié (*qualified name* en anglais), i.e. le nom de la fonction précédé par le nom du module où la fonction a été définie ;
- **définies** dans des **modules** faisant partie de **bibliothèques tierces** qui doivent être d'abord installées ;
- **définies** par le programmeur dans un **programme** (ou, plus généralement, dans un **module** écrit par le programmeur).

Callable

Une **fonction** est un objet de **type function**, donc un objet **appelable** ("*callable*" en anglais) sur lequel on peut faire un appel.

On peut vérifier si un objet est **callable** grâce à la fonction prédéfinie native **callable()** qui, appelée avec (la référence d') un objet comme argument, retourne :

- **True** si l'objet argument est **callable** (mais ceci ne garantit pas que cet objet peut être appelé) ;
- **False** autrement (et ceci garantit que l'objet ne peut pas être appelé).

```
#l'instruction ci-dessous affiche
#<class 'builtin_function_or_method'>
print(type(pow))

print(callable(pow))        #affiche True

print(callable(None))       #affiche False

x = 3.14
print(callable(x))          #affiche False

import math
```

```
#l'instruction ci-dessous affiche
#<class 'module'>
print(type(math))

print(callable(math))       #affiche False

#l'instruction ci-dessous affiche
#<class 'builtin_function_or_method'>
print(type(math.sqrt))

print(callable(math.sqrt)) #affiche True
```

Définition d'une fonction

Il faut distinguer :

- la **définition** de la **fonction** ;
- l'**appel** de la **fonction**.

Comme les fonctions mathématiques, une **fonction** Python peut avoir des **paramètres**.

Par la suite, on utilise les termes suivants :

- **paramètres formels** (ou paramètres **muets** ou, simplement, paramètres) dans le cas de la **définition** d'une **fonction** ;
- **arguments effectifs (**ou paramètres **effectifs** ou, simplement, arguments**)** dans le cas de l'**appel** d'une **fonction**.

La **définition** d'une **fonction** est une **instruction exécutable** qui :

- est exécutée (seulement et chaque fois) quand la **fonction** est appelée ;
- **lie** le nom de la **fonction** (dans l'espace de nom courant) à l'objet **fonction** qui encapsule le code exécutable de la **fonction**.

Définition d'une fonction (suite)

La **définition** d'une **fonction** est une **instruction structurée** formée :

- de l'**en-tête** de la **fonction** qui :
  - correspond à une première ligne **logique** ;
  - peut s'étaler sur une ou plusieurs lignes **physiques** ;
  - précise la **signature** de la **fonction** ;
- du **corps** de la **fonction** qui :
  - est **indenté** (vers la droite) par rapport à l'en-tête ;
  - contient le code à exécuter lors de l'**appel** de la **fonction**.

Syntaxe

```
def nom_fonction(liste_parametres):
    corps_fonction
```

En-tête et corps d'une fonction

L'**en-tête** d'une **fonction** contient dans l'ordre :

- le mot clef **def** suivi par le **nom** de la **fonction** qui doit être un identificateur valide qui commence (normalement) par une minuscule ;
- une paire de **parenthèses** rondes (celle de gauche ouvrante et celle de droite fermante) qui :
  - sont obligatoires à la fois dans la **définition** et dans l'**appel** de la fonction (y compris pour les **fonctions** qui n'ont aucun paramètre) ;
  - encadrent la liste des **paramètres formels** séparés par des virgules (et cette liste peut être éventuellement vide ou contenir un seul paramètre) ;
- le séparateur **deux points :** qui marque la fin de l'en-tête.

Le **corps** de la **fonction** :

- représente un **bloc** d'instructions qui est **indenté** (vers la droite) par rapport à l'en-tête ;
- ne peut **pas** être **absent** ; si une **fonction** ne fait (encore) rien, on réduit (d'habitude) son corps à l'instruction **pass** *;*
- peut commencer par une documentation (**Docstring**) qui est facultative mais fortement conseillée ;
- peut contenir aucune, une ou plusieurs instructions **return** ;
- est exécuté à chaque appel de la fonction.

Docstring

Le **corps** d'une **fonction** peut commencer avec un **commentaire** spécial qui :

- est nommé **Docstring** (ou chaîne de documentation) ;
- est délimité avec des **triples quottes** au début (à gauche) et à la fin (à droite) ;
- peut s'étaler sur **plusieurs lignes physiques** ;
- décrit (d'habitude) le but et explique le travail de la **fonction** grâce à :
  - un résumé ;
  - une description étendue ;
  - une liste de **paramètres formels** avec **leur types** et **leur rôles** ;
  - une précision concernant l'éventuel **résultat retourné** avec **son type** et **sa signification** ;
- peut être consulté en dehors de la fonction grâce à l'**attribut** d'instance **magique** (ou **dunder**) **__doc__** (précédé par le nom de la fonction et l'opérateur **.**).

On donne à la page suivante un exemple d'une **fonction** donc le **corps** commence par un commentaire **Docstring**.

Docstring – exemple

```
def sommer(x, y):
    """
    Résumé : Une fonction qui …
    Description étendue : …
    Paramètres :
    x (float) : le premier terme …
    y (float) : le second terme …
    Retourne :
    float : la somme des deux paramètres
    """
    return x + y
print(sommer(3.5, 6.5))
print(sommer.__doc__)
```

A l'exécution, le bloc de code de gauche affiche :

```
10.0

    Résumé : Une fonction qui …
    Description étendue : La somme …
    Paramètres :
    x (float) : le premier terme …
    y (float) : le second terme …
    Retourne :
    float : la somme des deux paramètres
```

Définition d'une fonction - Paramètres formels

Un **paramètre formel** peut être :

- **obligatoire** (*mandatory* en anglais), cas où :
  - dans l'**en-tête** de la **fonction** :
    - ce paramètre est précisé seulement par son **nom** (sans qu'une valeur lui soit associée) ;
    - il se trouve **en début** de la liste des paramètres (**avant** les paramètres **optionnels**) ;
  - à l'**appel** de la fonction, il **doit** être **précisé** par l'intermédiaire d'un **argument effectif** correspondant ;
- **optionnel** (*optional* en anglais) ou **par défaut** (*default* en anglais), cas où :
  - dans l'**en-tête** de la **fonction**, ce paramètre vient **après** les paramètres **obligatoires** ;
  - il est précisé par son **nom** et par sa **valeur par défaut** sous la forme :

nom_parametre = valeur_par_defaut

  - à l'**appel** de la **fonction**, il **peut** être **omis** (et l'**argument effectif** correspondant aura alors sa **valeur_par_défaut**).

Appel d'une fonction – Arguments effectifs

L'**appel** d'une **fonction** (ou, plus généralement, d'un objet **callable**) a comme but d'exécuter le **corps** de la **fonction** et cet **appel** :

- est formé du **nom** de la fonction suivi par une paire de **parenthèses** (celle de gauche ouvrante et celle de droite fermante) qui entourent la liste des **arguments effectifs** (qui peut être éventuellement vide) ;
- réalise d'abord les **liaisons de noms** entre les **arguments effectifs** et les **paramètres formels** ;
- exécute ensuite le **corps** de la fonction ;
- retourne finalement une **valeur** (éventuellement la **constante native None**), sauf si une exception est lancée (sans être traitée localement) durant l'exécution de la **fonction**.

Dans l'**appel** d'une **fonction** :

- on **doit** fournir un **argument effectif** pour chaque **paramètre formel obligatoire** ;
- on **peut** fournir un **argument effectif** pour un **paramètre formel optionnel** mais, si la valeur par défaut convient, l'**argument effectif** peut être **omis** et la **valeur par défaut** sera utilisée à sa place.

Arguments effectifs (suite)

Un **argument effectif** de l'**appel** d'une **fonction** peut être :

- **positionnel** (*positional argument* en anglais) s'il est précisé sous la forme d'une **expression** dont la **valeur** sera **liée** au **paramètre formel** correspondant grâce à sa **position** dans la liste des paramètres ;
- **nommé** ou **par mot clé** (*keyword argument* en anglais) s'il est de la forme :

  nom_argument = valeur_effective

  et la valeur de **valeur_effective** sera **liée** au **paramètre formel** correspondant grâce à son **nom_argument** ; ainsi, l'utilisateur de la fonction n'est pas obligé de se rappeler l'ordre des paramètres de l'en-tête de la fonction.

En outre, dans l'**appel** d'une **fonction** :

- les arguments **nommés** doivent **suivre** les arguments **positionnels** (et ils sont eux-mêmes convertis en arguments **positionnels**) ;
- si un argument **nommé** ne correspond à aucun paramètre **formel**, une erreur de type **TypeError** est lancée.

Nombre variable d'arguments

En outre, une **fonction** peut être **appelée** avec un nombre **variable** d'**arguments effectifs.**

Dans ce cas, dans la **signature** de la **fonction**, la liste de **paramètres formels** finit (normalement) :

- soit par un **dernier** paramètre dont le nom commence par * (par exemple ***args**) et qui permet l'appel de la **fonction** avec un nombre quelconque d'**arguments positionnels** supplémentaires ;
- soit par un **dernier** paramètre dont le nom commence par **, par exemple ****kwargs**, et qui permet l'appel de la **fonction** avec un nombre quelconque d'**arguments nommés** supplémentaires ;
- soit par un **avant-dernier** paramètre dont le nom commence par * suivi par un **dernier** paramètre dont le nom commence par ** (donc une combinaison des deux possibilités précédentes).

Ainsi, dans l'**appel** d'une **fonction**, si le nombre d'**arguments effectifs** (positionnels et/ou nommés) est plus grand que le nombre de **paramètres formels**, alors :

- si un **paramètre formel** utilise la syntaxe ***args**, on lui fait correspondre un **tuple** (accessible dans le corps de la fonction par **args** et) contenant les **arguments positionnels** en plus ;
- si un **paramètre formel** utilise la syntaxe ****kwargs**, on lui fait correspondre un **dictionnaire** (accessible dans le corps de la fonction par **kwargs** et) contenant les **arguments nommés** en plus ;
- sinon, une exception de type **TypeError** est lancée.

Fonctions - exemples

```
def soustraire(x1, x2):
    return x1 - x2

print(type(soustraire))                 #affiche <class 'function'>
print(soustraire(5, 3))                 #affiche 2
print(soustraire(x1 = 11, x2 = 7))      #affiche 4
print(soustraire(x2 = 33, x1 = 22))     #affiche -11
print(soustraire(100, x2 = 40))         #affiche 60

#ci-dessous SyntaxError : "positional argument follows keyword argument"
#soustraire(x1 = 100, 40)

#ci-dessous TypeError : "soustraire() got an unexpected keyword argument 'x3'"
#soustraire(x1 = 10, x3 = 5)

#ci-dessous TypeError : "soustraire() takes 2 positional arguments but 3 were
given"
#soustraire(1, 2, 3)
```

```
#ci-dessous SyntaxError : "non-default argument follows default argument"
def sommer(x1=0, x2):
    pass
```

Fonctions – exemples (suite)

```
def sommer(x1, x2=1):
    return x1 + x2

print(sommer(7))       #affiche 8
print(sommer(7, 10))   #affiche 17
```

```
def sommer(x1, x2, *args):
    somme = x1 + x2
    for x in args:
        somme += x
    return somme

print(sommer(1, 2))            #affiche 3

print(sommer(1, 2, 3, 4, 5))   #affiche 15

#ci-dessous TypeError : "sommer() missing 1 required positional argument: 'x2'"
#sommer(1)

#ci-dessous SyntaxError : "positional argument follows keyword argument"
#sommer(x1=1, x2=2, 3)
```

Fonctions - exemples (suite)

```
def sommer(x1, x2, **kwargs):
    somme = x1 + x2
    for (key, val) in kwargs.items():
        somme += val
    return somme

print(sommer(1,2))                  #affiche 3
print(sommer(1, 2, y=11, z=12))     #affiche 26
print(sommer(x1=1, x2=2, y=11))     #affcihe 14

#ci-dessous TypeError : "sommer() takes 2 positional arguments but 3 were given"
#sommer(1, 2, 3)
```

```
#ci-dessous SyntaxError : "invalid syntax"
def sommer(x1, x2, **kwargs, *args):
    pass
```

Fonctions – exemples (suite)

```
def sommer(x1, x2, *args, **kwargs):
    somme = x1 + x2
    for x in args:
        somme += x
    for (key, val) in kwargs.items():
        somme += val
    return somme

print(sommer(1, 2))                 #affiche 3
print(sommer(1, 2, 3))              #affiche 6
print(sommer(1, 2, 3, y=4, z=5))    #affiche 15
#ci-dessous erreur SyntaxError et message : "positional argument follows keyword argument"
#sommer(1, 2, y=4, z=5, 3)

#ci-dessous erreur SyntaxError et message : "positional argument follows keyword argument"
#sommer(x1=1, x2=2, 3, y=4)
```

Paramètres formels optionnels et objets mutables

De plus, dans l'**appel** d'une **fonction** :

- les valeurs **par défaut** des **paramètres formels optionnels** sont évaluées :
  - de gauche à droite ;
  - une seule fois ;
  - quand la définition de la fonction est rencontrée ;

  et ces **valeurs précalculées** sont ensuite utilisées à **chaque appel** ;
- si la **valeur par défaut** d'un paramètre **formel optionnel** est (la référencée d') un **objet mutable** (par exemple une **liste** ou un **dictionnaire**), celui-ci sera **partagé** par tous les appels qui ne précisent pas de **valeur explicite** pour l'argument effectif correspondant.

Par conséquent, si on veut éviter que les **appels successifs** d'une **fonction** partagent un **même objet mutable** (qui correspond à la **valeur par défaut** d'un de ses **paramètres formels optionnels**), on procède d'habitude comme dans le deuxième exemple ci-dessous.

Paramètres formels optionnels et objets mutables - exemples

```
lis = [0]
def ajouter(el, li = lis):
    li.append(el)              #on ajoute l'élément el à la liste li
    return li

lis = [-1]
print(ajouter(1))              #affiche [0, 1]
print(ajouter(2))              #affiche [0, 1, 2]
print(ajouter(777, [111]))     #affiche [111, 777]
print(ajouter(3))              #affiche [0, 1, 2, 3]
```

```
def ajouter(el, li = None):
    if li == None:
        li = []
    li.append(el)              #on ajoute l'élément el à la liste li
    return li

print(ajouter(1))              #affiche [1]
print(ajouter(2))              #affiche [2]
print(ajouter(777, [111]))     #affiche [111, 777]
print(ajouter(3))              #affiche [3]
```

Mot clé return

A la suite de son **appel** et grâce au mot clé **return**, une **fonction** peut **retourner** (à l'appelant de la fonction) **un** ou **plusieurs résultats** (séparés par des virgules).

Plus précisément, quand la **fonction** retourne **plusieurs résultats**, elle retourne en fait un résultat de **type tuple**.

De plus, vu que le **résultat** retourné par une **fonction** peut être l'adresse d'un objet de n'importe quel type (valide), si l'objet retourné est d'un type **container**, la fonction peut retourner ainsi plusieurs valeurs regroupées dans le **container** respectif.

Il convient de préciser que si l'**exécution** d'une **fonction** arrive à une instruction **return**, alors le **corps** de la **fonction** est **quitté** après l'exécution de cette instruction et le programme revient à l'endroit où la fonction a été appelée par son appelant (par exemple, une fonction appelante) et l'**appel** de la **fonction** est remplacé par la **valeur retournée**.

Mot clé return (suite)

Plus précisément, si le **corps** d'une **fonction** :

- ne contient **aucune** instruction **return**, alors quand la **fonction** est appelée :
  - elle est exécutée **intégralement** ;
  - la valeur **None** est **retournée** (à l'appelant de la fonction) ;
- contient **une** ou **plusieurs** instructions **return** (assez souvent intégrées dans des structures de choix et/ou de répétition), alors quand l'exécution de la **fonction** arrive à une telle instruction et le mot clé **return** :
  - n'est suivi d'aucune expression :
    - la fonction est **quittée** (immédiatement) ;
    - la valeur **None** est **retournée** (à l'appelant de la fonction) ;
  - est suivi d'une **expression** :
    - la fonction est **quittée** (immédiatement) ;
    - la valeur de cette expression est **retournée** (à l'appelant de la fonction).

Mot clé return - exemples

```
def saluer(s):
    print('Le début de la fonction !')
    print(s)
    print('La fin de la fonction !')

print(saluer('Bonjour !'))
```

Le code de gauche affiche :

```
Le début de la fonction !
Bonjour !
La fin de la fonction !
None
```

```
def saluer(s, n):
    print('Le début de la fonction !')
    if n <= 0 or n > 3:
        print(s)
        return
    for i in range(1, n+1):
        print(s)
    print('La "fin" de la fonction !')

print(saluer('Hello !', 10))
print('--------------------------')
print(saluer('Salut !', 2))
```

Le code de gauche affiche :

```
Le début de la fonction !
Hello !
None
--------------------------
Le début de la fonction !
Salut !
Salut !
La "fin" de la fonction !
None
```

Mot clé return – exemples (suite)

```
def get_max(x, y):
    print('Avant return !')
    return x if x > y else y
    print('Après return !')  #jamais exécutée

maxim = get_max(-3,3)

print(maxim)
```

Le code de gauche affiche :

```
Avant return !
3
```

```
def comparer(x, y):
    if x > y:
        s = 'Premier plus grand !'
        return s, 1
    if x < y:                    #pas besoin de "elif"
        s = 'Second plus grand !'
        return s, -1
    #pas besoin de "else"
    s = 'Egalité !'
    return s, 0

message, valeur = comparer(-3, -2)

print(message)

print(valeur)
```

Le code de gauche affiche :

```
Second plus grand !
-1
```

Fonctions imbriquées et fonctions d'ordre supérieur

En Python, une **fonction** est un **objet** et elle peut donc être :

- **affectée** (assignée) à une variable (ce qui revient à créer un **alias** pour cette fonction) ;
- utilisée comme **paramètre/argument** d'une (autre) fonction ;
- **retournée** comme résultat par une (autre) fonction.

Une **fonction imbriquée** ou **locale** (*nested* en anglais) est une **fonction** qui est définie (à l'aide d'une instruction structurée **def**) à l'**intérieur** (i.e. dans le corps) d'une autre fonction dite **fonction englobante**.

Une **fonction d'ordre supérieur** est une **fonction** dont au moins un **paramètre** ou le **résultat** retourné est une **fonction**.

En outre, une **fonction imbriquée** peut être employée dans le corps de sa **fonction englobante** :

- par un appel direct ;
- comme argument dans l'appel d'une autre fonction ;
- comme valeur retournée par la **fonction englobante** (qui est, dans ce cas, une **fonction d'ordre supérieur**).

Fonctions imbriquées et fonctions d'ordre supérieur - exemples

```
def reculer(n):
    if n > 0:
        return n-1
    return 0

rec = reculer
print(type(rec))                #affiche <class 'function'>
print(id(reculer)==id(rec))     #affiche True
print(rec(11))                  #affiche 10
del reculer
print(rec(-10))                 #affiche 0
#reculer(1)                     #NameError : "name 'reculer' is not defined"
```

```
def calculer_moyenne(note1, note2, poids1, poids2):
    def verifier(p1, p2):
        if p1>0 and p2>0 and p1+p2==1:
            return True
        return False

    if(verifier(poids1, poids2)):
       return note1*poids1 + note2*poids2
    return note1*0.5 + note2*0.5

print(calculer_moyenne(5, 6, .25, 0.75))    #affiche 5.75
```

Fonctions imbriquées et fonctions d'ordre supérieur – exemples (suite)

```
def verifier(p1, p2):
    if p1>0 and p2>0 and p1+p2==1:
        return True
    return False

def calculer_moyenne(note1, note2, poids1, poids2, tester):
    if(tester(poids1, poids2)):
       return note1*poids1 + note2*poids2
    return note1*0.5 + note2*0.5

print(calculer_moyenne(5, 6, 0.25, 0.75, verifier))    #affiche 5.75
```

```
def ajouter(x):
    y = 10
    def sommer(z):
        return x + y + z
    return sommer

additionner = ajouter(3)
print(additionner(5))        #affiche 18
additionner = ajouter(7)
print(additionner(5))        #affiche 22
```

Fonctions récursives

Une **fonction récursive** est une **fonction** dont le corps contient :

- soit (au moins) une instruction qui fait appel à la **fonction-même** (pour la **récursivité directe**) ;
- soit (au moins) une instruction qui appelle une **autre fonction** qui appelle, à son tour, la **fonction-même** (pour la **récursivité indirecte**).

Il faut toujours prévoir une **condition d'arrêt** ou **de fin** (*termination condition* en anglais) - appelée aussi **cas de base** (*base case* en anglais) - qui évite les appels **récursifs** infinis.

Le plus souvent, la **condition d'arrêt** correspond à un cas dit **cas de base** (*base case* en anglais) pour lequel le problème peut être résolu sans avoir besoin d'une autre récursion.

Souvent, l'approche **récursive** est utilisée quand on peut ramener la résolution d'un problème au calcul des solutions d'instances plus petites de ce même problème.

Cependant, si un même problème peut être résolu de manière **récursive** et **itérative**, souvent, la solution **récursive** **non optimisée** demande plus de temps de calcul et/ou de mémoire que la solution **itérative**.

Fonctions récursives – factorielle

Rappel : La factorielle d'un entier positif **n** est donnée par : **n! = n*(n-1)!** pour **n>1** (et **1!=1)** et **0! = 1**.

```
def factorielle_rec(n):                 #http://www.pythontutor.com/visualize.html
    if n == 0:
        return 1
    return n*factorielle_rec(n-1)

print(factorielle_rec(4))               #affiche 24
```

```
def factorielle_iter(n):                #http://www.pythontutor.com/visualize.html
    fact = 1
    for i in range(2, n+1):
        fact *= i
    return fact

print(factorielle_iter(4))              #affiche 24
```

```
def factorielle_rec_plus(n):
    print('n =', n, end = ' ')
    if n == 0:
        return 1
    return n*factorielle_rec_plus(n-1)

print(factorielle_rec_plus(4))     #affiche : n = 4 n = 3 n = 2 n = 1 n = 0 24
```

Fonctions récursives – suite de Fibonacci

Rappel : la suite de Fibonacci est une suite récurrente définie ainsi : $\mathbf{f_0 = 0}$, $\mathbf{f_1 = 1}$ et $\mathbf{f_n = f_{n-2} + f_{n-1}}$ pour $\mathbf{n>1}$.

```
#Suite de Fibonacci récursive
def fibo_rec(n):                          #http://www.pythontutor.com/visualize.html
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibo_rec(n-2) + fibo_rec(n-1)

print(fibo_rec(30))                       #affiche 832040
```

```
#Suite de Fibonacci itérative
def fibo_iter(n):                         #http://www.pythontutor.com/visualize.html
    old, new = 0, 1
    if n == 0:
        return 0
    for i in range(n-1):
        temp = new
        new = old + new
        old = temp
    return new

print(fibo_iter(30))                      #affiche 832040
```

Espaces de noms et portées

Un **espace de noms** (*namespace* en anglais), appelé aussi **contexte**, est un mapping (une correspondance) entres des **noms** et des **objets** qui assure que les **noms** soient **uniques** afin d'éviter toute ambiguïté.

En outre, deux **espaces de noms** différents peuvent utiliser un même **nom** sans générer des conflits de noms car chaque **nom** est associé à son **espace de noms**.

Par exemple, beaucoup de systèmes d'exploitation utilisent les **dossiers** (*directories* en anglais) comme des **espaces de noms** et, par conséquent :

- un **dossier** ne peut pas contenir deux **fichiers** (ou sous-dossiers) de même nom ;
- deux **fichiers** (ou sous-dossiers) peuvent avoir le même nom s'ils se trouvent dans des **dossiers** différents.

En Python, les **espaces de noms** sont implémentés à l'aide des **dictionnaires** mais cet aspect technique peut rester transparent pour le programmeur.

Espaces de noms et portées (suite)

On peut distinguer plusieurs **espaces de noms**, comme :

- **espace de noms natif** (*built-in*) qui contient les noms des **fonctions** et des **exceptions natives** ;
  - il est créé au début du travail de l'interpréteur et il existe tout au long de ce travail ;
- **espace de noms global** qui concerne un **module** ;
  - il est créé au moment où le module est lu et il existe jusqu'à la fin de l'interprétation/exécution du module ;
- **espace de noms local** à une **fonction** (ou, plus généralement, à un **bloc de code**) ;
  - il est créé quand la **fonction** est appelée et il est détruit à la fin de l'exécution de la fonction (y compris quand la fin est due à une exception lancée et pas traitée localement).

La **portée** (*scope* en anglais) **d'un identifiant** (ou d'un **nom**) est la partie d'un programme où le **nom** est **lié**, i.e. où il peut être utilisé sans ambiguïté.

En fait, la **portée** d'un **identifiant** est l'**espace de noms** de cet **identifiant**.

Espaces de noms et portées (suite)

Les **portées** (comme les **espaces de noms**) peuvent être **imbriquées** et, par exemple : la **portée locale** d'une **fonction** définie à l'intérieur d'une autre **fonction** est incluse dans la portée **locale** de la **fonction englobante** qui est incluse dans la **portée globale** du **module** qui est incluse dans la **portée native** de l'**interpréteur**.

L'**espace de nom global** associé à un **module** contient les **variables** dites **globales** qui sont des **attributs** de l'objet **module**.

L'**espace de nom local** associé à une **fonction** contient :

- les **noms** de ses éventuels **paramètres formels** (précisés dans l'en-tête de la **fonction**) ;
- les **noms** des **variables** dites **locales** à cette **fonction**, à savoir les **noms** qui sont **liés** (par des affectations ou par d'autres instructions qui créent des **liaisons de noms**) dans le corps de la fonction, à l'exception des **variables déclarées explicitement** avec le mot clé **global** ou **nonlocal**.

Une **variable locale** à une **fonction** n'est plus accessible (n'existe plus) en dehors de la **fonction** (i.e. quand l'exécution de la **fonction** à la suite de son appel a pris fin).

Mot clé global – fonction définie dans un module

Dans le corps d'une **fonction** définie dans un **module**, une **variable locale masque par défaut** une **variable globale** de même nom.

En outre, **par défaut**, dans une **fonction** qui n'a pas de **variable locale** de même nom, une **variable globale** :

- peut être utilisée "en lecture", i.e. on peut utiliser sa valeur (ou l'objet lié au nom de la **variable globale**) ;
- ne peut pas être utilisée "en écriture", i.e. on ne peut pas changer sa valeur car la tentative de **relier** le nom de la **variable globale** à un autre objet conduit à la création d'une **variable locale** de même nom.

Afin de **changer** ce comportement **par défaut**, il suffit de **déclarer** la **variable globale** concernée avec le mot clé **global** dans la **fonction**, comme dans les exemples ci-dessous.

Plus précisément, cette fois, la **fonction peut changer** la valeur de la **variable globale déclarée global** mais ne peut plus avoir une **variable locale** de même nom (autrement dit, une même variable ne peut **pas** être **à la fois locale** et **globale**).

En outre, un **paramètre** (formel) d'une **fonction** ne peut **pas** être **déclaré global** dans le corps de la fonction.

Mot clé nonlocal – fonction imbriquée

Pour une **fonction imbriquée** définie à l'intérieur d'une **fonction englobante** :

- les considérations faites au paragraphe précédent concernant les **variables globales** restent valables ;
- de plus, une **variable locale** à la **fonction imbriquée** **masque** **par défaut** aussi une **variable locale** de même nom de la **fonction englobante**.

Afin de **changer** ce comportement (supplémentaire) **par défaut**, il suffit de **déclarer** la **variable locale** de la **fonction englobante** concernée avec le mot clé **nonlocal** dans la **fonction imbriquée**, comme dans les exemples ci-dessous.

Plus précisément, cette fois, la **fonction imbriquée** **peut changer** la valeur de la **variable locale** à la **fonction englobante** **déclarée** **nonlocal** mais ne peut plus avoir une **variable locale** de même nom.

Un **paramètre** (formel) d'une **fonction** ne peut **pas** être **déclaré** **nonlocal** dans le corps de la fonction.

Fonction définie dans un module – exemple 1

```
def sommer():
    print('Fonction : nb1 =', nb1, 'et nb2 =', nb2)
    nb3 = 7
    return nb1 + nb2 + nb3

nb1, nb2 = 3, 5
print('Module, avant l\'appel : nb1 =', nb1, 'et nb2 =', nb2)
print(sommer())
print('Module, après l\'appel : nb1 =', nb1, 'et nb2 =', nb2)
#ci-dessous NameError et message : "name 'nb3' is not defined"
#print(nb3)
```

Le code ci-dessus affiche :

```
Module, avant l'appel : nb1 = 3 et nb2 = 5
Fonction : nb1 = 3 et nb2 = 5
15
Module, après l'appel : nb1 = 3 et nb2 = 5
```

Fonction définie dans un module – exemple 2

```
def sommer():
    #ci-dessous UnboundLocalError : "local variable 'nb1' referenced before assignment"
    #print(nb1)
    nb1, nb2, nb3 = 33, 55, 77
    print('Fonction : nb1 =', nb1, 'et nb2 =', nb2)
    return nb1 + nb2 + nb3

nb1, nb2 = 3, 5
print('Module, avant l\'appel : nb1 =', nb1, 'et nb2 =', nb2)
print(sommer())
print('Module, après l\'appel : nb1 =', nb1, 'et nb2 =', nb2)
#ci-dessous NameError et message : "name 'nb3' is not defined"
#print(nb3)
```

Le code ci-dessus affiche :

```
Module, avant l'appel : nb1 = 3 et nb2 = 5
Fonction : nb1 = 33 et nb2 = 55
165
Module, après l'appel : nb1 = 3 et nb2 = 5
```

Fonction définie dans un module – exemple 3

```
def sommer():
    global nb1, nb2
    nb1, nb2, nb3 = 33, 55, 77
    print('Fonction : nb1 =', nb1, 'et nb2 =', nb2)
    return nb1 + nb2 + nb3

nb1, nb2 = 3, 5
print('Module, avant l\'appel : nb1 =', nb1, 'et nb2 =', nb2)
print(sommer())
print('Module, après l\'appel : nb1 =', nb1, 'et nb2 =', nb2)
#ci-dessous NameError et message : "name 'nb3' is not defined"
#print(nb3)
```

Le code ci-dessus affiche :

```
Module, avant l'appel : nb1 = 3 et nb2 = 5
Fonction : nb1 = 33 et nb2 = 55
165
Module, après l'appel : nb1 = 33 et nb2 = 55
```

Fonction imbriquée – exemple 1

```
def sommer():
    nb1, nb2 = 33, 55
    print('Fonction englobante avant l\'appel : nb1 =', nb1, 'et nb2 =', nb2)

    def soustraire():
        print('Fonction imbriquée : nb1 =', nb1, 'et nb2 =', nb2)
        return nb2 - nb1

    print(soustraire())
    print('Fonction englobante après l\'appel : nb1 =', nb1, 'et nb2 =', nb2)
    return nb1 + nb2

nb1, nb2 = 3, 5
print('Module, avant l\'appel : nb1 =', nb1, 'et nb2 =', nb2)
print(sommer())
print('Module, après l\'appel : nb1 =', nb1, 'et nb2 =', nb2)
```

Le code ci-dessus affiche :

```
Module, avant l'appel : nb1 = 3 et nb2 = 5
Fonction englobante avant l'appel : nb1 = 33 et nb2 = 55
Fonction imbriquée : nb1 = 33 et nb2 = 55
22
Fonction englobante après l'appel : nb1 = 33 et nb2 = 55
88
Module, après l'appel : nb1 = 3 et nb2 = 5
```

Fonction imbriquée – exemple 2

```
def sommer():
    nb1, nb2 = 33, 55
    print('Fonction englobante avant l\'appel : nb1 =', nb1, 'et nb2 =', nb2)

    def soustraire():
        nb1, nb2 = 13, 15
        print('Fonction imbriquée : nb1 =', nb1, 'et nb2 =', nb2)
        return nb2 - nb1

    print(soustraire())
    print('Fonction englobante après l\'appel : nb1 =', nb1, 'et nb2 =', nb2)
    return nb1 + nb2

nb1, nb2 = 3, 5
print('Module, avant l\'appel : nb1 =', nb1, 'et nb2 =', nb2)
print(sommer())
print('Module, après l\'appel : nb1 =', nb1, 'et nb2 =', nb2)
```

Le code ci-dessus affiche :

```
Module, avant l'appel : nb1 = 3 et nb2 = 5
Fonction englobante avant l'appel : nb1 = 33 et nb2 = 55
Fonction imbriquée : nb1 = 13 et nb2 = 15
2
Fonction englobante après l'appel : nb1 = 33 et nb2 = 55
88
Module, après l'appel : nb1 = 3 et nb2 = 5
```

Fonction imbriquée – exemple 3

```
def sommer():
    nb1, nb2 = 33, 55
    print('Fonction englobante avant l\'appel : nb1 =', nb1, 'et nb2 =', nb2)

    def soustraire():
        nonlocal nb1, nb2
        nb1, nb2 = 13, 15
        print('Fonction imbriquée : nb1 =', nb1, 'et nb2 =', nb2)
        return nb2 - nb1

    print(soustraire())
    print('Fonction englobante après l\'appel : nb1 =', nb1, 'et nb2 =', nb2)
    return nb1 + nb2

nb1, nb2 = 3, 5
print('Module, avant l\'appel : nb1 =', nb1, 'et nb2 =', nb2)
print(sommer())
print('Module, après l\'appel : nb1 =', nb1, 'et nb2 =', nb2)
```

Le code ci-dessus affiche :

```
Module, avant l'appel : nb1 = 3 et nb2 = 5
Fonction englobante avant l'appel : nb1 = 33 et nb2 = 55
Fonction imbriquée : nb1 = 13 et nb2 = 15
2
Fonction englobante après l'appel : nb1 = 13 et nb2 = 15
28
Module, après l'appel : nb1 = 3 et nb2 = 5
```

Fonction imbriquée – exemple 4

```
def sommer():
    nb1, nb2 = 33, 55
    print('Fonction englobante avant l\'appel : nb1 =', nb1, 'et nb2 =', nb2)

    def soustraire():
        global nb1, nb2
        nb1, nb2 = 13, 15
        print('Fonction imbriquée : nb1 =', nb1, 'et nb2 =', nb2)
        return nb2 - nb1

    print(soustraire())
    print('Fonction englobante après l\'appel : nb1 =', nb1, 'et nb2 =', nb2)
    return nb1 + nb2

nb1, nb2 = 3, 5
print('Module, avant l\'appel : nb1 =', nb1, 'et nb2 =', nb2)
print(sommer())
print('Module, après l\'appel : nb1 =', nb1, 'et nb2 =', nb2)
```

Le code ci-dessus affiche :

```
Module, avant l'appel : nb1 = 3 et nb2 = 5
Fonction englobante avant l'appel : nb1 = 33 et nb2 = 55
Fonction imbriquée : nb1 = 13 et nb2 = 15
2
Fonction englobante après l'appel : nb1 = 33 et nb2 = 55
88
Module, après l'appel : nb1 = 13 et nb2 = 15
```

Fonction imbriquée – exemple 5

```
def sommer():
    global nb1, nb2
    nb1, nb2 = 33, 55
    print('Fonction englobante avant l\'appel : nb1 =', nb1, 'et nb2 =', nb2)

    def soustraire():
        global nb1, nb2
        nb1, nb2 = 13, 15
        print('Fonction imbriquée : nb1 =', nb1, 'et nb2 =', nb2)
        return nb2 - nb1

    print(soustraire())
    print('Fonction englobante après l\'appel : nb1 =', nb1, 'et nb2 =', nb2)
    return nb1 + nb2

nb1, nb2 = 3, 5
print('Module, avant l\'appel : nb1 =', nb1, 'et nb2 =', nb2)
print(sommer())
print('Module, après l\'appel : nb1 =', nb1, 'et nb2 =', nb2)
```

Le code ci-dessus affiche :

```
Module, avant l'appel : nb1 = 3 et nb2 = 5
Fonction englobante avant l'appel : nb1 = 33 et nb2 = 55
Fonction imbriquée : nb1 = 13 et nb2 = 15
2
Fonction englobante après l'appel : nb1 = 13 et nb2 = 15
28
Module, après l'appel : nb1 = 13 et nb2 = 15
```

Fonction imbriquée – exemple 6

```
def sommer():
    global nb1, nb2
    nb1, nb2 = 33, 55
    print('Fonction englobante avant l\'appel : nb1 =', nb1, 'et nb2 =', nb2)

    def soustraire():
        nb1, nb2 = 13, 15
        print('Fonction imbriquée : nb1 =', nb1, 'et nb2 =', nb2)
        return nb2 - nb1

    print(soustraire())
    print('Fonction englobante après l\'appel : nb1 =', nb1, 'et nb2 =', nb2)
    return nb1 + nb2

nb1, nb2 = 3, 5
print('Module, avant l\'appel : nb1 =', nb1, 'et nb2 =', nb2)
print(sommer())
print('Module, après l\'appel : nb1 =', nb1, 'et nb2 =', nb2)
```

Le code ci-dessus affiche :

```
Module, avant l'appel : nb1 = 3 et nb2 = 5
Fonction englobante avant l'appel : nb1 = 33 et nb2 = 55
Fonction imbriquée : nb1 = 13 et nb2 = 15
2
Fonction englobante après l'appel : nb1 = 33 et nb2 = 55
88
Module, après l'appel : nb1 = 33 et nb2 = 55
```

Chapitre 7

Types natifs containers – Première partie

Python

Considérations générales

Il y a plusieurs **types** (prédéfinis) **natifs** (*built-in*) intégrés à l'interpréteur Python (donc connus d'office par l'interpréteur) parmi lesquels nous avons déjà étudié les **types natifs** (simples) **numériques**.

On présente par la suite des **types natifs containers** qui permettent de regrouper et de traiter/manipuler plusieurs objets à la fois (et ces **types** sont largement utilisés dans la plupart des programmes Python).

En termes imagés, un **objet container** (ou, simplement, un **container**) peut être vu comme une sorte de boîte (ou sac/enveloppe/coque) qui contient (ou encapsule/regroupe/met ensemble) d'autres objets nommés couramment **éléments** (*items* en anglais).

Il y a des fonctions, des méthodes et des techniques qui permettent :

- d'obtenir des informations concernant les **éléments** d'un objet **container** ;
- de manipuler individuellement ou par tranche les **éléments** d'un objet **container** ;
- d'agir sur l'objet **container** globalement.

Considérations générales (suite)

En général :

- quand un **objet** est **instancié** (créé), on lui assigne une **identité unique** pour toute sa durée de vie ;
- le **type** d'un **objet** (instance) est déterminé **dynamiquement** au **Runtime** et il ne peut plus changer par la suite ;
- cependant, après sa création, l'**état** (ou le **contenu**) d'un **objet** (décrit par ses **attributs données**) :
  - peut changer et l'**objet** est dit alors **mutable/muable** ;
  - ne peut plus changer et l'**objet** est dit alors **immutable/immuable**.

Normalement, un **objet immutable** :

- est plus facile d'accès ;
- "son changement" revient en fait à la création d'un nouvel objet qui est une "copie modifiée" de l'**objet immutable**.

Il y a deux aspects essentiels à prendre en compte quand on travaille avec des **objets** de type **container** :

- le fait que l'objet container est **mutable** (i.e. modifiable) ou **immutable** (i.e. non modifiable) ;
- le fait que les éléments du container sont **ordonnés** ou **pas** (ce qui est essentiel pour l'identification de ces éléments).

Considérations générales (suite)

De plus, les **éléments mêmes** d'un **container** peuvent être des objets **mutables** ou **immutables**.

Par conséquent, par exemple :

- on ne peut pas ajouter ou supprimer un **élément** d'un **container immutable** ;
- on peut changer l'état d'un **élément mutable** d'un **container immutable**.

On utilise par la suite le terme **itérable** (*iterable* en anglais) pour un **objet** (**container**) sur lequel on peut itérer, i.e. un objet qui peut être parcouru (ou traversé) (par exemple avec une boucle **for**).

Plus précisément, quand on passe un objet **itérable** comme (premier) argument à la **fonction native iter()**, on obtient en retour un objet (container) dit **itérateur** (*iterator* en anglais) qui dispose d'une méthode **next()** qui :

- soit retourne l'élément suivant de l'**itérateur** tant qu'un tel élément existe ;
- soit lance une exception de type **StopIteration** s'il n'y a plus d'élément à retourner.

Comme tout objet, un **objet container** peut être affecté (assigné) à une **variable** dont le nom permet de référencer cet objet de manière globale.

Type natifs containers

Les principaux **type natifs containers** sont :

A. les types **séquentiels** (ou, simplement, **séquences**), à savoir les types :

- **A1** **list** (séquences **mutables** et **ordonnées**) ;
- **A2** **tuple** (séquences **immutables** et **ordonnées**) ;
- **A3** **range** (séquences **immutables** et **ordonnées**) ;
- **A4** **str** (séquences **immutables** et **ordonnées**) ;
- **A5** autres **séquences** pour les données binaires (comme **bytes**, **bytearray** et **memoryview**) ;

B. les types **sets**, à savoir les types :

- **B1** **set** (containers **mutables** avec des éléments **uniques**, **non ordonnés** et **immutables**) ;
- **B2** **frozenset** (containers **immutables** avec des éléments **uniques**, **non ordonnés** et **immutables**) ;

C. les types **mapping**, à savoir le (seul) type :

- **C1** **dict** (objets **mutables** nommé **dictionnaires** – *dictionaries* en anglais - avec des éléments **non ordonnés** sous forme de paires clé-valeur (*key-value* en anglais).

Les **séquences** acceptent les **doublons** (i.e. peuvent avoir deux ou plusieurs éléments avec la même valeur).

A. Types séquentiels

Les **séquences** sont des collections **ordonnées** d'éléments qui sont numérotés implicitement avec des **indices entiers à partir de zéro**.

Il y a des **opérations** communes :

- pour tous les **types séquentiels** ;
- pour les **types séquentiels mutables** ;

et elles seront présentées séparément pour le cas concret des **séquences** de type **list**.

Il convient de mentionner que les **séquences immutables** supportent aussi une opération dite de **hachage** (grâce à la fonction **hash()** qui, appelée pour une séquence **mutable**, provoque une erreur de type **TypeError**).

Plus précisément :

- l'appel de la fonction **hash()** avec un objet argument **immutable** retourne une valeur entière dite **valeur de hachage** (*hash value* en anglais) pour cet objet ;
- la **valeur de hachage** est utilisée pour comparer rapidement les objets **immutables** ;
- deux valeurs numériques égales (comme **1** et **1.0**) ont la même **valeur de hachage** (même si leurs **types** sont différents).

A1 Type (classe) list

Un **objet** de **type list** (ou, simplement, une **liste**) est une **séquence mutable** utilisée pour représenter une collection **ordonnée** (et, le plus souvent, **homogène**) d'éléments (pas forcément uniques).

Une **liste** peut être :

- **homogène** si ses éléments sont (plus ou moins) similaires (par polymorphisme) comme type ;
- **hétérogène** si ses éléments sont de types quelconques.

Par exemple, une **liste** peut être utilisée comme :

- un **tableau** (*array* en anglais) de taille variable (et éventuellement hétérogène) afin de grouper et de gérer des objets similaires (ou pas) ;
- une **pile** (*stack* en anglais) qui est une structure de données de type **LIFO** (**L**ast **I**n, **F**irst **O**ut) ;
- une **queue** ou une **file** (*queue* en anglais) qui est une structure de données de type **FIFO** (**F**irst **I**n, **F**irst **O**ut).

En outre, les **listes** peuvent être **imbriquées** (i.e. certains **éléments** d'une **liste** dite **englobante** peuvent être eux-mêmes des **listes** dites **imbriquées** ou **sous-listes**).

A1 Création d'un objet de type list

Un objet de type **list** peut être créé en utilisant :

- une paire de **crochets** (ouvrant – fermant) :
  - vide (afin de créer une **liste vide** - *empty list* en anglais) ;
  - qui entourent un **seul** élément (suivi ou non par une virgule) ;
  - qui entourent une **suite** d'éléments séparés par des **virgules** ;
- le **constructeur** de la **classe list** :
  - sans argument (afin de créer une **liste vide**) ;
  - avec un argument **itérable** (par exemple une **séquence** quelconque, voire une **liste**) ;
- une **liste en compréhension** (*list comprehension* en anglais) qui sera présentée plus tard ;
- des fonctions (comme **sorted()**) et des méthodes (comme **copy()**) qui retournent comme résultats des **listes**.

Dans l'objet **list** créé, on garde :

- l'**ordre** des éléments indiqués entre les crochets ;
- l'**ordre** prévu dans l'argument **itérable** du constructeur.

A1 Création d'un objet de type list - exemples

```
lis = []
print(type(lis))                #affiche <class 'list'>
print(len(lis))                 #affiche 0
print(lis)                      #affiche []

print([10,])                    #affiche [10]
print([10])                     #affiche [10]

print([10, 11, 12])             #affiche [10, 11, 12]

print(list())                   #affiche []

#ci-dessous => TypeError: 'int' object is not iterable
#print(list(10))

#ci-dessous => TypeError: list expected at most 1 arguments, got 3
#print(list(10, 11, 12))

print(list([20, 21, 22]))        #affiche [20, 21, 22]

lis_bis = list(lis)
print(lis_bis)                   #affiche []
print(id(lis) == id(lis_bis))    #affiche False

print(sorted([32, 30, 31]))      #affiche [30, 31, 32]
```

A1 Opérations communes aux séquences – cas du type list

No	Opérateur / Fonction / Méthode	Résultat retourné
1	**lis[i]**	l'élément d'indice i de **lis**
2	**lis[i:j]**	la tranche de **i** à **j** de **lis**
3	**lis[i:j:k]**	la tranche de **i** à **j** avec le pas **k** de **lis**
4	**lis + seq**	la liste obtenue par la concaténation des listes **lis** et **seq**
5	**lis * n** ou **n * lis**	la liste obtenue par la concaténation de **lis** avec elle-même **n** fois
6	**x in lis**	**True** si un élément de **lis** est égal à **x** ou **False** autrement
7	**x not in lis**	**False** si un élément de **lis** est égal à **x** ou **True** autrement
8	**lis.index(x)**	l'indice de la première occurrence de **x** dans **lis**
9	**lis.count(x)**	le nombre total d'occurrences de **x** dans **lis**
10	**len(lis)**	la longueur (la taille ou le nombre d'éléments) de **lis**
11	**min(lis)**	le plus petit élément de **lis**
12	**max(lis)**	le plus grand élément de **lis**

A1 Opérations communes aux séquences – cas du type list (suite)

Dans le tableau précédent, on a présenté les **opérateurs**, les **fonctions** et les **méthodes** **communs** aux **séquences** ainsi que les résultats obtenus à la suite de leur utilisation pour le cas des **listes**.

On a employé les notations suivantes :

- **lis** et **seq** sont des **listes** (ou, plus généralement, des **séquences** de même type) ;
- **x** est un **objet** quelconque (qui respecte les éventuelles restrictions imposées par **lis**) ;
- **i, j** et **k** sont des **indices** entiers ;
- **n** est un **nombre entier**.

On donne par la suite des précisions concernant certaines lignes du tableau précédent.

La ligne **1** => **lis[i]** :

- correspond à l'**indexation simple ;**
- permet d'identifier (et d'accéder à) un **élément** d'une **liste** grâce à son **indice i** placé entre des **crochets** qui suivent l'identificateur de la **liste** ;
- si l'indice **i** est **négatif**, l'indexation se fait à partir de la fin de la liste ;
- si l'indice **i** est trop grand ou trop négatif, une erreur **IndexError** se produit.

A1 Opérations communes aux séquences – cas du type list (suite)

Les lignes **2** et **3** => **lis[i:j]** et **lis[i:j:k]** :

- correspondent à l'**indexation étendue** ;
- permettent d'identifier une **tranche** (*slice* en anglais) d'une **liste** en précisant un **objet** de **type slice** entre les **crochets** qui suivent l'identificateur de la **liste**.

Un objet **slice** peut intervenir comme l'ensemble des indices de l'argument du constructeur de la classe **range** (qui a été déjà présentée).

Dans le contexte d'une **liste** (ou, plus généralement, d'une **séquence**), un objet **slice** :

- provient d'un certain **découpage** de la **liste**, opération appelée aussi **slicing** (terme repris tel quel de l'anglais) ;
- peut correspondre à une **liste** vide ou avec un ou plusieurs éléments de la **liste** découpée.

Syntaxe pour découper une liste

```
lis[start : stop : step]
```

A1 Opérations communes aux séquences – cas du type list (suite)

où :

- **lis** est la liste à découper ;
- **start** précise l'indice du début du **découpage** (et sa valeur est prise en compte dans la **tranche**) ; si **start** est omis, sa valeur par défaut est **0** ;
- **stop** précise l'indice de la fin du **découpage** (et sa valeur n'est pas prise en compte dans la **tranche**) ; si **stop** est omis, sa valeur par défaut est le **nombre d'éléments** de la **liste** ;
- **step** est le pas du découpage et, s'il est omis, sa valeur par défaut est **1** ;
- si les nombres entiers **start**, **stop** et **step** (qui peuvent être positifs, nuls ou négatifs – et dans ce dernier cas l'indexation est considérée à partir de la fin) rendent le découpage impossible, alors la tranche obtenue est une **liste** vide ;
- au moins le premier séparateur deux-points **:** doit apparaître entre les crochets.

La ligne **4 => lis + seq :**

- correspond à la **concaténation** de deux **listes** qui restent inchangées ;
- le résultat obtenu est une nouvelle **liste**.

A1 Opérations communes aux séquences – cas du type list (suite)

La ligne **5** => **lis * n** ou **n * lis** :

- correspond à la multiplication d'une **liste** (qui reste inchangée) par un nombre entier **n** ;
- le résultat obtenu est une nouvelle **liste** obtenue par la concaténation de **lis** avec elle-même **n** fois ;
- si le nombre entier **n** est négatif ou 0, la nouvelle **liste** est vide ;
- si le nombre **n** est réel (de type float), une erreur **TypeError** se produit.

La ligne **8** => **lis.index(x)** :

- si **x** n'est pas élément de **lis**, une erreur **ValueError** se produit ;
- la méthode **index()** peut être appelée aussi avec deux ou trois arguments cas où le **deuxième** argument est un indice à partir duquel la recherche doit **commencer** (y compris lui-même) et le **troisième** argument est un indice avant lequel la recherche doit **finir** (sans lui-même).

La ligne **10** => **len(s)** :

- permet de connaître la **longueur** (ou la taille ou le nombre d'éléments) d'une **liste** ;
- si la **liste** est **vide**, sa longueur est **zéro** ;
- la fonction native **len()** peut être utilisée aussi pour les **types set**, **frozenset** et **dict**.

A1 Opérations communes aux séquences – cas du type list - exemples

```
lis = [10, 11, 12, 13, 14, 15]

print(lis[2])                          #affiche 12

#ci-dessous => IndexError: list index out of range
#print(lis[10])

print(lis[-5])                         #affiche 11

print(lis[-len(lis)])                  #affiche 10

print(lis[1 : 4])                      #affiche [11, 12, 13]

print(lis[1 : 10 : 2])                 #affiche [11, 13, 15]

print(lis[-2 : -6 : -2])               #affiche [14, 12]

print(lis[1 : 5 : -2])                 #affiche []

lis1 = [10, 11]
lis2 = [20, 21]
lis3 = lis1 + lis2
lis4 = lis1 * 3
lis5 = lis1 * (-2)

print(lis1, lis2, lis3)                #affiche [10, 11] [20, 21] [10, 11, 20, 21]
print(lis4)                            #affiche [10, 11, 10, 11, 10, 11]
print(lis5)                            #affiche []
```

A1 Opérations communes aux séquences – cas du type list - exemples

```
lis1 = [10, 11]

#ci-dessous => TypeError: can't multiply sequence by non-int of type 'float'
#lis1 * 1.5

print(10 in lis1)                              #affiche True

print(15 not in lis1)                          #affiche True

lis = [10, -3, 10, 22, 15, 10]

print(lis.index(10))                           #affiche 0

#ci-dessous => ValueError: 33 is not in list
#lis.index(33)

print(lis.index(10, 1))                        #affiche 2

print(lis.index(10, 3, 11))                    #affiche 5

#ci-dessous => ValueError: 10 is not in list
#lis.index(10, 3, 5)

print(lis.count(10))                            #affiche 3

print(len(lis))                                 #affiche 6

print(min(lis))                                 #affiche -3

print(max(lis))                                 #affiche 22
```

A1 Opérations communes aux séquences mutables – cas du type list

No	Opérateur / Mot clé / Méthode	Effet / Résultat retourné
1	**lis[i] = x**	**x** remplace l'élément d'indice **i** de **lis**
2	**lis[i:j] = it**	le contenu de **it** remplace la tranche de **i** à **j** de **lis**
3	**lis[i:j:k] = it**	le contenu de **it** remplace la tranche de **i** à **j** avec le pas **k** de **lis**
4	**lis.append(x)**	ajoute **x** à la fin de **lis**
5	**lis.insert(i, x)**	insère **x** à la position d'indice **i** dans **lis** – comme **lis[i:i] = [x]**
6	**lis.extend(it)** ou **lis += it**	étend **lis** avec le contenu de **it**
7	**lis *= n**	met à jour **lis** avec son ancien contenu répété **n** fois
8	**del lis[i:j:k]**	supprime la tranche de **i** à **j** avec le pas **k** de **lis**
9	**lis.pop(i)**	retourne et supprime l'élément d'indice **i** de **lis**
10	**lis.remove(x)**	supprime le premier élément de **lis** qui est égal à **x**
11	**lis.clear()**	supprime tous les éléments de **lis** - comme **del lis[:]**
12	**lis.copy()**	retourne une copie superficielle de **lis** – comme **lis[:]**
13	**lis.reverse()**	inverse l'ordre des éléments de **lis** sur place

A1 Opérations communes aux séquences mutables – cas du type list (suite)

Dans le tableau précédent, on a présenté les **opérateurs** et les **fonctions communs** aux **séquences mutables** ainsi que les effets/résultats obtenus à la suite de leur utilisation pour le cas des **listes**.

Les notations mentionnées plus tôt restent presque les mêmes mais **lis** peut être une **liste** (comme auparavant) ou, plus généralement, une **séquence** qui est, cette fois, **mutable**.

De plus, **it** est un **objet itérable**.

A part les méthodes **pop()** et **copy()**, les autres **méthodes** du tableau précédent **modifient** l'objet **list appelant** et **retournent None**.

On donne par la suite des précisions concernant certaines lignes du tableau précédent.

Les lignes **1**, **2** et **3** => **lis[i] = x** et **lis[i:j] = it** et **lis[i:j:k] = it** :

- permettent de modifier un **élément** ou une **tranche** (*slice*) d'une **liste** ;
- afin de remplacer une **tranche** avec un pas **k** différent de 1, le contenu de l'itérateur **it** doit avoir le même nombre d'éléments que la **tranche** remplacée (car, sinon, une erreur **ValueError** se produit).

A1 Opérations communes aux séquences mutables – cas du type list (suite)

Les lignes **4** et **6** => **lis.append(x)** vs **lis.extend(it)** :

- l'argument **x** peut être un objet quelconque, en particulier une **liste** qui sera ajoutée comme un seul élément à la fin de la **liste lis** ;
- l'argument **it** :
  - doit être un objet **itérable**, en particulier une **liste**, dont les éléments seront ajoutés individuellement à la fin de la **liste lis** ;
  - ne peut pas être un objet **non itérable** (comme un nombre entier ou réel) car, dans un tel cas, une erreur **TypeError** se produit.

La ligne **5** => **lis.insert(i, x)** :

- l'objet **x** est inséré dans la **liste lis** à la position d'indice **i** ;
- l'ancien élément d'indice **i** ainsi que tous les éléments suivants sont déplacés d'une position vers la droite (leurs indices sont incrémentés d'une unité).

A1 Opérations communes aux séquences mutables – cas du type list (suite)

La ligne **9** => **lis.pop(i)** :

- retourne l'élément d'indice **i** qui est ensuite supprimé dans la **liste lis** (et les éléments situés à droite sont déplacés d'une position vers la gauche et leurs indices sont diminués d'une unité) ;
- la méthode peut être appelée aussi sans argument **lis.pop()** et, dans ce cas qui est équivalent à l'appel **lis.pop(-1)**, le dernier élément de **lis** est retourné et ensuite supprimé de la **liste**.

En outre, la classe **list** fournit la méthode **sort()** qui trie l'objet **list** appelant **sur place** (*in place* en anglais) en utilisant l'opérateur de comparaison **<**.

Il convient de souligner que :

- la méthode **sort()** change l'ordre des éléments de la **liste** appelante **sur place**, sans créer une nouvelle **liste** ;
- en revanche, afin de laisser la **liste à trier** inchangée et de créer une **nouvelle liste triée**, il faut appeler la fonction native **sorted()** avec la **liste à trier** comme argument.

A1 Opérations communes aux séquences mutables – cas du type list - exemples

```
lis = [10, 11, 12, 13, 14, 15]

lis[2] = 22
print(lis)                  #affiche [10, 11, 22, 13, 14, 15]

lis[1:3] = [31, 32]
print(lis)                  #affiche [10, 31, 32, 13, 14, 15]

lis[1:5:2] = [41, 43]       #affiche [10, 41, 32, 43, 14, 15]
print(lis)

lis[:2] = [50, 51, 53, 54]
print(lis)                  #affiche [50, 51, 53, 54, 32, 43, 14, 15]

lis[::3] = [60, 63, 66]
print(lis)                  #affiche [60, 51, 53, 63, 32, 43, 66, 15]

lis[:] = [70, 71, 72, 73, 74, 75]
print(lis)                  #affiche [70, 71, 72, 73, 74, 75]

lis.insert(3, 83)
print(lis)                  #affiche [70, 71, 72, 83, 73, 74, 75]

lis.insert(-2, 94)
print(lis)                  #affiche [70, 71, 72, 83, 73, 94, 74, 75]
```

A1 Opérations communes aux séquences mutables – cas du type list - exemples

```
lis1 = [10, 11, 12]
lis2 = [20, 21]

lis1.append(23)
print(lis1)                    #affiche [10, 11, 12, 23]

#ci-dessous => TypeError: 'int' object is not iterable
#lis1.extend(44)

lis1.append(lis2)
print(lis1)                    #affiche [10, 11, 12, 23, [20, 21]]

lis1.extend(lis2)
print(lis1)                    #affiche [10, 11, 12, 23, [20, 21], 20, 21]

lis1 += lis2
print(lis1)                    #affiche [10, 11, 12, 23, [20, 21], 20, 21, 20, 21]
print(lis2)                    #affiche [20, 21]

lis2 *= 2
print(lis2)                    #affiche [20, 21, 20, 21]

del lis1[2:7:2]
print(lis1)                    #affiche [10, 11, 23, 20, 20, 21]

lis.clear()
print(lis)                     #affiche []
```

A1 Opérations communes aux séquences mutables – cas du type list - exemples

```
lis = [10, 11, 12, 13, 14, 15]
print(lis.pop())                #affiche 15
print(lis)                      #affiche [10, 11, 12, 13, 14]

print(lis.pop(2))               #affiche 12
print(lis)                      #affiche [10, 11, 13, 14]

print(lis.pop(-2))              #affiche 13
print(lis)                      #affiche [10, 11, 14]

#ci-dessous => IndexError: pop index out of range
#lis.pop(10)

lis =[10, 11, 10, 12, 10, 13, 14]
print(lis.remove(10))            #affiche None
print(lis)                       #affiche [11, 10, 12, 10, 13, 14]

lis_bis = lis.copy()
print(lis)                       #affiche [11, 10, 12, 10, 13, 14]
print(lis_bis)                   #affiche [11, 10, 12, 10, 13, 14]
print(lis == lis_bis)            #affiche True
print(id(lis) == id(lis_bis))    #affiche False

print(lis.reverse())             #affiche None
print(lis)                       #affiche [14, 13, 10, 12, 10, 11]
```

A1 Liste de listes – copie superficielle vs copie profonde

Etant donné que les éléments d'une **liste** peuvent être des objets quelconques, ils peuvent être eux-mêmes des **listes** et, dans ce cas particulier, on parle des **listes imbriquées** (*nested lists* en anglais) :

- l'objet container global est la **liste englobante** ;
- ses éléments sont des **listes imbriquées** ou des **sous-listes**.

En particulier, une **liste** de **listes** avec un seul niveau d'imbrication peut être utilisée en tant que tableau bidimensionnel régulier (comme une matrice) ou pas.

Une nouvelle **liste** est une **copie superficielle** (*shallow copy* en anglais) d'une **liste** d'origine si :

- les deux listes ont des identités différentes ;
- les éléments correspondants de chaque **liste** ont les **même identités** et, donc, les **mêmes valeurs**.

Par conséquent, pour une **liste copie superficielle** et la **liste** d'origine :

- si tous les éléments sont des objets **immutables**, un changement de la valeur d'un élément par une des **listes** n'a pas d'effet sur l'autre **liste** ;
- en revanche, si un élément est **mutable** (par exemple une **sous-liste**), un changement du contenu (i.e. de l'état) de cet élément (de la **sous-liste**) par une des **listes** est répercuté aussi sur l'autre **liste**.

A1 Liste de listes – copie superficielle vs copie profonde (suite)

Une nouvelle **liste** est une **copie profonde** (*deep copy* en anglais) d'une **liste** d'origine si :

- les deux listes ont des identités différentes ;
- les éléments **immutables** correspondants des chaque **liste** ont les **même identités** et, donc, les **mêmes valeurs** ;
- les éléments **mutables** correspondants ont des **identités différentes** mais les **mêmes contenus** (i.e. états) et ceci de manière récursive.

Par conséquent, pour une **liste copie profonde** et la **liste** d'origine, un changement du contenu d'un élément (**mutable**) par une des **listes** n'a pas d'effet sur l'autre **liste**.

La méthode **copy()** de la **classe list** retourne une **liste copie superficielle** de la **liste** appelante.

La fonction **deepcopy()** du **module** standard **copy** retourne une **liste copie profonde** de la **liste** argument.

A1 Liste de listes – copie superficielle vs copie profonde - exemples

```
lis =[10, [11, 12]]
lis_superf = lis.copy()
print(id(lis) == id(lis_superf))            #affiche False
print(id(lis[0]) == id(lis_superf[0]))      #affiche True
print(lis[0] == lis_superf[0])              #affiche True
print(id(lis[1]) == id(lis_superf[1]))      #affiche True
print(lis[1] == lis_superf[1])              #affiche True
lis[0] = 20
print(lis_superf)                           #affiche [10, [11, 12]]
lis[1][0] = 31
print(lis_superf)                           #affiche [10, [31, 12]]
from copy import deepcopy
lis =[10, [11, 12]]
lis_prof = deepcopy(lis)
print(id(lis) == id(lis_prof))              #affiche False
print(id(lis[0]) == id(lis_prof[0]))        #affiche True
print(lis[0] == lis_prof[0])                #affiche True
print(id(lis[1]) == id(lis_prof[1]))        #affiche False
print(lis[1] == lis_prof[1])                #affiche True
lis[0] = 20
print(lis_prof)                             #affiche [10, [11, 12]]
lis[1][0] = 31
print(lis_prof)                             #affiche [10, [11, 12]]
```

A1 Liste de listes comme tableau bidimensionnel régulier– exemples

```
matrice = [[11, 12, 13], [21, 22, 23], [31, 31, 33]]

print(type(matrice))            #affiche <class 'list'>
print(matrice)                  #affiche [[11, 12, 13], [21, 22, 23], [31, 31, 33]]
print(len(matrice))             #affiche 3

print(type(matrice[0]))         #affiche <class 'list'>
print(matrice[0])               #affiche [11, 12, 13]
print(len(matrice[0]))          #affiche 3

print(type(matrice[1][2]))      #affiche <class 'int'>
print(matrice[1][2])            #affiche 23
#ci-dessous => TypeError: object of type 'int' has no len()
#print(len(matrice[1][2]))

matrice[2] = 99
print(matrice)                  #affiche [[11, 12, 13], [21, 22, 23], 99]

matrice[0][1] = 42
print(matrice)                  #affiche [[11, 42, 13], [21, 22, 23], 99]

lis = matrice[0]
print(lis)                      #affiche [11, 42, 13]

lis[2] = 53
print(matrice)                  #affiche [[11, 42, 53], [21, 22, 23], 99]
```

A1 Liste en compréhension

Un **objet** de **type list** est un **container mutable** qui peut être parcouru (ou traversé) à l'aide d'une boucle **for** qui permet de visiter chaque **élément** de la **liste** afin de :

- connaître (lire) sa valeur ;
- modifier (écrire) sa valeur (si l'élément est **mutable**) ;
- le remplacer par un autre objet.

Etant donné qu'assez souvent le parcours d'une **liste** a comme but la création d'une **nouvelle liste**, Python met à la disposition des programmeurs une façon élégante (et vraiment pythonique) pour réaliser une telle opération.

Il s'agit d'une approche dite **liste en compréhension** (*list comprehension* en anglais) qui correspond assez bien à une notation ensembliste du genre $B = \{f(x) \mid x \in A\}$ où A est un ensemble donné et $f(x)$ une fonction choisie convenablement.

Les **listes en compréhension** remplacent d'une certaine façon l'utilisation des fonctions **lambda** ainsi que des fonctions **map()**, **filter()** et **reduce()** (que nous n'étudions pas).

A1 Liste en compréhension (suite)

Syntaxe d'une liste en compréhension

```
[new_item for item in container suite_clauses]
```

où :

- **new_item** est une expression quelconque (qui peut utiliser ou pas la valeur de la variable **item**) ;
- **item** est un nom (identificateur valide choisi librement) d'une variable qui, à tour de rôle, correspond à chaque élément du **container** ;
- **container** est un objet **container** ou, plus généralement, **itérable** ;
- **suite_clauses** est une suite d'aucune, une ou plusieurs **clauses for** ou **if** (voir les exemples suivants).

Il convient de mentionner que la variable **item** est une **variable locale** à la **liste en compréhension**.

En outre, d'une manière similaire, on peut créer aussi des **sets en compréhension** ou des **dictionnaires en compréhension** en utilisant des **accolades** à la place des **crochets** (et aussi des **générateurs en compréhension** en utilisant des **parenthèses** à la place des **crochets**).

A1 Liste en compréhension - exemples

```
lis = [10, 13, -24, 11, 34, 0, 55]

lis_fois_dix = [nb*10 for nb in lis]
print(lis_fois_dix)                    #affiche [100, 130, -240, 110, 340, 0, 550]

#ci-dessous => NameError: name 'nb' is not defined
#print(nb)

lis_paire = [nb for nb in lis if nb%2==0]
print(lis_paire)                       #affiche [10, -24, 34, 0]

l_temp = [10.5, -3, 5, -7.5, 0, -15.3]
l_temp_neg = [temp for temp in l_temp if temp < 0]
print(l_temp_neg)                      #affiche [-3, -7.5, -15.3]

l_celsius = [20, 19.3, 24.25]
l_kelvin = [temp + 273.15 for temp in l_celsius]
print(l_kelvin)                        #affiche [293.15, 292.45, 297.4]

couleurs = ['rouge', 'noire']
objets = ['voiture', 'encre', 'chaussure']
objets_en_couleurs = [obj + ' ' + coul for obj in objets for coul in couleurs]

#ci-dessous => affiche : ['voiture rouge', 'voiture noire', 'encre rouge',
#'encre noire', 'chaussure rouge', 'chaussure noire']
print(objets_en_couleurs)
```

A2 Type (classe) tuple

Un **objet** de **type tuple** (ou, simplement, un **tuple**) est une **séquence immutable** utilisée pour représenter une collection **ordonnée** (et, le plus souvent, **hétérogène**) d'éléments.

Un objet de type **tuple** peut être créé (normalement) en utilisant :

- une paire de **parenthèses** (ouvrante – fermante) :
  - vide (afin de créer un **tuple vide** - *empty tuple* en anglais) ;
  - qui entourent un **seul** élément suivi par une **virgule** ;
  - qui entourent une **suite** d'éléments séparés par des **virgules** ;
- le **constructeur** de la **classe tuple** :
  - sans argument (afin de créer un **tuple vide**) ;
  - avec un argument **itérable** (par exemple une **séquence** quelconque, voire une liste ou un tuple) ;
- des fonctions et des méthodes qui retournent comme résultats des **tuples**.

Dans l'objet **tuple** créé, on garde :

- l'**ordre** des éléments indiqués entre les parenthèses ;
- l'**ordre** prévu dans l'argument **itérable** du constructeur.

A2 Type (classe) tuple (suite)

Il convient de mentionner que, dans le premier cas de création d'un **objet tuple** indiqué ci-dessus :

- la présence de la **virgule** ou, respectivement, des **virgules** est obligatoire ;
- l'utilisation des **parenthèses** est facultative grâce à un mécanisme nommé **emballage** (*packing/boxing* en anglais) sauf pour :
  - le cas du **tuple vide** ;
  - les cas où les **parenthèses** évitent une ambiguïté syntaxique (comme dans l'appel d'une fonction qui a comme argument effectif un **tuple** - voir les exemples ci-dessous).

En particulier, l'**emballage** permet à une fonction de retourner plusieurs valeurs (séparées par des **virgules**) à la fois.

Il y a un mécanisme complémentaire au **packing** nommé **déballage** (*unpacking/unboxing* en anglais) qui agit dans l'autre sens et permet d'affecter un **tuple** à plusieurs variables séparées par des virgules (à condition de prévoir autant de variables que d'éléments dans le **tuple**).

En particulier, les mécanismes **packing/unpacking** permettent de récupérer les valeurs retournées par une certaine fonction à l'aide d'une **seule** affectation.

A2 Type (classe) tuple (suite)

En outre, dans le **déballage** d'un **tuple**, à gauche de l'affectation, on peut nommer :

- chaque variable qui ne nous intéresse pas par un **tiret bas** (*underscore*) ;
- un groupe (une **liste**) de variables par un **identificateur** précédé par le caractère *****.

Les **opérations communes** aux **séquences** sont (bien sûr) valables pour les objets de type **tuple** (dont la numérotation des éléments commence à zéro et l'accès aux éléments se fait en utilisant les **crochets**).

Il faut garder à l'esprit le fait que les objets de type **tuple** sont **immutables** et donc on ne peut pas, par exemple, ajouter/supprimer/changer l'ordre ou les identités des éléments dans un tel **container**.

Toute tentative de modifier l'identité d'un élément d'un **tuple** (par une affectation, par exemple) produit une erreur **TypeError**.

Par conséquent, pour "modifier" un **tuple** il faut créer un nouveau **tuple** à partir du **tuple à modifier** et en tenant compte de la modification voulue.

En revanche, si un **élément** d'un tuple est un **objet** **mutable** (par exemple une **liste**), on peut modifier le contenu (i.e. l'état) de cet objet car son identité ne change pas.

A2 Type (classe) tuple - exemples

```
tup = ()
print(type(tup))                #affiche <class 'tuple'>
print(len(tup))                 #affiche 0
print(tup)                      #affiche ()

tup = (10)
print(type(tup))                #affiche <class 'int'>
print(tup)                      #affiche 10

tup = (10,)
print(type(tup))                #affiche <class 'tuple'>
print(tup)                      #affiche (10,)

print(10, 11, 12)               #affiche 10 11 12

print((10, 11, 12))             #affiche (10, 11, 12)

tup = 10, 11, 12
print(type(tup))                #affiche <class 'tuple'>
print(tup)                      #affiche (10, 11, 12)

print(tuple())                  #affiche ()

#ci-dessous => TypeError: 'int' object is not iterable
#tuple(10)
```

A2 Type (classe) tuple - exemples

```
#ci-dessous => TypeError: tuple expected at most 1 arguments, got 3
#tuple(10, 11, 12)

tup = tuple((10, 11, 12))
print(tup)                              #affiche (10, 11, 12)

tup_bis = tup
print(id(tup) == id(tup_bis))           # affiche True

tup_ter = tuple(tup)
print(id(tup) == id(tup_ter))           #affiche True

a, b, c = tup
print('a =', a, 'b =', b, 'c =', c)     #affiche a = 10 b = 11 c = 12

x, x, x = tup
print(x)                                #affiche 12

id_avant = id(tup)
tup += (13, 14, 15)
id_apres = id(tup)

print(tup)                              #affiche (10, 11, 12, 13, 14, 15)
print(id_avant == id_apres)             #affiche False

a, _, _, *b, c = tup
print('a =', a, 'b =', b, 'c =', c)     #affiche a = 10 b = [13, 14] c = 15
```

A2 Type (classe) tuple – exemples

```
tup = (10, [21, 22, 23], 12)
#ci-dessous => ValueError: too many values to unpack (expected 2)
#a, b = tup
#ci-dessous => TypeError: 'tuple' object does not support item assignment
#tup[2] = 22
print(tup)                              #affiche (10, [21, 22, 23], 12)
print(len(tup))                         #affiche 3
print(tup[1])                           #affiche [21, 22, 23]
print(tup[1][2])                        #affiche 23
id_1 = id(tup[1])
tup[1].extend([24, 25])
print(tup)                              #affiche (10, [21, 22, 23, 24, 25], 12)
print(tup[1])                           #affiche [21, 22, 23, 24, 25]
print(len(tup))                         #affiche 3
print(id(tup[1]) == id_1)               #affiche True
```

A3 Type (classe) range

Un objet de type **range** est une **séquence immutable** et **ordonnée** de **nombres entiers** utilisée le plus souvent dans l'en-tête d'une boucle **for** afin de préciser le nombre de tours de boucle à effectuer.

Le **type range** a été déjà présenté dans le cadre de l'étude des boucles.

Les **opérations communes** aux **séquences** sont valables pour les objets de type **range** (à part la **concaténation** et la **répétition** – voir les exemples ci-dessous).

```
ran1 = range(10, 20, 2)
print(ran1)                 #range(10, 20, 2)
print(list(ran1))           #affiche [10, 12, 14, 16, 18]

print(ran1[3])              #affiche 16

print(ran1.index(16))       #affiche 3

ran2 = range(20, 30, 3)

#ci-dessous => TypeError: unsupported operand type(s) for +: 'range' and 'range'
#ran = ran1 + ran2

#ci-dessous => TypeError: unsupported operand type(s) for *: 'range' and 'int'
#ran = ran1 * 3
```

Chapitre 8

Types natifs containers – Deuxième partie

Python

B. Types sets

Les types **sets** correspondent à des **collections non ordonnées** d'objets (d'éléments) :

- **uniques** (donc **distincts**) ;
- **immutables** (donc **hachables**).

Étant donné que les éléments des **containers sets** ne sont **pas ordonnés** :

- l'indexation (*indexing*),
- le découpage (*slicing*) et
- d'autres opérations spécifiques aux **séquences**

ne sont pas valables pour les **sets**.

En revanche, un **container set** (qui correspond en fait à la notion mathématique d'**ensemble**) peut être **parcouru** avec une **boucle** standard : **for el in container**.

Python prévoit deux **types natifs** pour les **sets** :

B1 la **classe set** (containers **mutables** avec des éléments **uniques**, **non ordonnés** et **immutables**) ;

B2 la **classe frozenset** (containers **immutables** avec des éléments **uniques**, **non ordonnés** et **immutables**).

B1 Type (classe) set

Un objet de type **set** (ou, simplement, un **set**) est une **collection** **mutable** et **non ordonnée** d'objets **uniques** et **immutables**.

Étant donné qu'elle est **mutable**, la **collection** correspondant à un objet de type **set** peut changer de contenu (grâce à des méthodes comme **add()** ou **remove()**).

Par conséquent, un **objet** **mutable** de type **set** ne peut pas être haché et, donc, ne peut pas être **élément** d'un autre **set** (ou **clé** d'un **dictionnaire**).

Un objet de type **set** peut être créé en utilisant :

- une paire d'**accolades** (ouvrante – fermante) :
  - qui ne peut **pas** être **vide** (car de cette façon on crée un **dictionnaire** **vide**) ;
  - qui entourent un **seul** élément ou une **suite** d'éléments séparés par des virgules ;
- le **constructeur** de la classe **set** :
  - sans argument (afin de créer un set vide – *empty set* en anglais) ;
  - avec un seul élément **itérable** (par exemple une **liste** ou un **set**) dont les éléments sont **immutables** (donc **hachables**).

B1-2 Types (classes) set (suite) et frozenset

En outre, un objet de type **set** peut être créé aussi en utilisant :

- un **set en compréhension** ;
- des **fonctions** et des **méthodes** qui retournent comme résultats des **sets**.

Un objet de type **frozenset** (ou, simplement, un **frozenset**) est une **collection immutable** (dont le contenu ne peut plus être modifié après sa création) et **non ordonnée** d'objets **uniques** et **immutables**.

Par conséquent, un **frozenset** peut être haché et peut être élément d'un **set** ou d'un **frozenset** (ou **clé** d'un **dictionnaire**).

Un objet de type **frozenset** :

- ne peut **pas** être créé en utilisant une paire d'**accolades** (qui sont réservées aux objets de type **set** et aux **dictionnaires**) ;
- peut être créé en utilisant :
  - le **constructeur** de la classe **frozenset** qui fonctionne comme le **constructeur** de la classe **set** ;
  - des **fonctions** et des **méthodes** qui retournent comme résultats des **frozensets**.

B1-2 Création des objets de types set et frozenset – exemples

```
ref = {}
print(type(ref))                                #affiche <class 'dict'>

ens = {1}
print(type(ens))                                #affiche <class 'set'>
print(len(ens))                                 #affiche 1
print(ens)                                      #affiche {1}

ens = {1, 3, 2}
print(type(ens))                                #affiche <class 'set'>
print(len(ens))                                 #affiche 3
print(ens)                                      #affiche {1, 2, 3}

#ci-dessous => TypeError: unhashable type: 'set'
#ens_ko = {1, {2, 3}}

ens = set()
print(type(ens))                                 #affiche <class 'set'>
print(len(ens))                                  #affiche 0
print(ens)                                       #affiche set()

#ci-dessous => TypeError: set expected at most 1 argument, got 3
#ens_ko = set('un', 'trois', 'deux')
```

B1-2 Création des objets de types set et frozenset – exemples (suite)

```
ens = set(['un', 'trois', 'deux', 'trois'])
print(type(ens))                              #affiche <class 'set'>
print(len(ens))                               #affiche 3
print(ens)                                    #affiche {'un', 'deux', 'trois'}

ens = {el for el in {1,2,3,4} if el%2==0}
print (type(ens))                             #affiche <class 'set'>
print(ens)                                    #affiche {2, 4}

ens = {i for i in range(5) if i%2!=0}
print(type(ens))                              #affiche <class 'set'>
print(ens)                                    #affiche {1, 3}

fens = frozenset((1, 'deux', 3))
print(type(fens))                             #affiche <class 'frozenset'>
print(fens)                                   #affiche frozenset({1, 3, 'deux'})
```

B Opérations avec des sets (objets de type set ou frozenset)

No	Opérateur / Fonction / Méthode	Résultat retourné
1	**len(se)**	le nombre d'éléments (la cardinalité) de **se**
2	**el in se**	**True** si **el** appartient au **se** ou **False** autrement
3	**el not in se**	**False** si **et** appartient à **se** ou **True** autrement
4	**s_this.isdisjoint(s_that)**	**True** si **s_this** et **s_that** sont disjoints ou **False** autrement
5	**s_this.issubset(s_that)**	**True** si **s_this** est un sous-ensemble de **s_that** ou **False** autrement
6	**s_this <= s_that**	comme ci-dessus
7	**s_this < s_that**	**True** si **s_this** est un sous-ensemble strict de **s_that** ou **False** autrement
8	**s_this.issuperset(s_that)**	**True** si **s_that** est un sous-ensemble de **s_this** ou **False** autrement
9	**s_this >= s_that**	comme ci-dessus
10	**s_this > s_that**	**True** si **s_that** est un sous-ensemble strict de **s_this** ou **False** autrement

B Opérations avec des sets (objets de type set ou frozenset) (suite)

No	Opérateur / Fonction / Méthode	Résultat retourné
11	**s_this.union(*s_that)**	un nouveau set qui est la réunion du **s_this** et des sets ***s_that**
12	**s_this | s_that_1 | …**	comme ci-dessus
13	**s_this.intersection(*s_that)**	un nouveau set qui est l'intersection du **s_this** et des sets ***s_that**
14	**s_this & s_that_1 & …**	comme ci-dessus
15	**s_this.difference(*s_that)**	un nouveau set qui est la différence entre **s_this** et les sets ***s_that**
16	**s_this - s_that_1 - …**	comme ci-dessus
17	**s_this.symmetric_difference(s_that)**	un nouveau set avec des éléments du **s_this** ou du **s_that** mais pas des deux
18	**s_this ^ s_that**	comme ci-dessus
19	**s_this.copy()**	une copie superficielle du **s_this**

B Opérations avec des sets (objets de type set ou frozenset) (suite)

Dans le tableau précédent (qui s'étale sur deux pages), on a présenté des **opérateurs**, des **fonctions** et des **méthodes** qui permettent de manipuler des **sets** (i.e. des objets de type **set** ou **frozenset**).

On a utilisé les notations suivantes :

- **se** pour un objet **set** ;
- **s_this** pour un objet de type **set** qui est l'objet **appelant** d'une **méthode** ;
- **s_that** pour un objet de type **set** qui est un objet **argument** d'une **méthode** ;
- ***s_that** pour un nombre variable d'objets de type **set** qui sont des objets **arguments** d'une **méthode**.

On donne par la suite des précisions concernant certaines lignes du tableau précédent.

Il convient de mentionner qu'aucune des **méthodes** présentes dans ce tableau ne retourne **None**.

La ligne **4** => **s_this.isdisjoint(s_that)** :

- les ensembles **s_this** et **s_that** sont **disjoints** :
  - s'ils n'ont pas d'élément en commun ;
  - si et seulement si leur intersection est l'ensemble vide.

B Opérations avec des sets (objets de type set ou frozenset) (suite)

La ligne **5** et **6** => **s_this.issubset(s_that)** et, respectivement, **s_this <= s_that** :

- en fait, on teste si chaque élément de l'ensemble **s_this** appartient à l'ensemble **s_that**.

La ligne **7** => **s_this < s_that** :

- en fait, on teste si **s_this <= s_that et s_this != s_that**.

La ligne **11** et **12** => **s_this.union(*s_that)** et, respectivement, **s_this | s_that_1 | …** :

- les éléments de l'ensemble retourné sont les éléments de l'ensemble **s_this ou** des autres ensembles ***s_that**.

La ligne **13** et **14** => **s_this.intersection(*s_that)** et, respectivement, **s_this & s_that_1 & …** :

- les éléments de l'ensemble retourné sont les éléments communs de l'ensemble **s_this et** des autres ensembles ***s_that**.

La ligne **15** et **16** => **s_this.difference(*s_that)** et, respectivement, **s_this - s_that_1 - …** :

- les éléments de l'ensemble retourné sont les éléments de l'ensemble **s_this et** qui ne sont pas dans les autres ensembles ***s_that**.

B Opérations avec des sets (objets de type set ou frozenset) (suite)

La ligne **17** => **s_this.symmetric_difference(s_that)** :

- les éléments de l'ensemble retourné sont les éléments de l'ensemble **s_this** **ou** de l'ensemble **s_that** mais **pas** des deux **à la fois**.

Deux containers de type **set** ou **frozenset** peuvent être comparés avec l'**opérateur ==** qui retourne **True** si et seulement si chaque élément de chaque container appartient aussi à l'autre container (i.e. chaque opérande est un sous-ensemble de l'autre opérande).

La **comparaison** d'un objet **set** avec un objet **frozenset** est basée seulement sur leurs éléments.

Le **type** de l'objet **retourné** par des opérations binaires qui mélangent des objets **set** avec des objets **frozenset** est le même que le type du **premier opérande**.

La **relation d'ordre** entre les **sets** n'est que **partielle** (par exemple, pour deux **sets** **se1** et **se2** non vides et disjoints les trois opérations suivantes retournent **False** : **se1 < se2**, **se1 == se2** et **se1 > se2**).

B Opérations avec des sets (objets de type set ou frozenset) – exemples

```
ens1 = {1, 2, 3, 4, 5}

print(len(ens1))                                    #afiche 5

#les 2 lignes ci-dessous => TypeError: 'set' object is not subscriptable
#print(ens1[0])
#print(ens1[1:5:2])

print(3 in ens1)                                    #affiche True
print(3 not in ens1)                                #affiche False

ens1 = {1, 2, 3, 4, 5}
ens2 = {-1, 0, -2}
ens3 = {2, 4}
ens4 = {-1, 0, 2, 4, 5}

print(ens1.isdisjoint(ens2))                        #affiche True

print(ens1.issubset(ens3))                          #affiche False
print(ens3 <= ens1)                                 #affiche True
print(ens3 < ens1)                                  #affiche True

print(ens1.issuperset(ens3))                        #affiche True
print(ens3 > ens1)                                  #affiche False
```

B Opérations avec des sets (objets de type set ou frozenset) – exemples (suite)

```
print({1,3}.union({0,-1}, {1,7}))                    #affiche {0, 1, 3, 7, -1}

print({1,5,7,3} & {-2,5,3} & {1,3,7,5})              #affiche {3, 5}

print({1,2,3,4,5} - {0,2} - {1,4})                   #affiche {3, 5}

print({1,2,3}.symmetric_difference({-1,1,2}))        #affiche {3, -1}

print({1,2,3}^{-1,1,2})                              #affiche {3, -1}

ens1 = {1,2,3}

ens2 = ens1

print(ens1 == ens2)                                   #affiche True

print(id(ens1) == id(ens2))                           #affiche True

ens3 = ens1.copy()

print(ens1 == ens3)                                   #affiche True

print(id(ens1) == id(ens3))                           #affiche False
```

B Opérations avec des objets mutables de type set

No	Opérateur / Fonction / Méthode	Effet / Résultat retourné
1	**s_this.update(*s_that)**	met à jour **s_this** comme la réunion du **s_this** et des sets ***s_that**
2	**s_this |= s_that_1 | …**	comme ci-dessus
3	**s_this.intersection_update(*s_that)**	met à jour **s_this** comme l'intersection du **s_this** et des sets ***s_that**
4	**s_this &= s_that_1 & …**	comme ci-dessus
5	**s_this.difference_update(*s_that)**	met à jour **s_this** comme la différence entre **s_this** et les sets ***s_that**
6	**s_this -= s_that_1 | …**	comme ci-dessus
7	**s_this.symmetric_difference_update(s_that)**	met à jour **s_this** en ne gardant que des éléments du **s_this** ou du **s_that** mais pas des deux
8	**s_this ^= s_that**	comme ci-dessus

B Opérations avec des objets mutables de type set (suite)

No	Opérateur / Fonction / Méthode	Effet / Résultat retourné
9	**s_this.add(el)**	ajoute l'élément **el** au set **s_this**
10	**s_this.remove(el)**	retire l'élément **el** du set **s_this** s'il est dans **s_this** ou lance une exception **KeyError** si **el** n'est pas dans **s_this**
11	**s_this.discard(el)**	retire l'élément **el** du set **s_this** s'il est dans **s_this** ou reste sans effet si **el** n'est pas dans **s_this**
12	**s_this.pop()**	retire et retourne un élément arbitraire du set **s_this** ou lance une exception **KeyError** si **s_this** est vide
13	**s_this.clear()**	supprime tous les éléments du set **s_this**

Dans le tableau ci-dessus (qui s'étale sur deux pages), on a présenté des **opérateurs** et des **méthodes** qui permettent de manipuler des objets **mutables** de type **set** (et qui ne concernent donc pas les **frozensets**).

Les notations utilisées plus tôt restent les mêmes et, de plus, **el** est un élément qui est ajouté au ou retiré du **set appelant** les méthodes respectives **add()** ou **remove()** et **discard()**.

B Opérations avec des objets mutables de type set (suite)

A part la méthode **pop()**, toutes les autres méthodes retournent **None** (si leur exécution ne lance pas d'exception).

Les méthodes aux lignes **1**, **3**, **5** et **7** peuvent avoir comme arguments des objets **set** ou des **itérables** (à conditions que leurs éléments soient **immutables**).

```
ens = {1,4,3,2,1}
print(ens)                                  #affiche {1, 2, 3, 4}
print(ens.update({11,1,3}))                 #affiche None
print(ens)                                  #affiche {1, 2, 3, 4, 11}

ens |= set([11,1,3,-15,23,-15])
print(ens)                                  #affiche {1, 2, 3, 4, 11, -15, 23}

ens &= {3,1,4} & {0,1,3,-15,0}
print(ens)                                  #affiche {1, 3}

ens |= {-1,10} | {10,-4}
print(ens)                                  #affiche {1, 3, 10, -4, -1}
```

B Opérations avec des objets mutables de type set – exemples

```
ens = {1, 3, 10, -4, -1}

ens -= {-1,0,1} | {-5,-4,-3}
print(ens)                                          #affiche {3, 10}

ens ^= {0, 10, 20}
print(ens)                                          #affiche {0, 3, 20}

#ci-dessous => TypeError: unhashable type: 'list'
#ens.update([0, [2, 3]])

ens.update(list([3,6,6,9]))
print(ens)                                           #affiche {0, 3, 6, 9, 20}

ens = {1,2,3,4}

#ci-dessous => TypeError: set.add() takes exactly one argument (2 given)
#ens.add(0, 10)

#ci-dessous => TypeError: unhashable type: 'list'
#ens.add([0,10])

print(ens.add('cinq'))                               #affiche None
print(ens)                                           #affiche {1, 2, 3, 4, 'cinq'}
```

B Opérations avec des objets mutables de type set – exemples (suite)

```
ens = {1, 2, 3, 4, 'cinq'}

print(ens.remove(1))                                    #affiche None
print(ens)                                              #affiche {2, 3, 4, 'cinq'}

#ci-dessous => KeyError: {2, 3}
#ens.remove({2,3})

print(ens.discard({2,3}))                               #affiche None
print(ens)                                              #affiche {2, 3, 4, 'cinq'}

#ci-dessous => KeyError: 5
#ens.remove(5)

print(ens.discard(5))                                   #affiche None
print(ens)                                              #affiche {2, 3, 4, 'cinq'}

ens.discard('cinq')
print(ens)                                              #affiche {2, 3, 4}

print(ens.pop())                                        #affiche 2
print(ens)                                              #affiche {3, 4}

print(ens.clear())                                      #affiche None
print(ens)                                              #affiche set()
```

C. Type mapping – C1 Type (classe) dict

Un objet de type **mapping** est un objet **mutable** qui **mappe**, i.e. **met en correspondance**, des **clés** (*keys* en anglais) et des **valeurs** (*values* en anglais).

La **classe dict** est la **seule** classe qui implémente le type **mapping**.

Un objet de type **dict** est un objet **mutable** dont les **éléments** sont des paires (des couples) **clés-valeurs** et ces éléments ne sont **pas ordonnés**.

Les **clés** d'un **dictionnaire** doivent être des objets **immutables** (donc hachables) et ces **clés** doivent être **uniques**.

Par conséquent, par exemple, la **clé** d'un **dictionnaire** :

- ne peut pas être une **liste** ou un **dictionnaire** ou un **set** ;
- peut être un **nombre** (normalement un **int**), une **string**, un **tuple** ou un **frozenset**.

Les **valeurs** d'un **dictionnaire** peuvent être des **objets quelconques** et ces **valeurs** ne doivent **pas** être **uniques** (donc les **doublons** sont admis).

C1 Création d'un objet de type dict

Un objet de type **dict** peut être créé en utilisant :

- une paire d'**accolades** (ouvrante – fermante) :
  - vide (afin de créer un **dictionnaire vide** – *empty dictionary* en anglais) ;
  - qui entourent un **seul** élément de la forme **key : value** (où le séparateur deux-points **:** fait partie de la syntaxe) ou une **suite** de tels éléments séparés par des **virgules** ;
- le constructeur de la **classe dict** :
  - sans argument (pour créer un **dictionnaire vide**) ;
  - avec un nombre variable d'**arguments (**optionnels**) nommés** de la forme **key = value** ;
  - avec comme (premier) seul **argument** (optionnel) **positionnel** :
    - soit un **dictionnaire** ;
    - soit un **itérable** d'objets itérables avec exactement deux éléments – par exemple une liste de couples de la forme **(key, value)** ; si une clé apparaît plusieurs fois, la dernière valeur associée à cette clé sera retenue par le nouveau **dictionnaire** créé ;
  - avec un premier argument **positionnel** (un dictionnaire ou un itérable comme ci-dessus) et des **arguments nommés** supplémentaires de la forme **key = value** ; si une clé apparaît dans le premier argument et aussi comme argument nommé, la valeur de l'argument nommé sera retenue.

C1 Création d'un objet de type dict (suite)

En outre, un objet de type **dict** peut être aussi créé en utilisant :

- la méthode **de classe fromkeys(keys, value)** de la classe **dict** qui crée un nouveau dictionnaire dont :
  - les clés sont précisées par le premier argument **keys** qui doit être un **itérable** ;
  - la valeur commune à toutes les clés est :
    - soit la valeur explicite correspondant au deuxième argument (optionnel) **value** ;
    - soit **None** si le deuxième argument est absent ;
- un **dictionnaire en compréhension** ;
- des **fonctions** et des **méthodes** qui retournent comme résultats des **dictionnaires**.

Il convient de mentionner que :

- les **éléments** d'un dictionnaire ne sont **pas ordonnés** dans le sens qu'ils ne sont **pas accessibles** en fonction de leur **position** (mais en fonction de leur **clé**) ;
- cependant, à partir de la version 3.7, l'ordre d'insertion des éléments est garanti et préservé par la suite ;
- cet ordre n'est pas modifié si la **valeur** associée à une **clé** est mise à jour ;
- en outre, à partir de la version 3.8, les dictionnaires sont **réversibles**.

Dans d'autres langages les **dictionnaires** sont appelés aussi **tableaux associatifs** ou **tables de hachage**.

C1 Création d'un objet de type dict – exemples

```
dic = {}
print(type(dic))                    #affiche <class 'dict'>
print(len(dic))                     #affiche 0
print(dic)                          #affiche {}

dic = {1:'un', 2:'deux', 3:'un'}
print(type(dic))                     #affiche <class 'dict'>
print(len(dic))                      #affiche 3
print(dic)                           #affiche {1: 'un', 2: 'deux', 3: 'un'}

#ci-dessous => SyntaxError: expression cannot contain assignment, perhaps you meant "=="?
#dic_ko = dict(1='un', 2='deux', 3='un')

#ci-dessous => SyntaxError: keyword argument repeated: un
#dic_ko = dict(un=1, deux=2, un=3)

#ci-dessous => NameError: name 'zero' is not defined
#dic_ko = dict(zero, un=1, deux=2, trois=3)
```

C1 Création d'un objet de type dict – exemples (suite)

```
dic = dict(un=1, deux=2)
print(dic)                                  #affiche {'un': 1, 'deux': 2}

dic = dict([(1,'un'), (2,'deux'), (1,'trois')])
print(dic)                                  #affiche {1: 'trois', 2: 'deux'}

#ci-dessous => NameError: name 'un' is not defined
#dic_ko = dict([(un,1), (deux,2)], deux=-2)

dic = dict([('un',1), ('deux',2)], deux=-2)
print(dic)                                    #affiche {'un': 1, 'deux': -2}

dic = dict.fromkeys([1,2,3], 0)
print(dic)                                    #affiche {1: 0, 2: 0, 3: 0}

dic = dict.fromkeys([1,2], [11,12])
print(dic)                                    #affiche {1: [11, 12], 2: [11, 12]}
```

C1 Opérations avec des dictionnaires

No	Opérateur /Fonction/Méthode	Effet / Résultat retourné
1	**len(dic)**	le nombre d'éléments de **dic**
2	**dic[key]**	l'élément de **dic** ayant la clé **key** si cette clé est dans **dic** ou lance une exception **KeyError** si **key** n'est pas dans **dic**
3	**dic[key] = val**	affecte **val** à la clé **key**
4	**del dic[key]**	supprime l'élément **dic[key]** de **dic** ou lance une exception **KeyError** si **key** n'est pas dans **dic**
5	**key in dic**	**True** si **dic** a la clé **key** ou **False** autrement
6	**key not in dic**	**False** si **dic** a la clé **key** ou **True** autrement
7	**dic.get(key)**	la valeur associée à **key** si la clé **key** est dans **dic** ou **None** si **key** n'est pas dans **dic**
8	**dic.pop(key)**	supprime **key** et retourne la valeur associée si **key** est dans **dic** ou lance une exception **KeyError** si **key** n'est pas dans **dic**
9	**dic.popitem()**	supprime et retourne une paire **(clé, valeur)** de **dic** ou lance une exception **KeyError** si **dic** est vide

C1 Opérations avec des dictionnaires (suite)

No	Opérateur/Fonction/Méthode	Effet/Résultat retourné
10	**dic.items()**	nouvelle vue des éléments **(clés-valeurs)** de **dic**
11	**dic.keys()**	nouvelle vue des **clés** de **dic**
12	**list(dic)**	une **liste** avec toutes les **clés** de **dic**
13	**iter(dic)**	un itérateur pour parcourir les **clés** de **dict**
14	**reversed(dic)**	un itérateur inversé pour parcourir les **clés** de **dic** (à partir de 3.8)
15	**dic.values()**	nouvelle vue des **valeurs** de **dic**
16	**dic.clear()**	supprime tous les éléments du **dic**
17	**dic.copy()**	une copie superficielle du **dic**
18	**dic.setdefault(key, val)**	retourne la valeur associée à **key** si **key** est dans **dict** ou, autrement, insère la paire **(key, val)** dans **dic** et retourne **val**
19	**dic1.update(dic2)**	met à jour **dic1** avec les paires **(clés, valeurs)** de **dic2** en récrivant (*overwriting* en anglais) les éventuelles **clés** existantes dans **dic1**

C1 Opérations avec des dictionnaires (suite)

No	Opérateur/Fonction/Méthode	Effet/Résultat retourné
20	**dic1 | dic2**	un nouveau dictionnaire obtenu en fusionnant **dic1** et **dic2**
21	**dic1 |= dic2**	met à jour **dic1** avec les **clés** et les **valeurs** de **dic2**

Dans le tableau précédent (qui s'étale sur plusieurs pages), on a présenté des **opérateurs**, des **fonctions** et des **méthodes** qui permettent de manipuler des **dictionnaires**.

On a utilisé les notations suivantes :

- **dic**, **dic1** et **dic2** sont des objets de type **dict** ;
- **key** est une clé d'un **dictionnaire** ;
- **val** est une valeur (associée ou affectée à une **clé**) d'un **dictionnaire**.

On donne par la suite des précisions concernant certaines lignes du tableau précédent.

Il convient de mentionner que les méthodes **clear()** et **update()** de la classe **dict** retournent **None**.

La ligne **3** => **dic[key] = val** :

- si la clé **key** n'est pas dans **dic**, la paire **key-val** est ajoutée au **dic**.

C1 Opérations avec des dictionnaires (suite)

La ligne **7** => **dic.get(key)** :

- la méthode peut être appelée aussi avec un deuxième argument **dic.get(key, val)** qui précise une valeur **val** qui est retournée si la clé **key** n'est pas dans **dic**.

La ligne **8** => **dic.pop(key)** :

- la méthode peut être appelée aussi avec un deuxième argument **dic.pop(key, val)** qui précise une valeur **val** qui est retournée si la clé **key** n'est pas dans **dic** (et il n'y a plus d'erreur lancée).

La ligne **9** => **dic.popitem()** :

- les paires **(clé, valeur)** sont retournées en ordre **LIFO** (**L**ast **I**n, **F**irst **O**ut) qui est garanti (à partir de la version 3.7).

Les lignes **10**, **11** et **15** => **dic.items()**, **dic.keys()** et **dic.values()** :

- ces méthodes retournent de nouvelles **vues** dynamiques (qui sont des objets qui changent quand le **dictionnaire** change et qui peuvent être itérés afin de parcourir) des **éléments**, des **clés** et, respectivement, des **valeurs** du dictionnaire appelant.

C1 Opérations avec des dictionnaires (suite)

La ligne **18** => **dic.setdefault(key, val)** :

- la méthode **setdefault()** peut être appelée aussi avec comme seul argument **key** et, dans ce cas, si **dic** n'a pas la clé **key**, un nouvel élément **(key, None)** est ajouté à **dic** et la constante **None** est retournée.

La ligne **20** => **dic1 | dic2** :

- dans le nouveau **dictionnaire** obtenu grâce à cet **opérateur** (ajouté dans la version 3.9), la **valeur** retenue pour une **clé** partagée par **dic1** et **dic2** est celle de **dic2**.

La ligne **21** => **dic1 |= dic2** :

- dans **dic1** modifié grâce à cet **opérateur** (ajouté dans la version 3.9), la **valeur** retenue pour une **clé** partagée par **dic1** et **dic2** est celle de **dic2** ;
- **dic2** peut être un **dictionnaire** ou un **itérable** de paires **clés-valeurs**.

Deux **dictionnaires** peuvent être comparés avec l'**opérateur ==** qui retourne **True** si et seulement si les deux opérandes ont les **mêmes** paires **(clés, valeurs)** indépendamment de l'ordre.

En revanche, pour les **dictionnaires**, les **opérateurs <, <=, >** et **>=** lancent une exception **TypeError**.

C1 Opérations avec des dictionnaires – exemples

```
dic = {'un':1, 'deux':2, 'un':3}
print(dic)                          #affiche {'un': 3, 'deux': 2}

#ci-dessous => KeyError: 2
#print(dic[2])

print(dic['deux'])                  #affiche 2

dic['deux'] = 'two'
print(dic)                          #affiche {'un': 3, 'deux': 'two'}

dic['trois'] = 3
print(dic)                          #affiche {'un': 3, 'deux': 'two', 'trois': 3}

del dic['deux']
print(dic)                          #affiche {'un': 3, 'trois': 3}

#ci-dessous => KeyError: 3
#del dic[3]

print('trois' in dic)               #affiche True
print(3 not in dic)                 #affiche True

print(dic.get('trois'))             #affiche 3
print(dic.get(3))                   #affiche None
```

C1 Opérations avec des dictionnaires – exemples (suite)

```
dic = {'red':'rouge', 'green':'vert', 'blue':'bleu'}

print(dic.pop('green'))                 #affiche vert
print(dic)                              #affiche {'red': 'rouge', 'blue': 'bleu'}

#ci-dessous => KeyError: 'yellow'
#dic.pop('yellow')

print(dic.popitem())                    #affiche ('blue', 'bleu')

print(dic.clear())                      #affiche None
print(dic)                              #affiche {}

#ci-dessous => KeyError: 'popitem(): dictionary is empty'
#dic.popitem()

dic = {'red':'rouge', 'green':'vert'}

#ci-dessous => affiche dict_items([('red', 'rouge'), ('green', 'vert')])
print(dic.items())

print(list(dic.items()))                #affiche [('red', 'rouge'), ('green', 'vert')]

print(list(dic.keys()))                 #affiche ['red', 'green']

print(list(dic))                        #affiche ['red', 'green']

print(list(iter(dic)))                  #affiche ['red', 'green']

print(list(reversed(dic)))              #affiche [green', 'red']

print(list(dic.values()))               #affiche ['rouge', 'vert']
```

C1 Opérations avec des dictionnaires – exemples (suite)

```
dic = {'red':'rouge', 'green':'vert'}

dic_bis = dic.copy()
print(dic_bis == dic)                    #affiche True
print(id(dic_bis) == id(dic))            #affiche False

print(dic.clear())                       #affiche None

print(list(dic.values()))                #affiche []

dic = {'red':'rouge', 'green':'vert'}
print(dic.setdefault('green', 'grün'))   #affiche vert
print(dic)                               #affiche {'red': 'rouge', 'green': 'vert'}

print(dic.setdefault('blue', 'bleu'))    #affiche bleu
print(dic)              #affiche {'red': 'rouge', 'green': 'vert', 'blue': 'bleu'}

del dic['red']

print(dic.setdefault('black'))           #affiche None
print(dic)              #affiche {'green': 'vert', 'blue': 'bleu', 'black': None}
```

C1 Opérations avec des dictionnaires – exemples (suite)

```
dic1 = {'red':'rouge', 'green':'vert'}
dic2 = {'green':'grün', 'blue':'blau'}

print(dic1.update(dic2))#affiche None
print(dic1)             #affiche {'red': 'rouge', 'green': 'grün', 'blue': 'blau'}

dic1 = {'red':'rouge', 'green':'vert'}

dic3 = dic1 | dic2
print(dic3)             #affiche {'red': 'rouge', 'green': 'grün', 'blue': 'blau'}

print(dic1)             #affiche {'red':'rouge', 'green':'vert'}

dic1 |= dic2
print(dic1)             #affiche {'red': 'rouge', 'green': 'grün', 'blue': 'blau'}

dic1 = {1:'un', 2:'deux'}
dic2 = {3:'trois'}

#ci-dessous => TypeError: '<' not supported between instances of 'dict' and 'dict'
#print(dic1 < dic2)
```

Chapitre 9

Types natifs containers – Troisième partie

Python

A4 Type (classe) str

Le type **str** est un des **types natifs séquentiels**.

Un **objet** de type **str** (ou, simplement, une **string**) est :

- une **séquence immutable** de caractères (donc une **chaîne de caractères** qui ne peut plus être modifiée après sa création) ;
- une instance de la **classe (prédéfinie) native str**.

Il convient de préciser que Python ne prévoit pas de type dédié/spécial pour des **caractères** individuels/isolés (mais seulement pour les **strings**, y compris la chaîne vide et les chaînes avec un seul caractère).

Cependant, selon le standard informatique (ou la norme) appelé(e) **Unicode**, à chaque **caractère** est associé :

- soit un **code Unicode simple** qui est un entier compris entre 0 et 65535 (codage UTF-16 normalement, stockage sur 2 octets correspondant au Basic Multilingual Plan) pour les premiers caractères de la norme ;
- soit un **point de code** (codage UTF-32 normalement, stockage sur 4 octets et utilisation des plans supplémentaires) pour les caractères suivants (dits aussi supplémentaires) de la norme.

A4 Création d'un objet de type str

Un **objet** de type **str** peut être créé en utilisant :

- une **paire** de **simples quottes** ou de **doubles quottes** ou de **triples quottes** :
  - vide (afin de créer une **string vide** – *empty string* en anglais) ;
  - qui entourent un **seul caractère** ou une **chaîne de caractères** ;
- le **constructeur** de la classe **str** :
  - sans argument (afin de créer une **string vide**) ;
  - avec comme seul argument un objet quelconque (qui est converti en **string** et) qui peut être, en particulier, une **chaîne de caractères** ;
  - avec deux ou trois arguments pour le cas où, normalement, le type du premier argument est le **type séquentiel bytes** , **bytearray** ou similaire ;
- des **fonctions** et des **méthodes** qui retournent comme résultats des **strings**.

L'appel (de la **fonction** native ou) du **constructeur str(obj)** avec comme seul argument l'objet **obj** retourne la valeur **string** obtenue :

- soit par l'appel de la **méthode** magique (*dunder*) **obj.__str__()**, si une telle méthode existe dans la classe de l'objet **obj** ;
- soit par l'appel de la **fonction** native **repr(obj)**, autrement.

A4 Création d'un objet de type str (suite)

Plus précisément :

- une classe peut prévoir des **méthodes** magiques **__str__()** ou/et **__repr__()** afin de proposer une/des version(s) **ad hoc** pour représenter une instance de cette classe comme **string** (affichable) ;
- la **fonction** native **repr(obj)** retourne :
  - soit la valeur obtenue par l'appel de la **méthode** magique **obj.__repr__()**, si une telle méthode existe dans la classe de l'objet **obj** ;
  - soit une **string** contenant une représentation affichable de l'objet **obj** (qui dépend de son type), autrement.

Si la **fonction** native **ord()** est appelée avec **un** argument **string** contenant un seul **caractère**, alors elle retourne le code Unicode de ce **caractère** ; appelée autrement, elle lance une exception **TypeError**.

Si la **fonction** native **chr()** est appelée avec **un** argument **int** qui est un code Unicode, alors elle retourne une **string** qui représente **le** caractère correspondant à ce code Unicode ; appelée autrement, elle lance une exception de type **ValueError** (si le seul argument est un entier dont la valeur est inappropriée) ou **TypeError** (dans les autres cas).

A4 Création d'un objet de type str (suite)

Parmi les caractères qui apparaissent dans une **string littérale** qui est délimitée par des **simples** ou **doubles** ou **triples quottes** :

- les quottes qui ne sont **pas** les **délimiteurs** de la **string** peuvent apparaître et sont affichées telles quelles ;
- les quottes qui **délimitent** la **string** peuvent apparaître et seront affichées telles quelles si on les **déspécialise** en les faisant précéder par le caractère **antislash** \ ;
- certains caractères dits **de contrôle** forment des séquences **d'échappement** qui ne sont pas affichées telles quelles car elles ont des significations spéciales pour la mise en forme (le formatage) de la **chaîne de caractères**, comme **\n** qui précise un saut de ligne ou **\t** qui précise une tabulation ;
- le caractère **antislash** doit être **doublé** (déspécialisé par lui-même) \\ afin de l'afficher comme un seul caractère antislash \ ;
- certains caractères (parmi les premiers 65536 et, en particulier, ceux qui ne se retrouvent pas sur les touches du clavier) peuvent être précisés par leur **code Unicode** (exprimé avec quatre chiffres hexadécimaux et) précédé par les caractères **\u**, par exemple **'\u0030'** pour le chiffre **'0'** de code Unicode décimal **48**, **'\u0041'** pour la lettre majuscule **'A'** de code Unicode décimal **65** ou **'\u0061'** pour la lettre minuscule **'a'** de code Unicode décimal **97**.

A4 Création d'un objet de type str (suite)

Il convient de préciser qu'une **string** définie avec les **triples quottes** :

- peut s'étaler sur **plusieurs** lignes physiques et contient toutes les "espaces blanches" (*whitespaces* en anglais) qui apparaissent entre les quottes, y compris les tabulations et les sauts de ligne ;
- peut être utilisée comme un **commentaire** sur plusieurs lignes physiques ;
- peut être placée au **début** du corps d'une **fonction** cas où cette **string** est un **Docstring**, i.e. une chaîne de documentation (qui peut être consulté(e) grâce à l'attribut magique (*dunder*) **__doc__** précédé/préfixé par/avec le nom de la **fonction** suivi par le séparateur point **.**).

A son tour, une **string littérale** délimitée par de **simples** ou **doubles quottes** peut être écrite sur **plusieurs** lignes physiques :

- si chaque ligne (sauf la dernière) finit par le caractère **antislash** **** (qui joue le rôle de caractère de continuation et qui n'est pas inclus dans la **string littérale**) ;
- les sauts de ligne évoqués ci-dessus ne sont pas inclus dans la **string littérale**.

Deux **strings littérales** qui font partie d'une même expression et qui sont séparées par une ou plusieurs espaces (ou tabulations) sont simplement **concaténées** (ou juxtaposées).

A4 Création d'un objet de type str (suite)

Une **string brute** (*raw string* en anglais), appelée aussi **r-string** :

- est une **string littérale** délimitée par des quottes précédées par la lettre **r** (ou **R**) ;
- la signification spéciale des **séquences d'échappement** qui apparaissent entre les quottes est ignorée et elles sont affichées telles quelles.

Une **string formatée** (*formatted string* en anglais), appelée aussi **f-string** :

- est une **string littérale** délimitée par des quottes précédées par la lettre **f** (ou **F**) ;
- sa **valeur** n'est pas constante et elle est évaluée à l'exécution (au Runtime) ;
- prend en compte les significations spéciales des **séquences d'échappement** ;
- ne peut **pas** contenir de **commentaire** ;
- les éventuelles **expressions** entre des **accolades { }** correspondent à des **champs de remplacement** (*replacement fields* en anglais) et elles sont **évaluées** comme du code Python et insérées ensuite dans la **string** finale (ce qui évite de concaténer des strings en convertissant les valeurs non-string) ;
- pour faire afficher une **accolade** dans la **string finale**, il faut la doubler et l'écrire **{{** ou **}}** ;
- afin d'afficher dans la **string finale** à la fois le texte entre les **accolades** et sa valeur, on peut ajouter un signe **=** après l'expression respective ;
- un **champ de remplacement** peut contenir aussi un **spécificateur de format** (*format specifier* en anglais) ainsi que des **champs de remplacement imbriqués**.

A4 Création d'un objet de type str – exemples

```
s = 'Bonjour "tout" le monde !'
print(type(s))                          #affiche <class 'str'>
print(s)                                #affiche Bonjour "tout" le monde !

s = "Bonjour 'tout' le monde !"
print(type(s))                          #affiche <class 'str'>
print(s)                                #affiche Bonjour 'tout' le monde !

#ci-dessous => SyntaxError: invalid syntax
#s = 'Bonjour 'tout' le monde !'

#les deux instructions ci-dessous => SyntaxError: EOL while scanning string literal
#s = 'Bonjour
#tout le monde !'

s1 = 'Bonjour\
tout le monde !'
print(s1)                               #affiche Bonjourtout le monde !

s2 = """Bonjour
tout le monde !"""
print(type(s2))                         #affiche <class 'str'>
print(s2)                               #affiche Bonjour
                                        #        tout le monde !
print('Bonjour\ntout le\tmonde !')      #affiche Bonjour
                                        #        tout le    monde !
print(r'Bonjour\ntout le\tmonde !')     #affiche Bonjour\ntout le\tmonde !
```

A4 Création d'un objet de type str – exemples (suite)

```
s = str()
print(type(s))                    #affiche <class 'str'>
print(len(s))                     #affiche 0

s = str('Bonjour !')
print(type(s))                    #affiche <class 'str'>
print(s)                          #affiche Bonjour !

print(type(str(55)))              #affiche <class 'str'>

from datetime import date
aoujourd_hui = date.today()
print(type(aoujourd_hui))          #affiche <class 'datetime.date'>

print(aoujourd_hui)                #par exemple, affiche 2022-10-26
print(str(aoujourd_hui))           #par exemple, affiche 2022-10-26
print(aoujourd_hui.__str__())      #par exemple, affiche 2022-10-26

print(repr(aoujourd_hui))          #par exemple, affiche datetime.date(2022, 10, 26)
print(aoujourd_hui.__repr__())     #par exemple, affiche datetime.date(2022, 10, 26)

x, y, z = 5, ['a','b'], {1:11}
#Les deux instructions non commentées ci-dessous affichent :
#Le premier résultat 5, le second y=['a', 'b'] et le dernier {1: 11} !
print(f'Le premier résultat {x}, le second {y=} \
et le dernier {z} !')

s = 'Moi,'  'Pierre le Grand'
print(s)                           #affiche Moi,Pierre le Grand
```

A4 Opérations avec des strings

Une **string** peut être vue comme un **container** (une **séquence**) **immutable** dont les éléments **ordonnés** sont des **caractères** (qui, isolés, sont en fait des **strings** de **longueur 1**).

Donc, toutes les **opérations communes** qui ont été présentées pour les **séquences immutables** (ainsi que les **remarques** associées) sont **valables** aussi pour les objets **string** (voir le tableau suivant qui reprend le tableau de la page 10 du Chapitre 7).

Par la suite, on utilise les notations suivantes :

- **s**, **s1** et **s2** sont des **strings** ;
- **sub** est (normalement) une **(sub)string** (ou une sous-chaîne de caractères) par rapport à la **string s** ;
- **i, j** et **k** sont des **indices** entiers ;
- **n** est un **nombre entier**.

Les **comparaisons** des :

- **caractères** individuels se font selon leurs codes Unicode (qui respectent l'ordre alphabétique usuel) ;
- **strings** se font **caractère par caractère**, de gauche vers la droite et en fonction des codes Unicode des **caractères**.

A4 Opérations avec des strings (suite)

No	Opérateur / Fonction / Méthode	Résultat retourné
1	**s[i]**	le caractère d'indice i de **s**
2	**s[i:j]**	la tranche (la substring) de **i** à **j** de **s**
3	**s[i:j:k]**	la tranche de **i** à **j** avec le pas **k** de **s**
4	**s1 + s2**	la string obtenue par la concaténation des strings **s1** et **s2**
5	**s * n** ou **n * s**	la string obtenue par la concaténation de **s** avec elle-même **n** fois
6	**sub in s**	**True** si une substring de **s** est égale à **sub** ou **False** autrement
7	**sub not in s**	**False** si une substring de **s** est égale à **sub** ou **True** autrement
8	**s.index(sub)**	le premier indice de la première occurrence de **sub** dans **s**
9	**s.count(sub)**	le nombre total d'occurrences de **sub** dans **s**
10	**len(s)**	la longueur (la taille ou le nombre de caractères) de **s**
11	**min(s)**	le plus petit élément (caractère) de **s**
12	**max(s)**	le plus grand élément (caractère) de **s**

A4 Opérations avec des strings (suite)

- la méthode **s.index(sub, *start*, *end*)** :
  - soit retourne le premier indice (le plus à gauche) de la première occurrence de **sub** dans la **string** appelante (éventuellement dans la tranche **range(*start*, *end*)** de la **string** appelante si ***start*** et/ou ***end*** sont présents dans l'appel), si au moins une telle occurrence existe ;
  - soit lance une exception **ValueError**, s'il n'y a aucune telle occurrence ;
- la méthode **s.count(sub, *start*, *end*)** retourne le nombre d'occurrences de la **string** argument **sub** qui ne se chevauchent pas :
  - soit dans toute la **string** appelante, si **sub** est le seul argument de l'appel ;
  - soit dans la tranche **range(*start*, *end*)** de la **string** appelante, si ***start*** et/ou ***end*** sont présents dans l'appel ;
- la méthode **s.len()** retourne le nombre de caractères dans la **string** appelante tout en tenant compte qu'une séquence d'échappement peut être considérée souvent comme un seul **caractère** ;
- un caractère **lettre** (minuscule ou majuscule ou capitalisée) doit être compris dans un sens large (i.e. comme *cased character* en anglais).

A4 Opérations avec des strings (suite)

Il convient de mentionner que pour :

- l'indexage (*indexing*) : si la valeur entière de l'indice est trop petite, i.e. **<-len(s)**, ou trop grande, i.e. **>len(s)-1**, une exception **IndexError** est lancée et un message "*string index out of range*" est affiché ;
- le découpage (*slicing*) : si le valeur de début est trop petite et/ou celle de fin trop grande, il n'y a pas d'exception/erreur et le découpage commence au début et/ou finit à la fin de la **string**.

En outre :

- il existe aussi le **module** standard **string** qui fournit (surtout) des constantes supplémentaires et des possibilités de formatage personnalisé des **strings** ;
- les **strings littérales** peuvent être aussi formatées grâce à la **fonction** native **format()** ou à la **méthode format()** de la classe **str** (en utilisant soit des champs de remplacement placés entre des **accolades** soit l'opérateur de formatage **%**).

Après quelques exemples d'opérations avec des **strings**, on présente de manière succincte une (petite) partie des **méthodes** disponibles dans la classe **str** et qui facilitent le travail avec des objets **chaînes de caractères**.

Les paramètres ***facultatifs*** dans les en-têtes de ces méthodes sont notés en ***italique*** (et **gras**).

A4 Opérations avec des strings – exemples

```
s ='0123456789567 89'
print(s[3])                                 #affiche 3
print(s[-3])                                #affiche 7
print(s[-15])                               #affiche 0

#ci-dessous => IndexError: string index out of range
#print(s[20])

print(s[-20:3])                             #affiche 012
print(s[3:20:3])                            #affiche 3697

print('567' in s)                           #affiche True

print(s.index('567'))                       #affiche 5

#ci-dessous => TypeError: count() takes at least 1 argument (0 given)
#print(s.count())

print(s.count('567'))                       #affiche 2

print(len(s))                               #affiche 15

s = 'Tu es\t mon meill\reur ami.'
print(s)                                    #affiche eur ami.on meill
print(len(s))                               #affiche 25

print(min('Pierre_le_Grand'))               #affiche G

print(max('Mon ami Pierrot'))               #affiche t
```

A4 Méthodes d'instance de la classe str

- **s.upper()**

retourne une copie de la **string** appelante dans laquelle toutes les **lettres** sont converties en **majuscules**.

- **s.lower()**

retourne une copie de la **string** appelante dans laquelle toutes les **lettres** sont converties en **minuscules**.

- **s.capitalize()**

retourne une copie de la **string** appelante dans laquelle le premier caractère est capitalisé et les autres **lettres** sont converties en **minuscules**.

- **s.split(*sep=None*, *maxsplit=-1*)**

retourne une **liste** de **(sub)strings** obtenues en découpant la **string** appelante selon :

  - les espaces blanches (comme les espaces, les tabulations ou les sauts de lignes), s'il n'y a pas d'argument ***sep*** ou s'il a la valeur **None** (et ces espaces ne font pas partie des **(sub)strings**) ;
  - l'argument ***sep*** de type **string**, s'il est présent dans l'appel (et le séparateur ne fait pas partie des **(sub)strings**).

En plus, les découpages commencent à gauche et s'arrêtent après :

  - tous les découpages possibles, si l'argument ***maxsplit*** est absent ou négatif ;
  - au maximum ***maxsplit*** découpages (ce qui donne au maximum (***maxsplit***+1) éléments) si l'argument ***maxsplit*** est présent dans l'appel (et il est non négatif).

A4 Méthodes d'instance de la classe str (suite)

- **s.rsplit(*sep=None*, *maxsplit=-1*)**

travaille comme la méthode **split()**, mais le découpage commence à droite.

- **s.splitlines(*keepends*)**

retourne une liste avec les **lignes** contenues dans la **string** appelante et découpées en fonctions des limites de lignes comme **\n** pour le saut de ligne (*line feed* en anglais) ou **\r** pour le retour de chariot (*carriage return* en anglais) et :

  - sans les limites de lignes, si l'appel est fait sans argument ;
  - avec les limites de lignes, si l'appel est fait avec **True** comme valeur de l'argument ***keepends***.

- **s.join(*iterable*)**

travaille de manière opposée par rapport à la méthode **split()** et :

  - soit retourne une nouvelle **string** obtenue en concaténant les **strings** de l'argument ***iterable*** séparées par la **string** appelante ;
  - soit lance une exception **TypeError**, s'il y a au moins une valeur **non string** dans l'argument ***iterable***.

A4 Méthodes d'instance de la classe str (suite)

- **s.find(sub, *start*, *end*)**

recherche dans toute la **string** appelante (ou seulement dans la tranche **range(*start*, *end*)**, si ***start*** et/ou ***end*** sont présents dans l'appel) la première occurrence (à partir de gauche) de l'argument **sub** et retourne :

  - soit la valeur du **premier index** (le plus à gauche) de cette occurrence, si au moins une telle occurrence existe ;
  - soit **-1**, si **sub** n'a pas été trouvée.

- **s.rfind(sub, *start*, *end*)**

travaille comme la méthode **find()** sauf que la recherche se fait de droite vers la gauche.

- **s.rindex(sub)**

travaille comme la méthode **index()** sauf que la recherche se fait de droite vers la gauche.

- **s.replace(old, new, *count*)**

retourne une copie de la **string** appelante dans laquelle toutes les occurrences de la **string** premier argument **old** sont remplacées par la **string** deuxième argument **new**.

En plus, si le troisième argument ***count*** est présent dans l'appel, seulement les ***count*** premières occurrences sont remplacées.

A4 Méthodes d'instance de la classe str (suite)

- **s.strip(*chars*)**

retourne une copie de la **string** appelante dans laquelle on supprime :

- toutes les éventuelles espaces blanches situées au début ou à la fin de la **string** appelante (mais pas à l'intérieur), si l'appel se fait sans argument ;
- toute combinaison de caractères apparaissant dans la **string** argument ***chars*** et qui est située au début ou à la fin de la **string** appelante, si l'appel se fait avec un argument.

- **s.lstrip()**

travaille comme la méthode **strip()** sauf que seulement les caractères du début sont supprimés.

- **s.rstrip()**

travaille comme la méthode **strip()** sauf que seulement les caractères de la fin sont supprimés.

- **s.startswith(prefix, *start*, *end*)**

retourne :

- **True** si la **string** appelante commence avec **prefix** (qui peut être aussi un tuple de préfixes) ou si la tranche **range(*start*, *end*)** de la **string** appelante commence ainsi ;
- **False** autrement.

A4 Méthodes d'instance de la classe str (suite)

- **s.endswith(suffix, *start*, *end*)**

retourne :

  - **True** si la **string** appelante finit par **suffix** (qui peut être aussi un tuple de suffixes) ou si la tranche **range(*start*, *end*)** de la **string** appelante finit ainsi ;
  - **False** autrement.

- **s.removeprefix(prefix)**

retourne (à partir de la version 3.9 de Python) :

  - une nouvelle **string** **s[len(prefix) :]**, si **s** commence par la **string** argument **prefix** ;
  - la **string** appelante **s**, autrement.

- **s.removesuffix(suffix)**

retourne (à partir de la version 3.9 de Python) :

  - une nouvelle **string** **s[: -len(suffix)]**, si **s** finit par la **string** argument **suffix** ;
  - la **string** appelante **s**, autrement.

- **s.swapcase()**

retourne une copie de la **string** appelante dans laquelle toutes les **minuscules** sont converties en **majuscules** et **vice versa**.

A4 Méthodes d'instance de la classe str – exemples

```
s = 'qu\'il Soit une Fois un Roi'
print(s.upper())              #affiche QU'IL SOIT UNE FOIS UN ROI
print(s.lower())              #affiche qu'il soit une fois un roi
print(s.capitalize())         #affiche Qu'il soit une fois un roi
print(s.swapcase())           #affiche QU'IL sOIT UNE fOIS UN rOI
li1 = s.split()
print(li1)                    #affiche ["qu'il", 'Soit', 'une', 'Fois', 'un', 'Roi']
li2 = s.split('oi')
print(li2)                    #affiche ["qu'il S", 't une F', 's un R', '']
li3 = s.split('oi', 1)
print(li3)                    #affiche ["qu'il S", 't une Fois un Roi']
li4 = s.rsplit('oi', 2)
print(li4)                    #affiche ["qu'il Soit une F", 's un R', '']
s1 = '''qu'il Soit
une Fois
un Roi'''
li5 = s1.splitlines()
print(li5)                    #affiche ["qu'il Soit", 'une Fois', 'un Roi']
print(s1.splitlines(True))    #affiche ["qu'il Soit\n", 'une Fois\n', 'un Roi']
```

A4 Méthodes d'instance de la classe str – exemples (suite)

```
li1 = ["qu'il", 'Soit', 'une', 'Fois', 'un', 'Roi']
s = ' '.join(li1)
print(s)                                #affiche qu'il Soit une Fois un Roi
li2 = ['il', 'est', 3, 'fois', 'roi']
#ci-dessous => TypeError: sequence item 2: expected str instance, int found
#s2 = ' '.join(li2)
print(s.find('oi'))                     #affiche 7
print(s.find('io'))                     #affiche -1
print(s.find('oi', 8))                  #affiche 16
print(s.rfind('oi', -10, -3))           #affiche 16
print(s.rindex('oi'))                   #affiche 24
print(s.replace('oi', '*'))             #affiche qu'il S*t une F*s un R*
print(s.replace('oi', '*', 2))          #affiche qu'il S*t une F*s un Roi
print(s.strip('oqi'))                   #affiche u'il Soit une Fois un R
print(s.lstrip('oqi'))                  #affiche u'il Soit une Fois un Roi
print(s.rstrip('oqi'))                  #affiche qu'il Soit une Fois un R
print(s.startswith('oi', 7))            #affiche True
print(s.endswith('oi', 0, 8))           #affiche False
print(s.endswith('oi', 0, 9))           #affiche True
```

Chapitre 10

Exceptions

Python

Exceptions

Une **exception** est une circonstance (souvent non désirée) qui peut intervenir durant l'**exécution** d'un programme et compromettre son déroulement normal.

Le langage Python implémente un **mécanisme d'exceptions** qui permet un traitement flexible, robuste et efficace des situations anormales/spéciales susceptibles d'apparaître au **Runtime** (dues, par exemple, à une mauvaise introduction de données par l'utilisateur ou à un problème de connexion entre le programme et un périphérique).

Parmi les avantages du **mécanisme d'exception** Python, on peut mentionner :

- une **bonne lisibilité** du code grâce à une séparation claire entre le code ordinaire et les instructions qui gèrent les "anomalies" ;
- la possibilité de **propager** les **exceptions** au niveau des appels successifs des méthodes, afin d'en assurer un traitement optimal.

Le **déclenchement** d'une **exception** est **intercepté** et produit une interruption du cours normal du programme dont l'exécution est aiguillée vers un **gestionnaire d'exceptions** approprié.

Erreurs versus exceptions

En Python, il y a deux catégories principales d'**erreurs** :

- **erreurs de syntaxe** (qui se produisent quand la syntaxe du langage n'est pas respectée et elles doivent être corrigées pour que l'exécution soit ensuite possible) ;
- **erreurs logiques** ou **exceptions** (qui peuvent apparaître **à l'exécution** d'un code qui n'a pas d'erreur de syntaxe et elles peuvent être traitées au **Runtime** sans que le programme s'arrête).

Il y a aussi d'**autres catégories** d'**erreurs** comme "*out of memory error*" ou "*recursion error*" mais, le plus souvent, ces erreurs ne sont pas traitées durant l'exécution et produisent l'arrêt du programme.

Une **classe** d'**exception** (prédéfinie ou non) :

- est associée à une certaine circonstance anormale/spéciale pouvant intervenir au cours du fonctionnement d'un programme (au **Runtime**) ;
- doit contenir (normalement) toutes les informations nécessaires à un traitement approprié de la situation occurrente.

Autrement dit, une **instance** d'une classe d'**exception** (ou, simplement, une **exception**) doit être un "objet diagnostic" qui décrit ce qui s'est produit afin de permettre à un **gestionnaire d'exceptions** de traiter l'anomalie qui a déclenché l'**exception**.

Classes d'exception

Toute **classe** d'**exception** (prédéfinie ou définie par le programmeur) doit hériter (directement ou pas) de la **classe de base BaseException** qui :

- est une classe native (*built-in*) ;
- est la **fille** de la classe **object** ;
- est la racine (*root* en anglais) de la hiérarchie d'**exceptions** ;
- a plusieurs **filles** (à savoir quatre) parmi lesquelles la classe native **Exception** dont certaines de ses classes descendantes natives seront présentées par la suite.

Malgré le grand nombre de **classes** d'**exception** prédéfinies (voir les exemples de **classes** natives ci-dessous), le programmeur peut et, parfois, doit définir ses propres **classes** d'**exception** (qui doivent hériter de la **classe** native **Exception**).

La qualité du traitement d'**exceptions** dépend en grande mesure de la capacité du programmeur d'anticiper les éventuelles situations non désirées pouvant intervenir au **Runtime** et de prévoir des solutions alternatives capables de permettre au programme de continuer à s'exécuter correctement.

Classes natives d'exceptions – exemples

La **classe Exception** est (normalement) la **classe de base** (directe ou pas) pour toutes les **exceptions** prédéfinies qui n'entraînent pas une sortie du système et pour les **exceptions** définies par le programmeur.

On lance ("automatiquement") une **exception** de type :

- **IndexError** quand l'indice d'une séquence est en dehors de la plage de valeurs valides ;
- **KeyError** quand une clé recherchée n'est pas trouvée parmi les clés existantes d'un dictionnaire ;
- **NameError** quand un nom (non qualifié) local ou global n'est pas trouvé ;
- **OverflowError** quand le résultat d'une opération arithmétique est trop grand pour être représenté ;
- **RuntimeError** quand une erreur qui n'a pas une catégorie prédéfinie plus spécifique se produit ;
- **StopIteration** quand la fonction **next()** signale qu'il n'y a plus d'élément produit par un itérateur ;
- **TypeError** quand une opération/fonction est appliquée pour un objet qui n'a pas le type approprié ;
- **ValueError** quand une opération/fonction reçoit un opérande/argument qui a le bon type mais une valeur non valide (et le problème n'est pas de type **IndexError**) ;
- **ZeroDivisionError** quand le deuxième opérande de l'opération de division ou de modulo est zéro ;
- **SystemExit** (fille de la classe **BaseException**) quand la fonction **sys.exit()** est appelée.

Mécanisme d'exceptions

Le **mécanisme d'exceptions** :

- fait intervenir plusieurs mots clés (*keywords*), à savoir : **try**, **raise**, **except**, **as**, **else** et **finally** ;
- implique une structuration spécifique du code qui, assez souvent, se présente comme dans l'instruction structurée **try** ci-dessous.

```
try :
        bloc_try
except (Exception_1_1, …, Exception_1_m) :
        bloc_except_1
…
except (Exception_n_1, … Exception_n_p) :
        bloc_except_n
else :
        bloc_else
finally :
        bloc_finally
```

Mécanisme d'exceptions (suite)

Plus précisément .

- l'en-tête **try** est suivi par le **bloc_try** (indenté) qui contient le code susceptible de produire des **exceptions** ;
- un en-tête **except** prévoit (assez souvent) une **classe** d'**exception** ou un tuple de **classes** d'**exceptions** à traiter dans le **bloc_except** correspondant (qui est indenté) ;
- l'en-tête **else** est suivi(e) par le **bloc_else** (indenté) qui contient du code qui est exécuté seulement si aucune exception n'est lancée ;
- l'en-tête **finally** est suivi(e) par le **bloc_finally** (indenté) qui contient le code qui est toujours exécuté (indépendamment du fait qu'une **exception** a été lancée ou pas).

En outre :

- la clause **try** doit être suivie par au moins une clause **except** ;
- un en-tête **except** peut avoir aussi d'autres formes qui seront mentionnées par la suite ;
- la clause **else** est facultative mais, si elle est présente, elle doit venir juste après les en-têtes **except** ;
- la clause **finally** est facultative mais, si elle est présente, elle doit venir tout à la fin.

Mécanisme d'exceptions (suite)

Une clause **except** représente en fait un **gestionnaire d'exceptions** (*exceptions handler* en anglais) qui attrape (*catches* en anglais) et traite l'**exception** ; son **en-tête** peut aussi être utilisé :

- sans aucune indication (entre le mot clé et le séparateur deux-points) et, dans ce cas, toutes les **exceptions** lancées sont traitées dans le **bloc_except** correspondant à cet en-tête ;
- avec un seul élément de la forme :

  ClasseException as nom_instance

  où **nom_instance** est un identificateur valide associé à une **instance** de la (i.e. un objet de type) **classe** d'**exception ClasseException**.

De plus :

- les **exceptions** placées dans plusieurs en-têtes **except** successifs doivent (normalement) commencer avec la plus spécifique et finir avec la plus générique ;
- la clause **else** permet d'éviter le traitement accidentel d'éventuelles **exceptions** qui n'ont pas été anticipées pour le code protégé par l'instruction structurée **try** ;
- le clause **finally** est en fait un **gestionnaire de nettoyage** (*clean-up handler* en anglais) car, dans le **bloc_finally**, on peut prévoir les opérations à effectuer obligatoirement à la fin des autres traitements.

Mécanisme de délégation

Il convient de mentionner que si une instruction structurée **try** qui comporte une clause **finally** est quittée :

- suite à une instruction **break** ou **continue** qui passe le contrôle en dehors de la boucle qui contenait le **try** ;
- suite à une instruction **return** qui fait sortir de la fonction dont le corps abritait le **try** ;

la clause **finally** est d'abord exécutée.

Une **exception** lancée peut être traitée par un **gestionnaire d'exceptions** (un bloc **except**) approprié conformément à un **mécanisme de propagation** spécifique dit **de délégation**.

Par exemple, si une **exception** est lancée à l'exécution d'une première fonction :

- cette **exception** peut être traitée localement (dans le corps de cette première fonction) ;
- sinon, l'**exception** est propagée vers la fonction appelante (i.e. celle qui a appelé la première fonction) et qui peut traiter l'exception ;
- sinon, la **délégation** du traitement de l'**exception** continue éventuellement jusqu'au programme principal qui est le dernier à pouvoir encore traiter l'exception ;
- sinon, le programme s'arrête avec un **message d'erreur** qui contient des informations utiles au débogage concernant l'**exception** produite et le contexte de son lancement (informations stockées dans un objet "trace d'appels" - *traceback* en anglais).

Mécanisme d'exceptions – fonctionnement

Concrètement, le **mécanisme d'exceptions** présenté auparavant fonctionne de la manière suivante :

- si aucune **exception** n'est lancée dans le bloc **try** :
  - aucun gestionnaire d'exceptions **except** n'est exécuté ;
  - l'éventuelle clause **else** est exécutée, sauf si l'instruction structurée a été quittée à la suite d'une instruction **break**, **continue** ou **return** ;
  - l'éventuelle clause **finally** est toujours exécutée ;
- si une première **exception** est lancée dans le bloc **try** :
  - le bloc **try** est quitté définitivement ;
  - un **gestionnaire d'exception** approprié est recherché en respectant l'ordre des clauses **except** ;
  - dès que le premier **gestionnaire** convenable est trouvé :
    - la clause **except** respective est exécutée ;
    - les éventuelles clauses **except** suivantes et l'éventuelle clause **else** sont ignorées ;
    - l'éventuelle clause **finally** est exécutée ;
  - si aucun **gestionnaire** convenable n'est trouvé parmi les clauses **except** :
    - l'éventuelle clause **finally** est exécutée ;
    - un gestionnaire approprié est recherché dans le code englobant ou dans la pile des appels.

Mécanisme d'exceptions – exemples

Par la suite, on donne :

- dans la colonne de gauche, le **code à exécuter** ;
- dans la colonne de droite, un ou deux **exemples de résultats** obtenus en exécutant le **code de gauche**.

`age = int(input('Votre âge : '))` `print(f'L\'âge = {age} ans.')`	`Votre âge : abc` `ValueError: invalid literal for int() with base 10: 'abc'`

`try:`
`    age = int(input('Votre âge : '))`
`print(f'L\'âge = {age} ans.')`
	`SyntaxError: invalid syntax`

`try:`
`    print('Début try !')`
`    age = int(input('Votre âge : '))`
`    print('Fin try !')`
`except:`
`    print('L\'âge a été corrigé !')`
`    age = 20`
`print(f'L\'âge = {age} ans.')`
	`Début try !`
	`Votre âge : 25`
	`Fin try !`
	`L'âge = 25 ans.`
	`----------`
	`Début try !`
	`Votre âge : abc`
	`L'âge a été corrigé !`
	`L'âge = 20 ans.`

Mécanisme d'exceptions – exemples (suite)

```
try:
    print('Début try !')
    age = int(input('Votre âge : '))
    print('Fin try !')
except Exception:
    print('L\'âge a été corrigé !')
    age = 20
print(f'L\'âge = {age} ans.')
```

```
Début try !
Votre âge : 25
Fin try !
L'âge = 25 ans.
```

```
Début try !
Votre âge : abc
L'âge a été corrigé !
L'âge = 20 ans.
```

```
try:
    print('Début try !')
    age = int(input('Votre âge : '))
    print('Fin try !')
except ValueError:
    age = 20
    print('L\'âge a été corrigé !')
else:
    print('Pas d\'exception !')
print(f'L\'âge = {age} ans.')
```

```
Début try !
Votre âge : 25
Fin try !
Pas d'exception !
L'âge = 25 ans.
```

```
Début try !
Votre âge : abc
L'âge a été corrigé !
L'âge = 20 ans.
```

Mécanisme d'exceptions – exemples (suite)

```
try:
    print('Début try !')
    age = int(input('Votre âge : '))
    print('Fin try !')
except ValueError:
    age = 20
    print('L\'âge a été corrigé !')
else:
    print('Pas d\'exception !')
finally:
    print('Toujours exécuté !')
print(f'L\'âge = {age} ans.')
```

```
Début try !
Votre âge : 25
Fin try !
Pas d'exception !
Toujours exécuté !
L'âge = 25 ans.
------------------------------------------------------------
Début try !
Votre âge : abc
L'âge a été corrigé !
Toujours exécuté !
L'âge = 20 ans.
```

```
try:
    print('Début try !')
    age = int(input('Votre âge : '))
    print('Fin try !')
except ValueError:
    age = 20
    print('L\'âge a été corrigé !')
else:
    print('Pas d\'exception !')
except Exception:
    print('Autre type d\'excpeption !')
print(f'L\'âge = {age} ans.')
```

```
SyntaxError: invalid syntax
```

Mécanisme d'exceptions – exemples (suite)

`try:`
`    age = int(input('Votre âge : '))`
`else:`
`    print('Pas d\'exception !')`
`print(f'L\'âge = {age} ans.')`
	`SyntaxError: invalid syntax`

`try:`
`    age = int(input('Votre âge : '))`
`except ValueError:`
`    age = 20`
`    print('L\'âge a été corrigé !')`
`finally:`
`    print('Toujours exécuté !')`
`else:`
`    print('Pas d\'exception !')`
`print(f'L\'âge = {age} ans.')`
	`SyntaxError: invalid syntax`

`try:`
`    age = int(input('Votre âge : '))`
`except (ValueError as ve, Exception as e):`
`    print(ve)`
`    age = 20`
`    print('L\'âge a été corrigé !')`
`else:`
`    print('Pas d\'exception !')`
`print(f'L\'âge = {age} ans.')`
	`SyntaxError: invalid syntax`

Mécanisme d'exceptions – exemples (suite)

```
try:
    print('Début try !')
    age = int(input('Votre âge : '))
    print('Fin try !')
except ValueError as ve:
    print('Avant correction :', ve)
    age = 20
    print('L\'âge a été corrigé !')
else:
    print('Pas d\'exception !')
print(f'L\'âge = {age} ans.')
```

```
Début try !
Votre âge : 25
Fin try !
Pas d'exception !
L'âge = 25 ans.
------------------------------------------------------------
Début try !
Votre âge : abc
Avant correction : invalid literal for
int() with base 10: 'abc'
L'âge a été corrigé !
L'âge = 20 ans.
```

```
try:
    print('Début try !')
    age = int(input('Votre âge : '))
    print('Fin try !')
except Exception as e:
    print('Avant correction :', e)
    age = 20
    print('L\'âge a été corrigé !')
else:
    print('Pas d\'exception !')
print(f'L\'âge = {age} ans.')
```

```
Début try !
Votre âge : 25
Fin try !
Pas d'exception !
L'âge = 25 ans.
------------------------------------------------------------
Début try !
Votre âge : abc
Avant correction : invalid literal for
int() with base 10: 'abc'
L'âge a été corrigé !
L'âge = 20 ans.
```

Mécanisme d'exceptions – exemples (suite)

```
try:
    age = int(input('Votre âge : '))
    print('Fin try !')
except (TypeError, ValueError) :
    age = 20
    print('L\'âge a été corrigé !')
else:
    print('Pas d\'exception !')
print(f'L\'âge = {age} ans.')
```

```
Votre âge : 25
Fin try !
Pas d'exception !
L'âge = 25 ans.
-------------------------------------------------------------
Votre âge : abc
L'âge a été corrigé !
L'âge = 20 ans.
```

```
try:
    age = int(input('Votre âge : '))
    print('Fin try !')
except TypeError :
    print('Le cas TypeError !')
except ValueError :
    print('Le cas ValueError !')
    age = 20
    print('L\'âge a été corrigé !')
except Exception:
    print('Le cas Exception !')
else:
    print('Pas d\'exception !')
finally:
    print('Toujours exécuté !')
print(f'L\'âge = {age} ans.')
```

```
Votre âge : 25
Fin try !
Pas d'exception !
Toujours exécuté !
L'âge = 25 ans.
-------------------------------------------------------------
Votre âge : abc
Le cas ValueError !
L'âge a été corrigé !
Toujours exécuté !
L'âge = 20 ans.
```

Lancement d'une exception

Parfois, le programmeur peut anticiper qu'une situation spéciale/non désirée/exceptionnelle pourrait se produire dans certaines circonstances quand une instruction (simple ou structurée) est exécutée.

Par conséquent, il peut prévoir une instruction **raise** qui, si la situation anticipée se produit, lance explicitement une **exception** et interrompt le déroulement "normal" du programme qui continue avec le traitement de cette **exception** par un éventuel **gestionnaire d'exceptions** approprié et trouvé **localement** ou grâce au **mécanisme de délégation**.

On dit que l'instruction **raise** lève ou lance une **exception** et elle peut se trouver à tout endroit du code, y compris dans des blocs **try** (qui peuvent être éventuellement imbriqués) ou dans des blocs **except** ou **finally**.

Une telle instruction commence par le mot clé **raise** qui peut être :

- seul, cas où elle **propage** la dernière **exception** active dans la portée courante (par exemple dans un bloc **except**) ;
- suivi par une **classe** d'**exception** (la classe prédéfinie native **Exception** ou une de ses descendantes natives ou définies par le programmeur) ;
- suivi par une **instance** d'une **classe** d'**exception** (le plus souvent créée sur place).

Lancement d'une exception (suite)

Plus précisément, si le mot clé **raise** est :

- seul et s'il n'y a pas d'**exception** active dans la portée courante, alors une exception **RuntimeError** est lancée/levée ;
- suivi par une expression qui correspond à une **classe** d'**exception**, alors une **instance** de cette **classe** est créée (à l'aide du constructeur sans argument de la **classe** respective) et lancée.

Le **bloc_try** d'une instruction structurée **try** contient d'habitude des instructions à risque, potentiellement capables de lancer (au moins) une **exception** qui peut être :

- créée pour l'occasion avec une instruction **raise** à cause d'une situation "anormale" anticipée et qui s'est produite à l'exécution ;
  - dans ce cas, si cette **exception** est traitée par une clause **except** qui vient par la suite (après le bloc **try**), on parle d'un **traitement local** de cette **exception** ;
- propagée et obtenue par délégation, par exemple, suite à l'appel d'une méthode qui a généré (à cause du contexte de son appel) l'**exception** à gérer sans que la méthode l'ait traitée elle-même localement ;
  - dans ce cas, on parle d'un **traitement par délégation** d'une telle **exception**.

Création et lancement d'une exception – exemples

```
def corriger_temp_abs(temp):
    try:
        if temp < 0:
            raise Exception('Temp KO !')
        print('Pas de problème !')
    except :
        if temp > - 273:
            temp = 273 + temp
            print('Traitement local !')
            return temp
        print('Délégation !')
        raise
        print('Après raise !')
    else:
        print('Pas d\'exception !')
        return temp
    finally:
        print('Toujours exécuté !')
print(corriger_temp_abs(20))
print('*' * 20)
print(corriger_temp_abs(-100))
print('*' * 20)
print(corriger_temp_abs(-500))
```

```
Pas de problème !
Pas d'exception !
Toujours exécuté !
20
********************
Traitement local !
Toujours exécuté !
173
********************
Délégation !
Toujours exécuté !
---------------------------------------
Exception: Temp KO !
```

Création et lancement d'une exception – exemples (suite)

```
try:
    print('Début try !')
    age = int(input('Votre âge : '))
    print('Fin try !')
except ValueError:
    age = 20
    print('L\'âge a été corrigé !')
else:
    print('Pas d\'exception !')
finally:
    print('Toujours exécuté !')
    raise Exception('Just for fun !')
    print('Trop tard !')

print(f'L\'âge = {age} ans.')
```

```
Début try !
Votre âge : 25
Fin try !
Pas d'exception !
Toujours exécuté !
-----------------------------------
Exception: Just for fun !
-----------------------------------------------------------------
Début try !
Votre âge : abc
L'âge a été corrigé !
Toujours exécuté !
-----------------------------------
Exception: Just for fun !
```