ENSG Géomatique

ÉCOLE NATIONALE
DES SCIENCES
GÉOGRAPHIQUES

Programmation avec Python

Division des enseignements en informatique

v1.1 - octobre 2016

Table des matières

Introduction

Ce cours est une initiation au langage de programmation Python. Il n'est pas nécessaire de déjà connaître ce langage, ni d'avoir des notions avancées en programmation, pour pouvoir l'aborder. Dans la mesure du possible, chaque notion est illustrée d'exemple. Le cours se concentre sur la description du fonctionnement du langage et des syntaxes employées.

Nous reviendrons en premier sur ce qui fait la spécificité de Python : ses caractéristiques, ses usages. Les structures de bases du langage seront ensuite présentées, ce qui nous permettra de construire nos premiers programmes. Nous aborderons ensuite la modularité en Python pour finir par introduire les bases de la programmation orientée objet dans ce langage. Des notions annexes, mais non moins importantes, comme la documentation, les tests ou la gestion des exceptions s'intercaleront en cours de route.

Des sujets comme la connexion à des bases de données, la réalisation d'interface graphiques ou encore la programmation web ne seront pas abordés dans ce cours. Plus généralement, tout ce qui a trait à l'utilisation de bibliothèques annexes, n'entre pas dans le périmètre de ce cours.

La version 3 de Python est la seule utilisée tout au long du cours.

Conventions d'écriture

Les conventions d'écriture employées dans ce cours sont les suivantes :

- les exemples de codes sont inscrits dans des cadres à fond gris :

```
Exemple de code
```

- le caractère \% correspond à un début de ligne dans une console système (il n'est donc pas à recopier).
- les caractères >>> correspondent à des débuts de lignes dans une console Python (ces caractères ne sont pas à recopier).
- les lignes ne commençant pas par un caractère spécial sont les résultats de l'exécution d'une commande.
- le caractère # correspond au début d'un commentaire.

Partie I

Introduction à Python

Avant d'aborder pleinement la programmation avec le langage Python, nous proposons de nous attarder sur ce qui fait la spécificité du langage Python : son histoire et ses caractéristiques principales en autres. Nous détaillerons ensuite quelques-uns des usages classiques qui peuvent être fait de ce langage. Pour finir cette introduction, nous apprendrons à configurer et utiliser un poste de travail pour pouvoir utiliser le langage Python.

1 L'histoire de Python

L'histoire de Python débute dans les années 1980 à l'institut national de recherche mathématiques et informatiques des Pays-Bas (CIW) sous l'impulsion de Guido van Rossum. Il y travaille sur le développement d'un nouveau langage de programmation, l'ABC, destiné à être un successeur du BASIC et du PASCAL.

L'ABC [1] présente la particularité de proposer une invite de commande interactive conservant automatiquement les entrées saisies. L'imbrication du code y est par ailleurs déterminée par l'indentation des lignes, et non pas par des points virgules comme c'est le cas en C ou en Pascal. Mais ce langage présentait plusieurs contraintes de tailles qui empêchèrent sa diffusion à un large public : lecture/écriture de fichiers difficiles, pas de concept de bibliothèque, système d'entrée/sortie peu souple.

Guido van Rossum est ensuite affecté à un projet de mise en place d'un système d'exploitation distribué. Il est chargé de créer un langage de script pour manipuler le système. S'inspirant fortement de l'ABC, dont il essaya de gommer les limitations, il conçut les premières versions d'un langage qu'il appela Python en hommage aux Monty Python qu'il admire.

La première version publique de Python, numérotée 0.9.0, est postée sur un forum en février 1991. Guido von Rossum quitte le CIW en 1995. A cette date Python en est à la version 1.2. Le langage séduit par les possibilités offertes d'ajout de nouveaux objets à partir de fichiers de code Python, de fichiers compilé en C, C++ ou encore Fortran.

Guido von Rossum continue néanmoins de travailler sur le langage Python les cinq années suivantes, en étant soutenu par des fonds de recherche. Python en est alors à la version 1.6.

Puis Guido von Rossum forme l'équipe PythonLabs qui travaillera au sein de différentes structures. Elle permettra à la version 2.0 d'être publiée dès 2000. En 2001 la Python Software Fundation est crée pour supporter le développement de Python et les versions 2.x se poursuivent alors jusqu'à la version 2.7 en 2010. Un système de gestion des compatibilités ascendantes et descendantes entre version est mis en place via un module spécifique, le module __future__.

1. Pour plus d'informations sur l'ABC on pourra se reporter à l'ouvrage ABC Programmer's Handbook, Guerts, Meertens, Pemberton, 2005, édition Prentice-Hall

Une version 3.0 est envisagée pour permettre de rectifier certains défauts du langage (redondance ou obsolescence de méthodes anciennes par exemple) ne pouvant être corrigés sans casser la compatibilité ascendante. Elle verra le jour en décembre 2008. Les développements se poursuivront et Python 3.5 sortira en septembre 2015.

Depuis la version 2.0.1, Python est distribué par la Python Software Fundation (PSF) sous une licence LGPL (Lesser GNU Public License). Cela signifie que les outils Python peuvent être utilisés sans restriction pour produire des logiciels de tous types, même si ceux-ci sont distribués avec une licence plus restrictive que la GPL. Il est possible de combiner des outils Python à d'autres technologies qui ne seraient pas sous licence GPL pour réaliser des programmes. Le code source de Python lui-même est disponible et modifiable.

2 Caractéristiques du langages

Avant de commencer à apprendre un langage de programmation, il convient de savoir pourquoi utiliser ce langage plutôt qu'un autre. Dans cette sous-partie, nous nous proposons donc d'explorer les caractéristiques essentielles du langage.

Python est portable. On distingue les langages de programmation compilés des langages interprétés. Les premiers ont pour eux de meilleures performances, mais il nécessitent d'être compilés sur chacune des machines où ils seront exécutés. Les langages interprétés sont généralement moins performants mais peuvent être exécutés directement sur une machine compatible, sans avoir besoin d'être compilé au préalable. Python est un langage interprété compatible avec la plupart des OS du marché (Microsoft Windows, Mas OS, Linux, Android, etc.).

Python est un langage orienté objet et fonctionnel. Python permet de mettre en oeuvre les concepts de l'orienté objet (polymorphisme, héritage, etc.). Mais Python ne s'arrête pas là : c'est aussi un langage de script qui rend possible la programmation fonctionnelle. Il permet ainsi une grande variété d'approches pour résoudre un problème.

Python est à typage dynamique fort. Cela signifie qu'il n'est pas nécessaire de déclarer le type des données que l'on manipule avant de le manipuler. Mais qu'il n'est pour autant pas possible de manipuler ensemble des données dont les types sont incompatibles entre eux.

Python est performant. Python n'est pas un langage interprété "classique". A l'instar du Java, le code source Python est transformé lors de la première exécution en bytecode Python, plus proche du langage machine que le code source, et donc plus rapide lors des exécutions suivantes. Python implémente également un mécanisme de "garbage collector" qui permet de libérer de l'espace mémoire pendant l'exécution du programme et ainsi d'optimiser l'exécution des programmes.

Python est associable. On entend par là que Python peut facilement être combiné à d'autres langages de programmation : il est possible d'appeler une bibliothèque C ou C++ dans un programme Python et inversement pour étendre les fonctionnalités de Python ou améliorer ses performances.

Python est facile à lire et à écrire. En Python, pas de point virgule ou d'accolade pour

indiquer la fin d'une instruction ou d'un bloc de code, c'est l'indentation qui fait la structure du programme. Cela rend les programmes écrit en Python très faciles à lire, même pour un débutant en programmation. Par ailleurs, la syntaxe du langage est très simple, notamment comparée à d'autres langages comme le C ou le Java. Un des atouts majeur de Python est ainsi la facilité d'écriture d'un programme.

Python est un langage de haut niveau. Il permet donc d'écrire des programmes en faisant abstraction du matériel utilisé. Il n'y a pas à se soucier de la gestion de la mémoire via l'utilisation de pointeur : les variables sont toujours passées par référence.

Python possède une communauté très active. Python est libre et gratuit, mais cela n'est pas pour autant synonyme d'absence de support. La communauté présente sur les forums internet répond aux question des utilisateurs avec la même efficacité (rapidité, précision) qu'un support commercial. Dans le même ordre d'idées, la documentation est complète et à jour.

3 Python, pour quoi faire ?

Parfois considéré comme un simple langage de scripts, Python révèle bien d'autres possibilités lorsque l'on s'y intéresse un peu plus. Le champs d'application de Python est très vaste et l'IEEE (Institut des Ingénieurs Electriciens et Electroniciens ; organisation dédiée à l'ingénierie et aux sciences appliquées du monde) cite souvent Python comme l'un des cinq langage de programmation les plus utilisés au monde aux côté du Java, C, C++ et C# (enquêtes se basant sur une dizaine de critères comme les requêtes Google, les commits Github, les offres d'emploi du site Career Builders aux Etats-Unis, etc.).

Le langage Python est par exemple utilisé pour :

— Du scripting (adminstration, lecture/manipulation de fichiers, parsing, etc.). C'est la base du langage.

— Des scripts en environnements métier. Dans le monde de la géomatique, ArcGIS, QGIS, ERDAS, Autodesk utilisent Python comme langage de script. C'est également le cas de GIMP, Inkscape ou Gedit.

— Des calculs scientifiques et du big data. La NASA affirme par exemple utiliser Python comme langage de programmation.

— Des applications bureautiques : BitTorrent ou DropBox sont écrit en Python.

— Des extensions. C'est possible par exemple dans ArcGIS ou QGIS.

— Des jeux vidéo : Civilization IV ou Battlefield 2 utilisent Python.

— Du web. Python fait parti des langages de programmation utilisés par Google. La plateforme YouTube est écrite en Python. Les groupes de discussion Yahoo l'utilisent également.

— Des films d'animation : Pixar utilise Python pour produire ses films d'animation.

PIXAR

— Python est également utilisé par des constructeurs de composants informatiques (Intel, Hewlett-Packard, Seagate, IBM) pour tester les performances de leurs nouveaux produits.

Pour d'autres exemples d'applications utilisant Python comme langage de programmation, on pourra se reporter à http://www.python.org/about/success/.

Citons également ici quelques bibliothèques disponibles en Python. Même si l'objet de ce cours ne sera pas de les étudier chacune en détail, il est intéressant d'en connaître quelques unes pour avoir un aperçu des possibilités offertes par le langage :

— administration système (gestion de fichier, expression en ligne de commande, etc.) : Sh, Grin, Env, ;
— manipulation de réseaux : Scapy, Paramiko, Tornado, Python-oauth2 ;
— interfaces graphiques : Tkinter, WxWindows, PyQt ;
— développement web : Django, Bottle, CherryPy, Pelican ;
— intégration de code d'autres langages : Win32, Ctypes ;
— utilisation de bases de données relationnelles : mysql-python, Psycopg, PyODBC, sqlite3 ;
— mais également NoSQL : PyMongo, Redis, CouchDB ;
— gestion de données géographiques : Shapely, PySHP, osgeo.ogr, Fiona, ArcPy, PyQGIS, SpatiaLite ;
— programmation scientifique : Numpy, Scypy ou Matplotlib, Pandas ;
— manipulation d'images : Pillow, PyOpenGL.

4 Exécution d'un programme par Python

Lorsque l'on exécute un code Python, un programme lit notre code et exécute les instructions qu'il contient. Ce programme est appelé un interpréteur Python. Un interpréteur est une sorte de programme qui exécute d'autres programmes.

Lorsque Python est installé sur une machine, un interpréteur Python est systématiquement installé. Cet interpréteur peut prendre des formes diverses (exécutable, jeu de librairies liées dans un programme) et est lui-même implémenté dans un langage de programmation (C, Java ou autre). Sa présence est indispensable pour pouvoir exécuter du code écrit en Python.

L'installation de l'interpréteur Python dépend du système d'exploitation :

— Python est généralement installé en standard sur les postes Linux et Mac OS X ;
— si ce n'est pas le cas, l'utilisateur devra compiler Python à partir des fichiers sources de sa distribution ;

— sur Windows, l'installation de Python à partir d'un fichier de setup est généralement à faire par l'utilisateur.

Se reporter au site web http://www.python.org pour le téléchargement de la bonne version de Python.

Remarque : certains programmes Windows embarquent leur propre version de Python (ArcGIS ou QGIS par exemple dans le monde de la géomatique). Plusieurs versions de python peuvent alors parfois cohabiter sur le même poste et il convient alors d'être vigilent lors de l'installation de libraires pour être certain qu'elles seront liées à la bonne version de Python.

Pour illustrer le fonctionnement de l'interpréteur Python, nous exécuterons un programme simple se contentant d'afficher *"Hello world !"* dans le flux de sortie. Le fichier .py à exécuter contient uniquement l'instruction suivante :

```
print("Hello world !")
```

On peut l'exécuter en faisant `python hello_world.py` dans une terminal, en s'étant préalablement placé dans le répertoire contenant le fichier *hello_world.py*, comme montré figure 1.

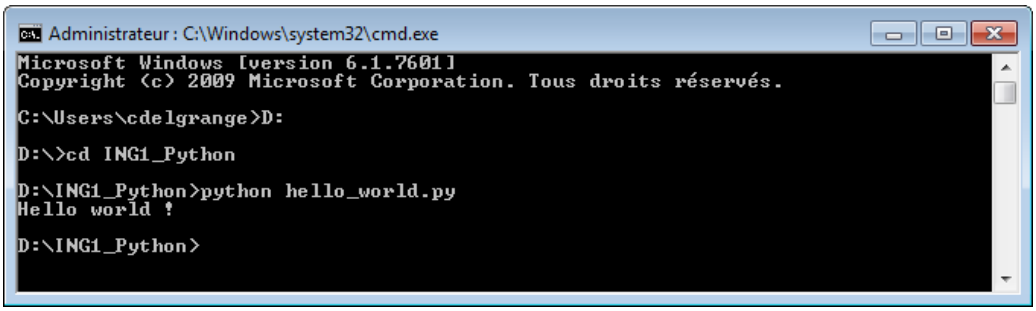

FIGURE 1 – Hello world en Python

Le code est en réalité compilé en bytecode. Un fichier .pyc est automatiquement créé et c'est ce fichier qui est ensuite interprété par la machine virtuelle Python. La figure 2 représente ce schéma.

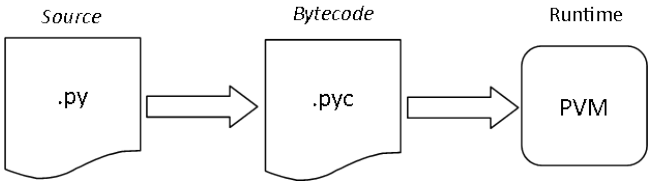

FIGURE 2 – Fonctionnement de l'interpréteur Python : le code source est traduit en bytecode qui est interprété par la machine vituelle Python (PVM)

Le bytecode n'est pas aussi performant que du C ou C++ compilé puisque la machine virtuelle doit toujours interpréter le bytecode, ce qui est plus couteux en temps que d'exécuter des instructions CPU. Mais, contrairement à un langage interprété classique, Python n'a pas besoin de ré-analyser et re-parser le code source à chaque exécution. La vitesse d'exécution d'un programme Python se situe ainsi quelque part entre celles d'un langage compilé classique et d'un langage interprété classique.

5 Exécuter un programme en Python

5.1 L'invite de commande interactive

Le premier moyen de faire du Python est d'utiliser l'invite de commande interactive Python (on parle également d'interpréteur intéractif). En supposant que Python ait correctement été installé sur le système, on accède à interpréteur interactif en tapant `python` dans une invite de commande.

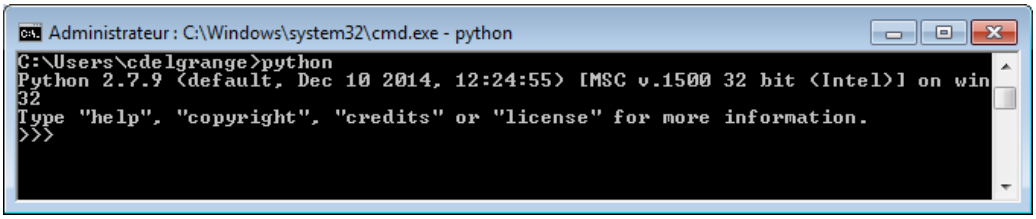

FIGURE 3 – Démarrage classique de l'interpréteur interactif Python

Lorsque plusieurs versions de Python cohabitent sur le même système, il est possible de préciser quelques options pour appeler une version spécifique. On utilise pour cela la commande `py -v` où v est le numéro de la version (note : cette technique n'est plus valable à partir de la version 14.04 d'Ubuntu). Par exemple :

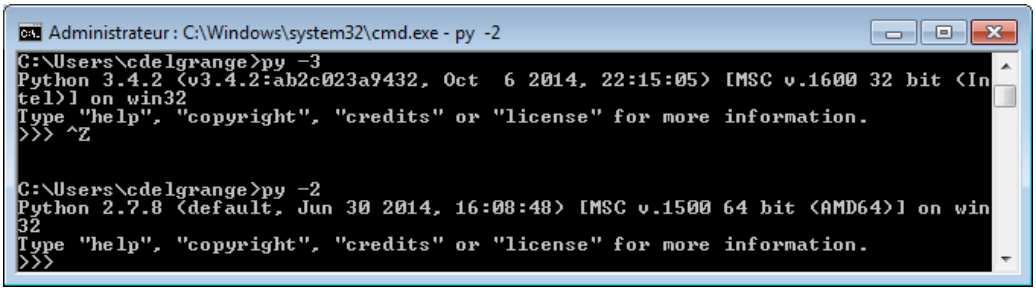

FIGURE 4 – Démarrage d'une version spécificique de l'interpréteur interactif Python

Sur un poste Windows, on peut également accéder à la ligne de commande Python interactive via le menu démarrer (cf. figure 5).

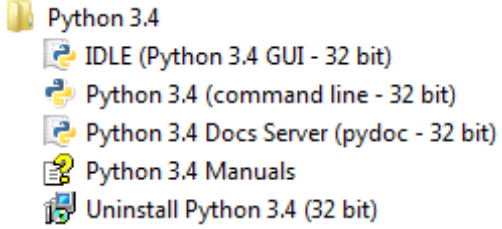

FIGURE 5 – Menu Python 3.4 du Menu Démarrer de Microsoft Windows 7 lorsque l'installation de Python a été effectuée

Sur un poste Linux où une version Python 2.X cohabiterait avec une version 3.X, la commande python3 permet d'appeler directement la version 3.X de Python, tandis que la commande python lancera généralement la version 2.X (cf figure 6).

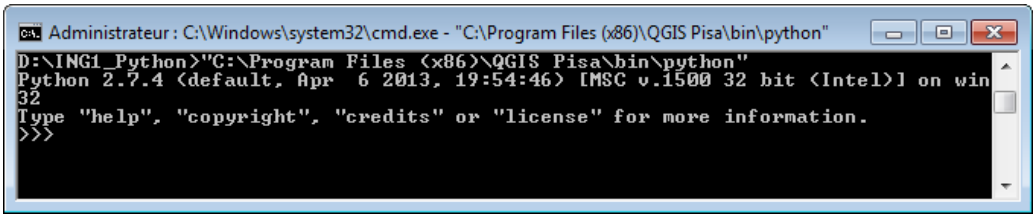

FIGURE 7 – Démarrage de la version de l'interpréteur Python embarquée par QGIS

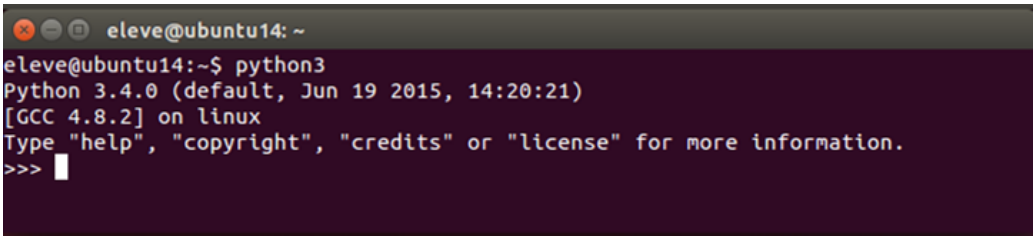

FIGURE 6 – Démarrage d'une version spécifique de l'interpréteur interactif Python sur un poste Linux

Si la commande `python` n'est pas reconnue par votre système, c'est probablement que le chemin d'installation de Python n'est pas présent dans la variable d'environnement PATH. Dans ce cas, il est toujours possible d'appeler explicitement Python comme illustré figure 7. Cela peut être notamment utile, sur un poste Windows, pour exécuter une version embarquée par un logiciel.

La ligne de commande interactive permet d'exécuter des instructions Python. Les instructions sont interprétées et exécutées dès qu'elles ont été saisies.

```
>>> print("Hello world !")
Hello world !
```

La fenêtre garde toutefois en mémoire tout ce qui a déjà été exécuté. Par exemple :

```
>>> x = 3
>>> y = 2
>>> print(x + y)
5
```

Remarque : pour quitter l'interpréteur Python interactif, on utilise Ctrl-D sur un poste Linux, ou Ctrl-Z + Entrer sur un poste Windows.

Notons qu'il est également possible d'appeler depuis une console l'interpréteur Python pour exécuter un script (c'est ce que nous avons fait dans le premier exemple avec le script hello_world.py, cf. page ??). Les schémas vu précédemment pour ouvrir l'interpréteur interactif sont valables également pour exécuter un script :

— `py -v` pour utiliser un numéro de version spécifique de Python ;
— `python3` pour utiliser Python 3 ;
— `C:\chemin_vers_une_version_de_python\python` pour utiliser une version précise.

On retiendra également qu'il est possible d'écrire le flux de sortie (les `print(...)` dans un fichier texte plutôt que de les afficher dans la console. Pour cela on utilisera la syntaxe :

```
% python script.py > result.txt
```

En résumé, l'invite de commande interactive est très utile pour tester des instructions à la volée. En cas de doute sur un extrait de code Python, pour tester une fonction ou un script,

c'est l'outil idéal. Mais comme elle ne permet pas de sauvegarder le code, elle se révèle peu efficace pour écrire des programmes en entier. Pour cela, on utilisera plutôt un éditeur de texte.

Remarque : double-clic sur les fichier .py sous Windows Fichiers .py associés à Python automatiquement sous Windows. Double-cliquer dessus lance interprète de manière transparente. En revanche, si votre script contient uniquement des instructions rapides, quelques `print` et se termine, vous n'aurez probablement pas le temps de voir ce qu'il s'est passé. Windows ouvre une invite de commande DOS, appelle l'interprète et exécute le script, affiche les résultats dans l'invite de commande et referme celle-ci une fois les traitements terminés. Pour avoir le temps de visualiser les résultats, on utilisera une astuce qui consiste à ajouter une instruction `input()` à la fin du programme (question à l'utilisateur).

Mais si une erreur survient avant la fin de l'exécution, l'instruction `input()` ne sera pas lue et l'invite de commande se refermera sans laisser le temps de voir de quelle erreur il s'agit. Bref, sauf à y être contraint pour des besoins de production, on évitera cette méthode pour exécuter un programme Python.

5.2 IDLE

IDLE est un outil doté d'une interface graphique et qui permet d'éditer, d'exécuter et de débuger des programmes Python. Il est inclut à l'installation standard de Python et est disponible sur la plupart des plateformes (Windows, Linux, Mac, etc.). IDLE est lui-même écrit en Python (il s'agit en fait uniquement d'un script Python présent dans le répertoire d'installation).

A l'ouverture, on retrouve une console Python, semblable à celle que l'on a utilisé dans le paragraphe précédent (cf. figure 8). Quelques différences notables avec la l'invite de commande Python classique sont tout de même à signaler :

— coloration syntaxique des expressions ;
— auto-complétion avec la touche *Tab* ;
— aide contextuelle lors de l'ouverture d'une parenthèse ;
— liste de choix lorsque l'on veut utiliser une méthode ou un attribut.

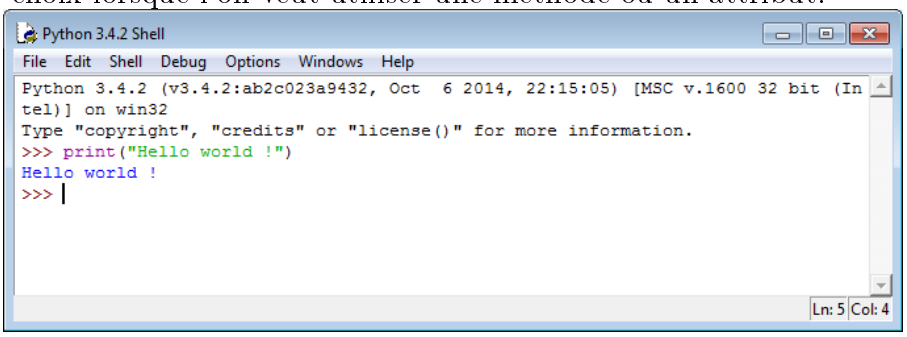

FIGURE 8 – Console Python de l'interface IDLE

IDLE permet également de parcourir les bibliothèques disponibles (celles installées pour la version de Python utilisée). Enfin, pour créer un programme Python, il faut actionner le menu *File* > *New File* qui ouvre une seconde fenêtre fonctionnant comme un éditeur de texte avec les même avantages que la console IDLE (coloration, auto-complétion, aide contextuelle, etc. ; cf. figure 9). Le programme peut être exécuté via le menu *Run* > *Run Module*, ce qui initialise l'interpréteur Python.

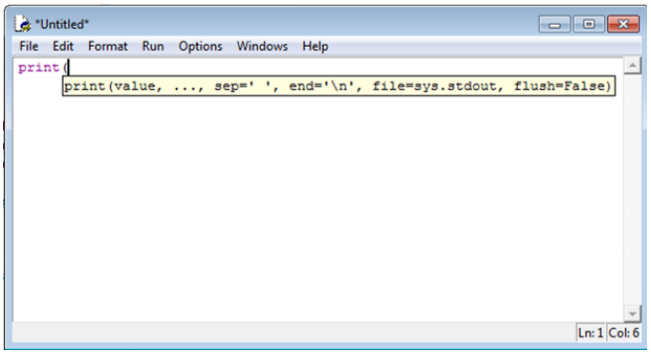

FIGURE 9 – Fenetre de programme Python de l'interface IDLE

Remarque : l'interface et le fonctionnement d'IDLE peuvent être légèrement personnalisés pour rendre son fonctionnement plus agréable. Le lecteur intéressé visitera le menu *Options* > *Configure IDLE* et testera les différentes options.

5.3 Environnements de développement intégrés

Il existe de nombreux environnements de développement permettant d'écrire des programmes en Python. Cette partie se contentera d'en lister quelques uns. Le choix d'un environnement de développement intégré (IDE) pour Python relèvera essentiellement du goût personnel pour un outil plutôt que de critères techniques.

- **PyScripter** [2] : environnement de développement léger qui intègre un éditeur de texte, une console Python et un explorateur de solution dans la même fenêtre. Permet de faire du débogage de code.
- **PyCharm** [3] : environnement de développement complet de l'éditeur Jet Brains. Une version gratuite et une version payante comprenant plus de fonctionnalités (intégration de framework web Django, Pyramid, etc. ainsi qu'une interface de gestion de bases de données).
- **Eclipse et PyDev** [4] : Eclipse est un environnement de développement prenant en charge de nombreux langages de programmation, dont Python via l'extension PyDev.

5.4 Autres outils utiles

IPython (Jupyter) / Notebook : possibilité d'intégrer du code Python à du texte et d'exécuter ce code Python pour voir le résultat. Très utile pour l'étudiant pour prendre des notes, sauvegarder des morceaux de code avec les explications qui vont avec.

2. http://sourceforge.net/projects/pyscripter/
3. http://www.jetbrains.com/pycharm/
4. http://www.pydev.org/

Partie II

Types et opérations

Ce chapitre va nous permettre de commencer à utiliser le langage Python. Nous nous limiterons pour commencer à des opérations basiques comme des opérations mathématiques ou des manipulations de chaînes de caractères. Plus précisément, nous nous limiterons dans ce chapitre aux opérations sur des types de base définis dans le langage.

Avant cela, nous définirons ce que sont une variable et un type en informatique. Nous introduirons un concept important en Python, le typage dynamique, et montrerons comment sont gérées données dans ce langage.

6 Variables et types en Python

6.1 Généralités

La programmation consiste essentiellement à manipuler des données. Ces données de natures diverses sont stockées sous forme binaire dans la mémoire de l'ordinateur. Pour y accéder, un programme définit des **variables**.

Nous définirons ainsi une variable comme l'association d'un nom et d'une valeur. Le nom est utile au programmeur pour savoir quelle donnée il manipule, tandis que l'ordinateur utilise la valeur associée. De manière imagée, le nom peut être vu comme une simple étiquette sur une donnée.

En Python, le nom (on parle parfois d'**identificateur**) :
— ne peut contenir que des lettres, des chiffres et des blancs soulignés ;
— ne peut pas contenir d'espace, d'accent ou de caractères spéciaux (`éç...`) ;
— ne peut pas commencer par un chiffre ;
— distingue minuscules et majuscules (on dit que le langage est **sensible à la casse**) ;
— doit être unique.

Par convention, pour les noms de variable, on séparera les mots par des blancs soulignés : `mon_nom_de_variable`. Les noms de variable commençant par deux _ et se terminant par deux _ sont réservés au langage (exemple : `__str__`). Certains mots-clé sont également réservés par le langage. Il n'est pas possible de créer une variable portant l'un de ces noms (cf. table 1).

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

TABLE 1 – Mots réservés du langage Python

En Python, la valeur d'une variable est une référence vers une adresse mémoire (=un emplacement précis dans la mémoire vive de l'ordinateur). Le contenu de la variable est stocké à cet adresse mémoire sous forme d'une suite de nombres binaires. Pour distinguer les différents contenus, les langages de programmation utilisent des **types** de variable qui permettent de spécifier le contenu attendu.

Contrairement à d'autres langages, il n'est pas nécessaire de déclarer le type d'une variable avant de pouvoir l'utiliser. Plus précisément, l'interpréteur Python détecte lui-même le type de l'objet en référence lorsque la variable est utilisée dans une expression.

Par exemple, lorsque l'on écrit `x = 3`, on crée une variable `x` à laquelle on associe l'objet `3` qui est stocké dans la mémoire de l'ordinateur. L'interpréteur Python détecte que `3` est un nombre entier et associe donc ce type à la variable `x`.

Il est important ici de faire la distinction entre les variables et les objets liés : dans `x = 3`, la variable `x` ne contient pas directement la valeur `3`, mais un objet `3` est associé à cette variable. Concrètement, lorsque l'instruction `x = 3` est saisie, les étapes suivantes sont enchaînées :

— création d'un objet représentant la valeur `3` ;
— création d'une variable `x`, si elle n'existe pas déjà ;
— association de la variable `x` à l'objet `3`.

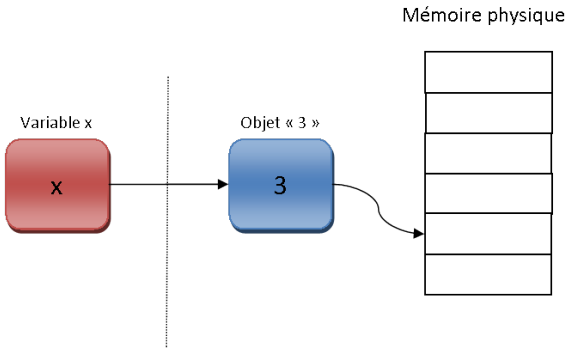

FIGURE 10 – Processus d'affectation d'une valeur à une variable. La variable x devient une référence de l'objet '3').

Cette association d'une variable à un objet est appelé **référence**. De manière imagée, on peut voir une variable comme une étiquette sur un objet.

Remarque : la fonction `id()` permet de récupérer l'adresse mémoire d'un objet. Jouons avec

cette fonction :

```
>>> x = 3
>>> id(x)
19774651  # c'est l'adresse mémoire de a dans la table système
>>> y = "ensg"
>>> id(y)
23154984
>>> id(2)
56489413 # l'objet 2 possède une adresse mémoire même s'il n'est
   pas référencé par une variable
```

Sur chaque ordinateur les chiffres devraient être différents. Ce qui importe c'est de constater qu'ils sont tous différents. De même, on devrait observer que l'adresse mémoire d'un même objet est toujours identique :

```
>>> x = 10
>>> id(y)
1446297760
>>> b = x
>>> id(y)
1446297760
>>> id(10)
1446297760
```

Python garde un objet en mémoire tant qu'il existe une référence vers cet objet :

```
>>> x = 3
>>> y = x
>>> x = "a" # x référence un nouvel objet "a"
>>> y
3           # la variable y référence toujours l'objet 3
>>> y = 5
```

Suite à la dernière ligne de l'exemple ci-dessus, l'objet 3 n'est plus référencé par aucune variable. Python va lancer un processus de nettoyage (aussi appelé *garbage collector*) pour effacer cet objet de la mémoire. Voir illustration figure 11.

Un fois qu'un objet a été supprimé, si on en recrée un ayant la même valeur, il s'agira d'un nouvel objet différent du premier (son adresse mémoire est différente) :

```
>>> x = "ensg"
>>> id(x)
47017920
>>> del(x)
>>> y = "ensg"
>>> id(y)
49017216
```

Pour conclure ce paragraphe, nous retiendrons :
— qu'une variable est créée lorsqu'on lui affecte pour la première fois une valeur ;
— qu'une variable n'est jamais associée à un type en Python. C'est l'objet qu'elle référence qui porte le type ;
— qu'une variable est automatiquement remplacée par l'objet qu'elle référence dans une expression.

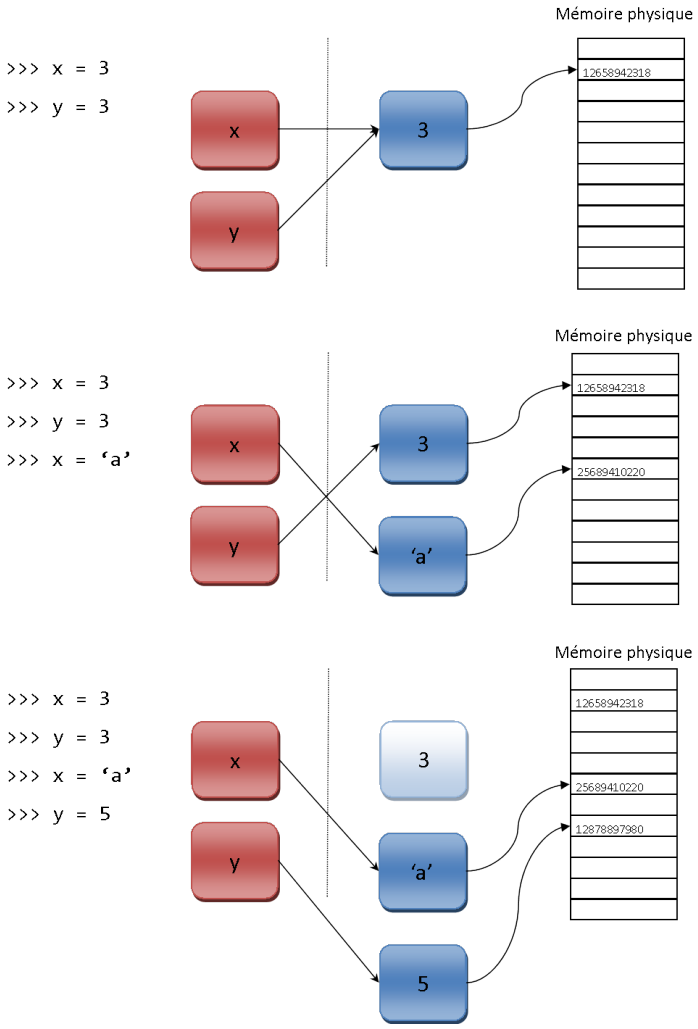

FIGURE 11 – Illustration du processus de nettoyage lorsqu'une suite d'instructions est saisie.

6.2 Types de base

Python met à disposition du programmeur un certain nombre de types ou structures de données. On parle de **types intégrés** (*builtins*). Nous verrons au chapitre VI qu'il est également possible de définir ses propres structures de données.

Quel est donc l'intérêt d'utiliser les structures de données intégrés au langage ? Tout d'abord, nous remarquerons que ces types de base sont des composants naturels des structures manipulées dans les programmes (on manipule naturellement des entiers, des chaînes de caractères). Cela facilite donc grandement la programmation : il n'y a pas besoin de tout redéfinir pour chaque nouveau programme. Enfin, ces structures de données sont plus efficaces que des structures de données que l'on aurait reconstruites (les algorithmes ont été optimisés, écrit dans des langages performants comme le C).

Type	**Exemples de valeur**
Nombre	`3, 1.1456, -3.5`
Chaîne de caractère	`'mot', 'petit texte'`
Liste	`[1, 2, 3], ['a', 'b'], ['a', 5, 2.3, 'f']`
Tuple	`(1, 3), ('a', 'b', 'c')`
Ensemble	set('abc'), 'a', 'b', 'c'
Dictionnaire	1 :'a', 2 :'b'
Fichier	`open('fichier.txt')`
Booléen	True, False
Rien	`None`

TABLE 2 – Types de base en Python

7 Les types numériques

Les nombres sont des objets relativement basiques et dont l'utilisation est assez naturelle. Python inclus les types numériques "classiques", que l'on retrouve dans la plupart des langages de programmation : nombres entier, nombres à virgule ; mais également d'autres moins courants : nombres complexes, nombres rationnels. Ces types couvrent la plupart des besoins standards.

Type	**Exemple de valeur**
Integer	2, -3, 156235789
Float	2.3, -4.12, 1.0, 189489.23891
Complex	2 + 3j
Decimal	Decimal('0.1')
Fraction	Fraction(1, 3)

TABLE 3 – Types numériques

Remarque : les ensembles (cf. paragraphe 12) sont parfois comptés dans les types numé-

riques car ils supportent les opérations mathématiques classiques : addition, soustraction, union, intersection...

Les nombres Python supportent les opérations mathématiques classiques listées dans la table 4. Dans le cas d'expression avec des parenthèses, Python applique la règle mathématique classique et calcule en premier les sous-expressions entre parenthèses.

Opérateur	**Signification**
+	Addition
-	Soustraction
*	Multiplication
**	Puissance
/	Division
//	Division entière
%	Reste d'une division entière (modulo)

TABLE 4 – Opérateurs des types numériques

7.1 Nombres entiers et réels

Python supporte les **nombres entiers** (`int`) et les **nombres réels** (`float`).

```
>>> a = 3
>>> b = 3.0
>>> type(a)
<class 'int'>
>>> type(b)
<class 'float'>
```

Les fonctions `int()` et `float()` permettent de convertir un nombre quelconque en entier ou réel respectivement.

```
>>> a = 3
>>> print(a)
3
>>> print(float(a))
3.0
>>> b = 3.0
>>> print(b)
3.0
>>> print(int(b))
3
```

Si les types sont mélangés au sein d'une même expression, Python convertit automatiquement le résultat vers le type le plus "complèxe".

```
>>> 2 + 3.18
5.18
```

D'une manière générale, les opérations mathématiques complexes retournent systématiquement un réel [5] :

5. Nous reviendrons plus tard dans ce cours sur l'instruction import employée dans l'exemple. Nous l'utiliserons pour l'instant sans savoir ce qu'elle fait réellement.

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

7.2 Nombres complexes

Les **nombres complexes** sont gérés en standard en Python. Ils sont représentés en ajoutant un `j` ou `J` à la partie imaginaire du nombre complexe.

```
>>> 1j * 1j
(-1+0j)
>>> 2 + 3j
(2+3j)
>>> 2j * (2 + 3j)
(-6+4j)
```

Attention, la syntaxe rend indispensable la présence du nombre devant le `j` de la partie imaginaire, même s'il vaut 1. L'instruction `j * j` retourne par exemple une erreur. C'est `1j * 1j` qui est compris par Python.

7.3 Autres types numériques

Python possède également un type **Decimal** qui permet de gérer des nombres décimaux avec une précision définie par l'utilisateur. Cela est particulièrement utile dans les cas où la la conversion des nombres en binaires par la machine introduit des erreurs :

```
>>> 0.1 + 0.1 + 0.1 - 0.3
5.551115123125783e-17
```

La syntaxe de création d'un nombre décimal est différentes des types précédents, puisqu'au lieu d'écrire un simple expression littérale, il faudra appeler une fonction.

```
>>> from decimal import Decimal
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1')  - Decimal('
    0.3')
Decimal('0.0')
>>> print(Decimal('0.1') + Decimal('0.1') + Decimal('0.1') -
    Decimal('0.3'))
0.0
```

Le type permet également de visualiser la valeur réellement stockée par l'ordinateur d'un nombre. Pour cela on n'utilise pas les apostrophes encadrant le nombre dans la fonction `Decimal` :

```
>>> Decimal('0.1')
Decimal('0.1')
>>> Decimal(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625'
    )
```

Enfin, les **nombres rationnels** (Fraction) ont été introduit en Python 2.6. Ils permettent de gérer des fractions sans perte de précision. La syntaxe est similaire à celle des nombres décimaux.

```
>>> 1/6 + 1/2
0.6666666666666666
>>> from fractions import Fraction
>>> Fraction(1, 6) + Fraction(1, 2)
Fraction(2, 3)
```

Notons qu'il est toujours possible de revenir à une forme plus classique :

```
>>> print(Fraction(1, 6) + Fraction(1, 2))
2/3
>>> float(Fraction(1, 6) + Fraction(1, 2))
0.6666666666666666
```

Des modules standards[6], comme *math* étendent par ailleurs largement les opérations possible sur les types numériques (racine carrée, exponentielle, sinus, valeur absolue, etc.).

7.4 Les booléens

Pendant longtemps, Python n'a pas eu de type booléen et utilisait l'entier 0 pour faux et l'entier 1 pour vrai. Puis une sorte de sur-couche a été ajoutée à ces entiers 0 et 1 pour rendre le langage plus propre et permettre d'utiliser un vrai type booléen. Les valeurs **True** (vrai) et **False** (faux) sont les deux seules valeurs permises.

```
>>> type(True)
<class 'bool'>
```

Concrètement, les booléens `True` et `False` ont le même comportement que les entiers 1 et 0, à l'exception que quelques méthodes qui ont été redéfinies (`str` pour l'affichage par exemple). La plupart de opérations sur les types numériques sont donc compatibles avec les booléens.

```
>>> True + 2
3
>>> float(False)
0.0
```

Si le booléen `True` vaut 1, l'objet `True` est bien un objet différent de 1 :

```
>>> True == 1
True
>>> True is 1   # l'opérateur is compare les adresse mémoire des
   objets
False
```

Les opérations logiques retournent des booléens. La table 5 liste les opérateurs de comparaison. Ces opérateurs s'appliquent à tous les types de données, à condition que les types soient comparables (comparer 5 à "toto" n'a pas vraiment de sens).

6. Nous définirons précisément ce qu'est un module au chapitre 19

Opérateur	Signification
>	Supérieur
<	Inférieur
>=	Supérieur ou égal
<=	Inférieur ou égal
==	Egal
!= ou <>	Différent de
is	Objet identique
is not	Objet différent

TABLE 5 – Opérateurs logiques

Dans l'exemple ci-dessous, nous affectons à x le résultat de la comparaison `3 != 5`. Ce résultat est un booléen :

```
>>> x = (3 != 5)
>>> x
True
```

Les expressions booléennes peuvent enfin être combinées entre elles à l'aide des mots clé `and`, `or` et `not`.

Opérateur	Signification
X and Y	Est vrai si X et Y sont vrai
X or Y	Est vrai si X ou Y est vrai
not X	Est vrai si X est faux

TABLE 6 – Opérateurs booléens

Exemple :

```
>>> x = ((2 != 3) or (4 < 1)) and (not 5 == 2)
>>> x
True
```

8 Le type "rien"

Python propose un type spécial qui ne peut prendre qu'une seule valeur possible, **None**, et qui est utilisé pour signifier qu'une variable ne contient rien. Ou plus exactement, elle référence un objet et cet objet est l'objet "rien".

Il s'agit d'une valeur qui est utilisée par défaut lorsqu'une opération n'a pas de retour explicite ou n'a pas pu être menée à son terme. Le type "rien" est ainsi utilisé pour vérifier que des parties de programme ont correctement fonctionné.

```
>>> x = None
>>> x
>>> type(x)
<class 'NoneType'>
```

9 Les chaînes de caractères

Après avoir introduit les principaux types numériques, nous allons nous attarder sur un autre type de données fondamental : les **chaînes de caractères**. Une chaîne de caractères est définie comme une suite finie de caractères.

En Python, il n'existe qu'un seul type de données textuelles, qu'il s'agisse de caractère individuel, de mot ou de phrase ou de textes complets. Ce type est accompagné d'un nombre conséquent d'outils et de méthodes.

9.1 Création d'une chaîne de caractères

Les apostrophes et guillemets s'utilisent indifféremment pour définir une chaîne de caractères `"..."` ou `'...'` :

```
>>> "demain" == 'demain'
True
```

L'intérêt d'utiliser l'une ou l'autre des deux formes est de pouvoir intégrer des apostrophes ou guillemets dans une chaîne :

```
>>> "aujourd'hui", 'le type "rien"'
```

Si vous êtes amené à devoir utiliser des apostrophes au sein d'une chaîne définie à l'aide d'apostrophes, il vous faudra utiliser un anti-slash avant l'apostrophe à conserver dans la chaîne :

```
>>> 'aujourd\'hui'
```

D'une manière plus générale, les anti-slashs sont utilisés pour signaler un caractère spécial :
— `\n` pour une nouvelle ligne ;
— `\t` pour une tabulation ;
— `\\` pour un anti-slashe ;
— etc.
Nous remarquerons que cette syntaxe n'est pas des plus pratiques lorsqu'il s'agit d'enregistrer un chemin d'accès. Par exemple, `'C:\nouvel_an\photos\tri'` ne va pas être lu correctement par Python (un retour à la ligne et une tabulation remplaceront les `\n` `\t`). Aussi, pour simplifier l'écriture des chaînes de caractères comportant beaucoup de caractères spéciaux, Python propose d'utiliser une syntaxe qui va interpréter les `\` comme de simples `\` : les *raw strings*.

Si un `'r'` apparaît juste devant une chaîne de caractère, les caractères sont interprétés comme ils apparaissent :

```
>>> print('C:\nouvel_an\photos\tri')
```

```
C:
ouvel_an\photos    ri
>>> print(r'C:\nouvel_an\photos\tri')
C:\nouvel_an\photos\tri
```

Python propose un troisième mode de définition de chaînes de caractères qui commence et se termine par trois guillemets : """..."""". Cette syntaxe permet de conserver la mise en forme d'un bloc de texte.

```
>>> """Les trois quotes :
permettant d'enregistrer la mise en forme.

    => y compris les sauts de ligne, les tabulations..."""
"Les trois quotes :\npermettant d'enregistrer la mise en forme.\n\n
   \t=> y compris les sauts de ligne, les tabulations..."
```

9.2 Manipulation d'un chaîne de caractères

Les chaînes de caractères supportent de nombreuses opérations : tester la présence d'un caractère ou d'un groupe de caractères dans une chaîne, accéder à un caractère à partir de sa position dans la chaîne, compter la longueur de la chaîne, etc. La table 7 liste les principaux opérateurs applicables aux chaînes de caractères tandis que la table 8 recense quelques fonctions s'appliquant sur ces chaînes. La syntaxe générale des fonctions s'appliquant aux chaînes de caractères est : `resultat = chaine.fonction(...)`.

Opérateur	Signification
`+`	Concaténation de chaînes de caractères
`*`	Répétition d'une chaîne de caractères
`in`	Inclusion d'une chaîne dans une autre
`not in`	Non inclusion d'une chaîne dans une autre
`[i]`	Caractère à la i-ième position dans la chaîne
`[i:j]`	Caractères compris entre les i-ième et j-ième positions

TABLE 7 – Opérateurs du type chaîne de caractères

Fonction	Signification
`chaine.count(s)`	Compte le nombre d'occurrence du caractère `s` (ou chaîne de caractères `s`)
`chaine.find(s)`	Retourne l'indice de la première occurrence de `s` dans la chaîne ou -1 si elle n'est pas présente
`chaine.isalpha()`	Retourne `True` si la chaîne est composée exclusivement de lettres
`chaine.isdigit()`	Retourne `True` si la chaîne est composée exclusivement de chiffres
`chaine.replace(old, new)`	Retourne un copie de la chaîne où les caractères `old` ont été remplacés par `new`
`chaine.split(sep)`	Découpe la chaîne en se servant de `sep` comme délimiteur
`chaine.strip(s)`	Supprime les espaces au début et à la fin de chaîne. Supprime aussi `s` s'il est renseigné
`chaine.upper()`	Remplace les minuscules par des majuscules
`chaine.string.lower()`	Remplace les majuscules par des minuscules
`chaine.capitalize()`	Met en majuscule la première lettre
`chaine.endwith(s)`	Retourne vrai si la chaîne se termine par les caractères `s`
`chaine.startwith(s)`	Retourne vrai si la chaîne commence par les caractères `s`

TABLE 8 – Fonctions s'appliquant aux chaînes de caractère

Il est possible de convertir le contenu d'une variable quelconque en chaîne de caractère en utilisant la fonction `str()`.

```
>>> a = 2.1745
>>> b = str(a)
>>> type(b)
<class 'str'>
```

La fonction `len()` retourne la longueur de la chaîne de caractère.

```
>>> a = "ensg"
>>> len(a)
4
```

Dans l'exemple suivant, nous montrons que la concaténation de deux chaînes de caractères entraîne la création d'un nouvel objet de type chaîne de caractère. Puis nous utilisons la fonction `replace()` pour remplacer des caractères. Là encore un nouvel objet est créé. Plus généralement, les chaînes de caractères ne peuvent pas être modifiées, on dit qu'elles sont immuables.

```
>>> txt = "Boum"
>>> id(txt)
44826144
>>> txt = txt + "!"
>>> txt
"Boum!"
>>> id(txt)
44768576          # il ne s'agit plus du même objet. La concaté
   nation crée un nouvel objet.
>>> txt = txt.replace("ou", "a")
>>> txt
"Bam!"
>>> id(txt)
```

```
44766304          # il s'agit bien d'un nouvel objet
```

9.3 Formatage de texte

Python offre plusieurs possibilités de formatage de chaînes de caractères qui sont utiles pour afficher les valeurs contenues dans les variables. Deux syntaxes existent depuis la version 2.6 de Python. La syntaxe originale :

```
>>> "... %c ..." % (valeur)
```

Le format est indiqué après le premier symbole % sous la forme d'une lettre correspondant un type de formatage. La valeur est elle indiquée en fin de ligne après le second symbole %. Il est possible de formater plusieurs chaînes en même temps :

```
>>> "...%c1...%c2..." % (v1, v2)
```

Les valeurs courantes des codes de formatage sont les suivantes :
— s texte simple (résultat de la fonction str(v))
— i nombre entier
— e nombre réel au format exponentiel
— f nombre réel au format décimal
La précision du formatage peut être précisée entre le % et le code. La syntaxe générale est `%[align][largeur][.precision][code]` :

```
>>> n = 3.2547
>>> 'n = %s' %(n)
'n = 3.2547'
>>> 'n = %15s' %(n)
'n =          3.2547'
>>> 'n = %f' %(n)
'n = 3.254700'
>>> 'n = %.2f' %(n)
'n = 3.25'
```

L'autre syntaxe est la suivante :

```
>>> "...{}...".format(valeur)
```

Il est également possible de formater plusieurs chaînes simultanément :

```
>>> "...{}...{}...".format(v1, v2)
```

Et le format de sortie peut aussi être spécifié :

```
>>> "...{:c}...".format(valeur)
```

Les codes de formatage sont identiques à ceux de la première forme présentée.

Remarque : pour la très grande majorité des usages, les deux méthodes sont équivalentes. Excepté pour des usages avancés dépassant le cadre de ce cours, le choix de l'une ou l'autre n'aura pas d'importance.

10 Les séquences

Les séquences ne sont pas un type de données à proprement parler mais plutôt une catégorie qui regroupe un ensembles de structures de données partageant des propriétés communes. Python dispose de trois types "basiques" de séquences : `list`, `tuple` et `range`. Les chaînes de caractères (`string`, introduites partie précédente) constituent également un type particulier de séquences.

Nous commencerons par évoquer les aspects communs à toutes les séquences puis nous détaillerons les spécificités de chacune.

10.1 Généralités sur les séquences

Les séquences partagent en commun le fait d'être **itérables** et **indexables**. Cela signifie que :

— une séquence est capable de retourner les éléments qui la composent les uns à la suite des autres (=itérable) ;
— une séquence est capable de retourner n'importe lequel de ses éléments sans avoir à parcourir l'ensemble de la séquence (=indexable).

Les chaînes de caractères étudiées au chapitre précédent sont un exemple de séquence. Il est en effet possible de récupérer une lettre par son indice dans la chaîne :

```
>>> s = 'abracabra'
>>> s[3]    # 3ème élément de s
'a'
```

Nous en profitons pour remarquer que l'indexation d'une séquence commence à 0 : `s[0] = 'a'`

Il est également possible de parcourir la chaîne caractère par caractère[7] :

```
>>> s = 'abracabra'
>>> iterateur = iter(s) # on se prépare à parcourir s
>>> next(iterateur) # on se positionne sur le premier élément de s
'a'
>>> next(iterateur) # on passe au suivant
'b'
>>> next(iterateur) # on passe au suivant...
'r'
```

Nous distinguerons deux grandes catégories de séquences :

— les **séquences immuables** ne supportent pas la modification de leurs éléments ;
— les **séquences modifiables** (ou mutables) peuvent voir leurs éléments modifiés sans avoir à être redéfinies.

Toutes les séquences supportent les opérations répertoriées dans la table 9. Les chaînes de caractères étant des séquences, les opérations listées dans cette table recoupent celles déjà listées dans la table 7.

7. L'exemple utilise des fonctions `iter()` et `next()` qui ne sont pas à connaître et sur lesquelles nous reviendrons ultérieurement

Opérateur	Signification
`x in s`	x est dans s
`x not in s`	x n'est pas dans s
`s + t`	concaténation de s et t
`s * n`	ajouter n fois s à lui-même
`s[i]`	élément à la i-ème position
`s[i:j]`	sous-séquence de s, de la i-ème à la j-ème position
`s[i:j:k]`	sous-séquence de s, de la i-ème à la j-ème position avec un pas de k
`len(s)`	nombre d'éléments de la séquence s
`min(s)`	plus petit élément de la séquence s (lorsque cela à un sens)
`max(s)`	plus grand élément de la séquence s (lorsque cela à un sens)
`sum(s)`	somme des éléments de la séquence s (lorsque cela à un sens)
`s.index(x, i, j)`	index de la première occurence de x dans s, à partir de l'indice i et jusqu'à j
`s.count(x)`	nombre d'occurence de x dans s

TABLE 9 – Opérateurs sur les séquences

10.2 Les tuples

Un **tuple** est un tableau d'objets qui peuvent être de n'importe quel type. Les tuples, comme les chaînes de caractères ou les numériques, ne sont pas modifiables : ce sont des *séquences immuables*.

Pour créer un tuple, on sépare les différentes valeurs qui vont le composer par des virgules :

```
>>> t = 2016, "ENSG", 'Géomatique'
>>> type(t)
<class 'tuple'>
>>> t
(2016, 'ENSG', 'Géomatique')
```

Pour améliorer la lecture du code, on utilise souvent des parenthèses lors de la définition d'un tuple :

```
>>> t = (2016, "ENSG", 'Géomatique')
```

Les tuples peuvent être imbriqués :

```
>>> t, (1, 2, 3, 4)
((2016, 'ENSG', 'Géomatique'), (1, 2, 3, 4))
```

Les tuples sont toujours représentés entre parenthèses. Comme pour une chaîne de caractère, il est impossible de modifier la valeur d'un élément d'un tuple.

```
>>> t = 2016, 'ENSG', 'Géomatique'
>>> t[0] = 2017
TypeError: 'tuple' object does not support item assignement
```

Tous les opérateurs définis dans le paragraphe de généralités sur les séquences s'appliquent bien entendu aux tuples.

Remarque : pour créer un tuple de zéro ou un élément, on utilise une subtilité de la syntaxe :

```
>>> t = ()      # crée un tuple vide
>>> t
()
>>> t = 1,      # crée un tuple de un élément
>>> t
(1,)
```

Mais attention, utiliser les parenthèses sans virgule ne fonctionne pas :

```
>>> t = (15)      # crée un entier
>>> type(t)
<class 'int'>
```

10.3 Les listes

Une **liste** est similaire à un tuple à la différence près qu'il est possible de modifier, ajouter ou supprimer des éléments d'une liste sans avoir à la redéfinir : les listes sont des *séquences modifiables*.

Tous les opérateurs définit pour les séquences seront utilisable sur les listes (cf. table 9). Mais du fait de leur caractère modifiable, elles supporteront des opérateurs supplémentaires listés dans le table 10 (opérateurs par ailleurs communs à tous les types de séquences modifiables).

Une liste est définit par un ensemble de valeurs entre deux crochets :

```
>>> a = [2016, "ENSG", 'Géomatique']
>>> type(a)
<class 'list'>
>>> a
[2016, 'ENSG', 'Géomatique']
```

Les listes peuvent également être imbriquées :

```
>>> b = [a, [1, 2, 3, 4]]
>>> b
[[2016, 'ENSG', 'Géomatique'], [1, 2, 3, 4]]
```

Pour créer une liste vide, nous utilisons des crochets sans rien à l'intérieur :

```
>>> c = []
>>> type(c)
<class 'list'>
```

Opérateur	Signification
`L[i] = x`	Elément à la position i remplacé par x
`L[i:j] = t`	Partie entre les position i et j remplacé par la liste t
`del(L[i:j])`	Suppression des éléments entre les positions i et j. Equivalent à `L[i:j] = []`
`L[i:j:k] = t`	Partie entre les position i et j avec un pas de k remplacé par la liste t
`del(L[i:j:k])`	Suppression des éléments entre les positions i et j avec un pas de k. Equivalent à `L[i:j:k] = []`
`L += t`	Ajoute t à L
`L *= n`	Ajoute n-1 fois L à elle-même

TABLE 10 – Opérateurs sur les listes

Du fait de leur caractère modifiables, un certains nombres de fonctions viennent avec les listes (cf. table 11).

Fonction	Signification
`l.append(x)`	Ajoute l'élément `x` à la fin de la liste
`l.extend(t)` ou `l += t`	Ajoute les éléments de la liste `t` en fin de liste `l`
`l.insert(i, x)`	Insertion de l'élément x à la position i
`l.remove(x)`	Suppression de tous les éléments x
`l.pop(i)`	Retourne et supprime de la liste l'élément à la position i
`l.clear()`	Vide la liste
`l.reverse()`	Inverse l'ordre des éléments de la liste
`l.copy()`	Crée une copie de la liste
`l.sort()`	Trie les éléments de la liste par ordre croissant [8]

TABLE 11 – Fonctions s'appliquant listes

Remarque : les nombres entiers et flottants sont généralement compatibles et gérés automatiquement par Python (cf. partie). L'une des rares exceptions à cette compatibilité est leur utilisation comme index de listes où seuls les entiers sont acceptés.

```
>>> liste = [0, 1, 2, 3, 4, 5]
>>> liste[5]
5
>>> liste[5.0]
TypeError: list indices must be integers, not float
```

10.4 Les ranges

Les ranges sont des séquences immuables de nombres entiers. Elles sont définies à l'aide d'une borne finale et éventuellement d'une borne initiale et d'un pas. Si la borne initiale est omise, la valeur par défaut sera zéro. Le pas par défaut vaut quant à lui 1.

```
>>> a = range(10)
>>> type(a)
<class 'range'>
```

Pour visualiser le contenu d'un objet de type `range`, on le convertit en liste à l'aide de la fonction `list()` :

```
>>> a = list(range(10))
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> b = list(range(2, 10))
>>> b
[2, 3, 4, 5, 6, 7, 8, 9]
>>> c = list(range(2, 10, 2))
>>> c
[2, 4, 6, 8]
```

Les `range` sont couramment utilisés comme compteur dans les boucles `for` (cf. chapitre 17).

11 Les dictionnaires

Les **dictionnaires** sont des objets semblables aux listes mais plus souple en ce qui concerne le référencement des valeurs. Alors qu'une liste référence les éléments qu'elle contient à l'aide de leur position (un entier compris entre 0 et la taille de la liste), un dictionnaire se contente d'associer l'élément à un autre élément de type immuable (entier, réel, chaîne, tuple). On dit que le dictionnaire *associe une valeur à une clé*.

Les éléments d'un dictionnaire *ne sont pas ordonnés* (il n'est pas possible de les trier) et on y accède uniquement par leurs clés. La structure d'un dictionnaire n'est pas figée, on peut ajouter/modifier/supprimer des éléments sans avoir besoin de recréer le dictionnaire : le dictionnaire est *modifiable*.

La syntaxe de création d'un dictionnaire est la suivante :

```
>>> dico = {cle1: valeur1, cle2: valeur2...}
```

On peut créer un dictionnaire vide :

```
>>> dico = {}
>>> dico = dict()
```

Pour accéder à la valeur d'un élément, la syntaxe est semblable à celle d'une liste, mais l'indice n'est plus uniquement un entier : `dico[cle]` retourne la valeur associé à la clé.

```
>>> dico = {'prenom': 'paul', 'age': 24}
>>> dico['prenom']
'paul'
```

L'ajout d'une nouvel élément est assez aisé :

```
>>> dico[0] = 'rouge'
>>> dico
{0: 'rouge', 'age': 24, 'prenom': 'paul'}
```

Les autres opérations et méthodes pour les dictionnaires sont listées respectivement dans les tables 12 et 13.

Opérateur	Signification
`x in d`	x est dans une des clés de d
`x not in d`	x n'est pas dans les clés de d
`d[c]`	retourne l'élément associé à la clé c
`len(d)`	nombre d'éléments de d
`del(d[c]`	supprime l'élément associé à la clé c

Table 12 – Opérateurs sur les dictionnaires

Fonction	Signification
`d.has_key(x)`	retourne vrai si x est une clé
`d.items()`	retourne une liste des couples (clé, valeur) du dictionnaire
`d.keys()`	retourne une liste des clés du dictionnaire
`d.values()`	retourne une liste des valeurs du dictionnaire
`d.update(dico2)`	dico reçoit le contenu de dico2
`d.clear()`	vide le dictionnaire

Table 13 – Fonctions disponibles pour les dictionnaires

12 Les ensembles

Les ensembles (`set`) sont des collections non ordonnées d'objets. Ils possèdent des propriétés communes avec les listes ou les dictionnaires, et sont également proches sur certains points des types numériques. Les ensembles :

- sont itérables : il est possible de retourner les éléments qui les composent les uns à la suite des autres) ;
- sont modifiables (supportent les ajouts/suppressions d'éléments) ;
- peuvent contenir tous types d'objets ;
- supportent les propriétés mathématiques des ensembles (union, intersection, etc. ; cf. table 14).

Mais les ensembles :

- sont pas indexables : il est impossible de récupérer un élément à partir d'une position ;
- ne contiennent pas d'objets en double.

Opérateur	Signification
`|`	Union
`&`	Intersection
`-`	Différence
`^`	Différence symétrique

Table 14 – Opérateurs des ensembles

On définit un ensemble soit à l'aide du mot clé `set` soit entre acolades :

```
>>> a = set('abracadabra')
>>> type(a)
<class 'set'>
>>> a
{'c', 'r', 'd', 'a', 'b'}  # les doublons ont disparus
>>> b = {'a', 'l', 'a', 'c', 'a', 'z', 'a', 'm'}
>>> type(b)
<class 'set'>
>>> b
{'m', 'c', 'z', 'l', 'a'}
```

Comme les autres collections déjà introduites, les ensembles supportent les opérations `x in set`ou `len(set)` et les fonctions `add(x)`, `remove(x)` ou encore `pop()`.

```
>>> a & b
{'c', 'a'}
>>> a ^ b
{'l', 'b', 'd', 'm', 'z', 'r'}
>>> a.add(14)
>>> a
{'b', 14, 'd', 'c', 'r', 'a'}
```

Si on essaye d'ajouter un élément déjà présent dans un ensemble, Python le détecte et ne fait rien :

```
>>> a.add(14)
>>> a
{'b', 14, 'd', 'c', 'r', 'a'}
```

13 Les fichiers

Dans ce paragraphe, nous nous attarderons sur le dernier type de base qui sera présenté dans ce cours : les fichiers. Nous montrerons comment lire ou écrire dans un fichier.

13.1 Chemins d'accès aux fichiers

Avant de commencer à manipuler les fichiers, précisons qu'il existe deux possibilités pour parcourir l'arborescence d'un système : utiliser des chemins *absolus* ou *relatifs*.

Lorsque l'on utilise des chemins absolus, on décrit le fichier en partant de la racine du disque. Les chemins prennent la forme : `D:\Exercices Python\resultat.txt`.

Avec les chemins relatifs, on tiens compte du répertoire courant, c'est à dire celui depuis lequel l'interpréteur Python est exécuté (si l'on exécute un fichier .py directement, le répertoire courant est celui contenant le programme ; si on utilise l'interpréteur Python, le répertoire courant est celui contenant l'interpréteur). Ainsi, si on est dans le répertoire `D:\Exercice Python` , on appellera la fichier `resultat.txt` de ce répertoire en saisissant tout simplement `resultat.txt`.

Remarque : il arrive parfois que l'on ne sache pas quelle est le répertoire courant. Pour le

connaître on peut utiliser la commande `os.getcwd()`. Il arrivera également qu'il faille changer de répertoire courant. On utilisera alors la commande `os.chdir(repertoire)`.

13.2 Ouverture d'un fichier

Pour ouvrir un fichier, on utilise la fonction `open()`. Elle prend en paramètre :
— le chemin du fichier ;
— le mode d'ouverture :
  — `'r'` pour lire uniquement ;
  — `'w'` pour écrire uniquement (le contenu précédent est écrasé) ;
  — `'a'` pour écrire à la fin uniquement (le contenu précédent est conservé) ;
Il est possible de combiner les modes pour, par exemple, lire et ajouter à la fin.

Lorsque le fichier n'est plus utilisé par le programme, il est nécessaire de le fermer, sans quoi un verrou persiste empêchant son utilisation par une autre personne. On utilise la fonction `close()`.

13.3 Lecture d'un fichier

Plusieurs modes de lecture d'un fichier sont possible. Les plus courants sont :
— la lecture en entier avec la fonction `read()`
— la lecture ligne par ligne avec la fonction `readline()`
D'autres modes permettent la lecture bit par bit ou caractère par caractère.

Un exemple de lecture :

```
>>> fichier = open("fichier.txt", 'r')
>>> contenu = fichier.read()
>>> fichier.close()
>>> print(contenu)
Je suis le contenu d'un fichier texte !
```

13.4 Ecriture dans un fichier

Pour écrire dans un fichier, on utilise la méthode `write(texte)` en lui passant en paramètre la chaîne de caractères à écrire. Elle renvoie le nombre de caractères qui ont été écrits. Cette fonction peut être appelées plusieurs fois pour écrire plusieurs chaînes. Elle fonctionne que le fichier soit ouvert en mode `'w'` ou `'a'`.

```
>>> fichier = open("fichier.txt", 'w')
>>> fichier.write("J'écris pour la première fois dans un fichier
    texte via Python")
62
>>> fichier.close()
```

13.5 L'ouverture avec `with`

L'ouverture et la manipulation de fichiers est une opération source d'erreurs en programmation. De plus, il est facile d'oublier de refermer un fichier après son utilisation, ce qui peut être problématique également.

Pour simplifier l'utilisation des fichiers, Python permet d'utiliser un mot clé `with`. Avec cette forme d'ouverture d'un fichier, la fermeture du fichier est automatique :

```
>>> with open("fichier.txt", 'r') as fichier:
...     contenu = fichier.read()
...     print(contenu)
...
Je suis le contenu d'un fichier texte !
```

Partie III

Syntaxe du langage

Au chapitre précédent, nous avons étudié les structures de base intégrées dans le langage Python. Maintenant que nous maîtrisons les différents types de données, nous pouvons commencer à construire nos premières expressions en Python.

Dans ce chapitre, nous allons commencer par rappeler les règles générales de construction des expressions en Python, puis nous apprendrons à construire les structures élémentaires du langage.

14 Structure du code en Python

En Python, la règle générale est que la fin d'un ligne correspond à la fin d'une instruction[9]. Il n'y a pas de symbole de fin d'instruction comme dans d'autres langages.

L'exemple suivant en Java :

```
instruction1 ;
instruction2 ;
```

S'écrit simplement comme suit en Python :

```
instruction1
instruction2
```

Pour définir un bloc de code, Python utilise l'indentation. De plus la ligne d'en tête du bloc se termine par deux points (:). Ce qui donne :

```
ligne_d_en_tete:
    bloc_d_instructions
```

La fin de l'indentation correspond à la fin du bloc d'instructions. Il n'est pas nécessaire d'ajouter un symbole spécial pour marquer la fin du bloc. On n'utilise pas non plus d'accolades pour délimiter le bloc d'instructions.

Par exemple, un test qui en Java aurait la forme :

```
if (condition) {
    instruction1
    instruction2
}
```

S'écrirait en Python :

9. Il est toutefois possible d'écrire une instruction sur plusieurs lignes s'il s'agit d'une expression entre parenthèse, crochet, accolade. On utilise cette possibilité pour limiter la taille des grandes instructions et faciliter la lecture

```
if condition:
    instruction1
    instruction2
```

Ces quelques règles assurent une certaine homogénéité du code entre différents programmeurs : il n'y pas pas de risque de voir des développeurs écrire plusieurs instructions sur la même ligne, indenter ou pas leurs codes, etc. La figure 12 représente de manière imagée la structure d'un programme Python.

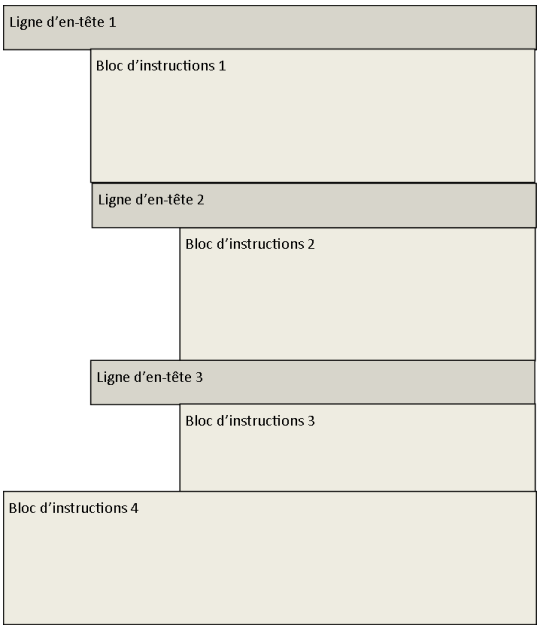

FIGURE 12 – Structure du code en Python

Notons également que les espaces et sauts de ligne sont ignorés par l'interpréteur Python.

```
ligne_d_en_tete:
    instruction1
instruction2
```

Est équivalent à :

```
ligne_d_en_tete:
    instruction1

instruction2
```

Le langage permet enfin d'intégrer des commentaires au sein du code, qui seront ignorés par l'interpréteur. Ceux-ci peuvent prendre deux formes différentes :

— les commentaires simples : ils débutent pas un caractère # et se terminent à la fin de la ligne ;
— les docstrings (documentation string) : ce sont de simples chaînes de caractères (""" ... """) qui sont ignorées par l'interpréteur Python mais peuvent être exploitées par divers outils pour générer de la documentation (nous y reviendrons dans la partie V).

15 Les affectations

Pour affecter une valeur à une variable (ou créer un référence selon la terminologie Python ; cf. partie 6), on utilise le signe = :

```
a = 3
```

On peut créer plusieurs références dans la même instruction :

```
a,b  = 3, 2 # équivaut à a = 3 et b = 2
```

Si l'on veut référencer le même objet dans plusieurs variables, on peut utiliser la syntaxe suivante :

```
a = b = c = 3
```

Qui équivaut à :

```
c = 3
b = c
a = b
```

Python offre également la possibilité d'effectuer des **affectations augmentées** : Les deux instructions de l'exemple suivant sont équivalentes :

```
>>> a = 3
>>> a = a + 3 # forme traditionnelle
>>> a
6
>>> a += 3    # affectation augmentée
9
```

Les affectations augmentées fonctionnent avec tous les types de données et tous les opérateurs standards : `+= -= *= /= //= %=` etc.

16 Les tests

Les tests permettent d'exécuter des instructions différentes selon la valeur d'une condition. La syntaxe Python est la suivante :

```
if condition:
    instruction1
    instruction2
    ...
```

Si des instructions sont à exécuter lorsque la condition n'est pas réalisée, on utilise une clause `else` :

```
>>> x = 1
>>> if x >= 0:
...     print("x positif")
```

```
... else:
...     print("x négatif")
...
x positif
```

S'il est nécessaire d'enchaîner plusieurs tests, on utilise une clause `elif` :

```
>>> x = 5
>>> if x % 2 == 0:
...     print("x multiple de 2")
... elif x % 3 == 0:
...     print("x multiple de 2")
... elif x % 5 == 0:
...     print("x multiple de 5")
... else:
...     print("x n'est multiple ni de 2, ni de 3, ni de 5")
...
x multiple de 5
```

Comme c'est l'indentation qui fait la structure du programme en Python. Le décalage des instructions à exécuter dans chacune des clauses `if`, `elif` et `else` est indispensable pour obtenir un résultat correct.

Il est possible de simplifier l'écriture de certains tests lorsque la valeur est comparée à `None`, un booléen ou zéro.

Valeur	Test	Test simplifié
None	`if v==None:`	`if v:`
None	`if v!=None:`	`if not v:`
Booléen	`if v==True:`	`if v:`
Booléen	`if v==False:`	`if not v:`
Zéro	`if v==0:`	`if not v:`
Zéro	`if v!=0:`	`if v:`

TABLE 15 – Ecriture simplifiée de certains tests

Lorsqu'aucune instruction ne doit être exécutée quand une condition est réalisée, on utilise le mot clé `pass`. Ne rien écrire lorsqu'il ne doit rien se passer provoque un erreur. La présence du mot clé `pass` est obligatoire.

```
if x > 0:
    print("signe positif")
elif x==0:
    pass
else:
    print("signe négatif")
```

Notons également que la présence des parenthèses autour de la condition est optionnel en Python. Ainsi les deux instructions suivantes seront identiques :

```
if x > 3:
    ...
if (x > 3):
    ...
```

17 Les boucles

Les boucles permettent d'exécuter plusieurs fois un même bloc d'instructions.

Le langage Python propose plusieurs types de boucles : la boucle `while` qui permet d'exécuter des instructions tant qu'une condition est réalisée, la boucle `for` qui permet d'exécuter des instructions pour tous les éléments d'une séquence.

17.1 La boucle while

La boucle `while` permet d'exécuter un bloc d'instructions tant qu'une condition est réalisée. C'est la structure itérative la plus générale en Python.

La syntaxe d'une boucle `while` est la suivante :

```
while condition:
    instructions
```

Une clause `else` peut être ajoutée pour exécuter des instructions lorsque toutes les itérations de la boucle sont terminées.

```
while condition:
    instructions1
else:
    instrucitons2 # exécutées à la sortie de la boucle
```

Cela donne par exemple :

```
>>> a = 0
>>> while a < 5:
...     print(a)
...     a += 1
... else:
...     print("Fin de l'itération")
...
0
1
3
4
Fin de l'itération
```

17.2 la boucle for

La boucle `for` est un itérateur générique en Python : elle permet d'exécuter un bloc d'instrcutions pour tous les éléments d'une séquence (liste, tuple, range, chaîne) ou de tout autre objet itérable (dictionnaire par exemple).

La syntaxe générale est la suivante, la clause `else` étant optionnelle :

```
for element in iterable:
    instructions1
else:
    instructions2
```

Les boucles `for` s'utilisent avec n'importe quel type de séquence. Par exemple avec une chaîne de caractères :

```
>>> mot = 'Youpi'
>>> for lettre in mot:
...     print(lettre)
... else:
...     print("Longueur de la chaîne = {}".format(len(mot)))
Y
o
u
p
i
Longueur de la chaîne = 5
```

Un usage fréquent des boucles `for` est le parcours de séquences d'entiers. On utilise alors un `range` :

```
>>> for i in range(0, 10, 2):
...     print(i)
... else:
...     print('Fin')
...
0
2
4
6
8
Fin
```

Si l'on utilise une boucle `for` pour itérer une liste de tuples, l'élément de la boucle peut lui-même être un tuple :

```
>>> liste = [(1, 2), (3, 4), (5, 6)]
>>> for (a, b) in liste:
...     print(a, b)
...
(1, 2)
(3, 4)
(5, 6)
```

Avec des dictionnaires, on peut effectuer une itération sur les clés seules ou sur les clés et les valeurs. Dans les deux cas, on remarquera que le dictionnaire n'étant pas ordonné, la boucle `for` ne parcourt pas nécessairement les éléments dans l'ordre de saisie :

```
>>> dico = {'un': 1, 'deux': 2, 'trois': 3}
>>> for cle in dico:
...     print(cle, '=>', dico[cle])
...
un => 1
trois => 3
deux => 2
```

On utilise la propriété `items()` du dictionnaire pour le transformer en liste de tuples et itérer sur ces éléments :

```
>>> list(dico.items())
[('un', 1), ('trois', 3), ('deux', 2)]
>>> for (cle, valeur) in dico.items():
```

```
...     print(cle, '=>', valeur)
...
un => 1
trois => 3
deux => 2
```

Les fichiers sont également itérables :

```
>>> for ligne in open('test.txt').readlines():
...     print(ligne)
```

Remarque : l'instruction `pass` est utilisable dans tous les blocs d'instruction, y compris les boucles :

```
>>> for i in range(0, 5):
...     pass    # on ne fait rien
... else:
...     print("Fin de l'itération")
...     print("i vaut {}".format(i))
...
Fin de l'itération
i vaut 4
```

17.3 Les sorties de boucle

Le langage Python permet d'utiliser des syntaxes pour sortir d'une boucle avant la fin de l'exécution normale. On parle de **débranchements**. Ils sont de deux types possibles : la sortie définitive de boucle (`break`) et le retour au début des instructions constituant la boucle (`continue`).

Le mot clé `continue`, utilisé dans une boucle, permet de retourner directement au début du bloc d'instructions répété.

```
>>> for i in range(3):
...     print("Début du bloc")
...     continue
...     print("Fin du bloc")
... else:
...     print("Fin de l'itération")
...
Début du bloc
Début du bloc
Début du bloc
Fin de l'itération
```

Remarque : La position de l'instruction `continue` dans la boucle `while` est importante. Mal placée, elle peut provoquer une boucle sans fin. La boucle suivante par exemple ne se termine jamais car le compteur n'est pas incrémenté :

```
>>> a = 0
>>> while a < 5:
...     continue
...     a += 1
...     print(a)
```

Dans une boucle `for`, le problème ne se pose pas. L'utilisation de `continue` entraîne l'incrémentation du compteur. Dans l'exemple suivant, i est bien incrémenté, et l'instruction `a = i` n'est jamais lue car placée après le `continue`.

```
>>> for i in range(3):
...     print(i)
...     continue
...     a = i
...
0
1
2
>>> a
NameError: name 'a' is not defined
```

L'instruction `break` permet quand à elle de sortir immédiatement d'une boucle.

```
>>> a = 0
>>> for i in range(5):
...     print(i)
...     if i == 2:
...         break
...
0
1
2
```

Dans le cas de l'utilisation d'un débranchement `break`, la clause `else` n'est pas lue. Imaginons par exemple que nous cherchions l'indice dans une liste de l'élément ayant pour valeur "Toto" :

```
>>> for element in ['a', 'b', 'Toto', 14, 20]:
...     if element == "Toto":
...         print("Toto trouvé !")
...         break
... else:
...     print("Toto est introuvable")
...
Toto trouvé !
```

Ce code n'affichera "Toto est introuvable" que si l'intégralité de la boucle a été parcourue sans trouver l'élément recherché dans la liste. Si la valeur a été trouvé, le programme affichera "Toto trouvé !".

Remarque : le même résultat pourrait être obtenu sans utiliser de clause `else`. Cela donnerait :

```
trouve = False
for element in sequence:
    if element == "toto":
        print("Toto trouvé !")
        trouve = True
        break
if not trouve:
    print("Toto est introuvable")
```

Mais on remarque alors immédiatement l'intérêt de la forme avec la clause `else` : la première version du programme est bien plus concise.

17.4 Les itérateurs en Python

L'itération est très importante en Python. On itère beaucoup et sur beaucoup d'objets : les listes, les chaînes de caractères, les fichiers, etc. On dit que ces objets sont des **itérables**. En Python, un itérable est un objet qui peut retourner un **itérateur**. Un itérateur d'un objet très simple, qui ne peut faire qu'une seule chose : passer à la valeur suivante. A chaque fois que nous avons utilisé une boucle `for`, nous avons en fait, sans le savoir, créé un itérateur.

Concrètement, on crée un itérateur à l'aide de l'opérateur `iter()`. On notera qu'il est possible de créer plusieurs itérateur à partir du même objet. L'itérateur ne supporte qu'une seule méthode : `next()` permettant de passer à la valeur suivante.

Dans l'exemple suivant on crée deux itérateurs de la liste. Les deux itérateurs peuvent faire référence à des éléments différents de la liste de départ.

```
>>> liste = [1, 2, 3]
>>> y = iter(liste)
>>> type(liste)
<class 'list'>
>>> type(y)
<class 'list_iterator'>
>>> z = iter(liste)
>>> next(y)
1
>>> next(y)
2
>>> next(z)
1
```

Implicitement, lorsque l'on écrit `for x in ('a', 'b', 'c'): ...`, on effectue donc les opérations suivante :

```
>>> iterateur = iter(('a', 'b', 'c'))
>>> x = next(iterateur)
...
```

Remarque : certains objets comme les fichiers sont leurs propres itérateurs. Il n'est pas possible d'appeler la fonction `iter()` sur ces objets (ou cela n'a aucun effet). Une des conséquence importante à cela est que, contrairement à l'exemple précédent avec la liste, il n'est pas possible de parcourir un fichier avec deux itérateurs en même temps (concrètement, il n'est pas possible de lire deux lignes différentes du fichier en même temps).

```
>>> fichier = open(r'D:\test.txt', 'r')
>>> type(fichier)
<class '_io.TextIOWrapper'>
>>> type(iter(fichier))
<class '_io.TextIOWrapper'>
>>> next(fichier)
'Contenu du fichier'
```

17.5 Les listes de compréhension

Il est assez fréquent en programmation de vouloir créer une liste à partir d'un traitement effectué sur une autre liste (par exemple : retourner le carré des entiers d'une liste). Le langage Python met à disposition du programmeur une syntaxe basée sur l'utilisation d'une boucle `for`

pour effectuer cette tâche. On parle de listes de compréhension ou parfois de listes en intention. Il s'agit d'une application

La syntaxe générale est la suivante, la condition étant optionnelle :

```
liste = [ instruction for element in iterable if condition ]
```

Par exemple pour calculer le carré des membres d'une liste :

```
>>> liste = [1, 2, 3, 4]
>>> liste2 = [ x ** 2 for x in liste ]
>>> liste2
[1, 4, 9, 16]
```

Equivaut à :

```
>>> liste = [1, 2, 3, 4]
>>> liste2 = []
>>> for x in liste:
...   liste2.append(x**2)
>>> liste2
[1, 4, 9, 16]
```

Si l'on souhaite ne conserver que le carré des nombres pairs :

```
>>> liste = [1, 2, 3, 4]
>>> liste2 = [ x**2 for x in liste if not x%2 ]
>>> liste2
[4, 16]
```

La syntaxe présenté ici en l'appliquant uniquement à des listes est en fait applicable à n'importe quels types d'objets à la seule condition qu'ils soient itérables (tuples, dictionnaires, ensembles, etc.).

Partie IV

Modules et fonctions

18 Les fonctions

Dans les parties précédentes de ce cours, nous avons utilisé des fonctions à de nombreuses reprises (par exemple pour afficher un résultat dans une console avec `print()`, connaître le type d'une données avec `type()` ou pour récupérer les clés d'un dictionnaires `dico.keys()`). Aussi, si le terme *fonction* nous est familié, il n'a jamais été définit.

Nous nous proposons dans ce paragraphe de définir ce qu'est une fonction et d'apprendre à en créer nous même.

18.1 Définition, syntaxe

Une fonction est un groupe d'instructions qui peut être exécuté plusieurs fois dans un programme. Les fonctions peuvent retourner des résultats et permettent de spécifier des paramètres en entrée de la fonction. Les paramètres en entrée pourront varier d'une exécution à l'autre de la fonction.

De manière plus pragmatique, les fonctions sont une alternative aux recopiages multiples de portions de code. Elles permettent de découper un système complexe en petites partie qui seront plus facile à coder et qui pourront être réutilisées.

L'**appel d'une fonction** dans un programme provoque l'exécution des instructions qui la compose.

Le **corps de la fonction** désigne toutes les instructions qui seront exécutées lorsque la fonction sera appelée dans le programme. Les variables définies à l'intérieur d'une fonction n'existent par défaut qu'à l'intérieur de la fonction.

La création d'une fonction en Python débute par la mot clé `def` suivi du nom de la fonction et de parenthèses. Les instructions qui forment le corps de la fonction débutent à la ligne suivante et se termine lorsque l'indentation retourne à son niveau initial :

```
def ma_fonction():
    instruction_1
    instruction_2
    ...
    instruction_n
```

La convention veut que l'on écrive les noms de fonctions en minuscule en séparant les mots par des underscores : ma_fonction. On remarquera qu'il s'agit de la même convention que pour les noms de variables.

L'appel d'une fonction se fait en écrivant dans le programme appelant le nom de la fonction avec les parenthèses :

```
...
ma_fonction()
...
```

La fonction suivante affiche par exemple "hello world !" dans la console :

```
>>> def hello_world():
...     print("Hello world !")
...
>>> hello_world()
Hello world !
```

Si la fonction retourne un résultat, il est spécifié à l'aide du mot-clé `return`. Le corps d'une fonction peut contenir plusieurs `return`, mais un seul sera lu : dès que ce mot clé est rencontré, l'exécution de la fonction se termine et les instructions suivantes sont ignorées.

```
def ma_fonction():
    instruction_1
    ...
    instruction_n
    return resultat
```

Pour récupérer le résultat, nous l'affectons à une variable de manière classique :

```
x = ma_fonction()
```

Lorsque la fonction retourne plusieurs résultats, ceux-ci sont transmis sous forme de tuple. Il est possible de les récupérer individuellement en effectuant une affectation multiple :

```
def ma_fonction():
    instruction_1
    ...
    instruction_n
    return resultat_1, resultat_2, ..., resultat_n

x_1, x_2, ... x_n = ma_fonction()
```

Les paramètres qui peuvent être spécifiées en entrée de la fonction s'ajoutent entre les parenthèses :

```
def ma_fonction(parametre_1, ..., parametre_n):
    instruction_1
    ...
    instruction_n
    return resultat_1, .. resultat_n
```

Lors de l'appel de la fonction il convient alors de spécifier la valeur des différents paramètres. La fonction suivante retourne par exemple le carré d'un nombre donné en paramètre :

```
>>> def puissance_2(nombre):
...     nombre_2 = nombre ** 2
...     return nombre_2
...
>>> x = puissance_2(4)
>>> x
16
```

Si la fonction attend plusieurs paramètres, lors de l'appel, les paramètres doivent être passés dans le même ordre que dans la définition de la fonction. Une alternative consiste à nommer les

paramètres lors de l'appel. Dans l'exemple ci-dessous, les deux appels de la fonction puissance sont équivalents :

```
>>> def puissance(nombre, facteur):
...     resultat = nombre ** facteur
...     return resultat
...
>>> x = puissance(2, 3)
>>> x
8
>>> y = puissance(facteur=3, nombre=2)
>>> y
8
```

Remarque : Python ne permet pas que plusieurs fonctions portent le même nom dans un programme. Cela reste vrai même si les fonction n'attendent pas les mêmes paramètres. On dit que Python n'autorise pas la *surcharge de fonctions*.

18.2 Portée des variables

On définit la portée d'une variable comme la portion de programme à l'intérieur de laquelle le nom de la variable référence une adresse mémoire. C'est la portion de code à l'intérieur de laquelle la variable "existe" (cf. chapitre 6 pour les notions de référence et d'adresse mémoire).

Par défaut :
- les variables définies à l'intérieur d'une fonction n'existent qu'à l'intérieur de la fonction ;
- les variables définies à l'extérieur d'une fonction sont utilisables à l'intérieur de la fonction si elles ont été déclarée avant la fonction ;
- les variables définies à l'extérieur d'une fonction ne sont pas modifiables dans cette fonction.

On désigne également de **locale** les variables dont la portée se limite à la fonction dans laquelle elles sont définies.

A l'inverse, une variable **globale** est visible dans l'ensemble du programme.

Observons avec quelques exemples qu'une variable définie avant une fonction est lisible dans la fonction mais pas modifiable et que les variable de la fonction ne sont visibles que dans la fonction :

```
>>> x = 1
>>> def test1():
...     y = x + 1
...     print(y)
...
>>> test1()
2
>>> y
NameError: name 'y' is not defined    # y n'existe que dans la
   fonction
>>> def test2():
...     x = 2   # modification de la variable locale
...
>>> test2()
>>> x
1   # la variable globale n'est pas modifiée
```

```
>>> def test3():
...     x += 1
...     print(x)
...
>>> test3()
UnboundLocalError: local variable 'x' referenced before assignment
```

Le langage offre toutefois des solutions pour modifier une variable globale dans une fonction. On pourra par exemple utiliser le mot clé `global` :

```
>>> x = 1
>>> def test():
...     global x
...     x = 2
...     print(x)
...
>>> test()
2
>>> x
2
```

Mais dans la mesure du possible, on évitera d'utiliser des variables globales dans les fonctions car cela revient à faire passer un paramètre caché à la fonction. On préférera écrire explicitement les entrées/sorties de la fonction.

On notera par ailleurs qu'en Python, une fonction est un type d'objet particulier mais dont les règles de déclaration sont identiques à celles des autres variables. Aussi, une fonction peut être déclarée n'importe où dans un programme, y compris à l'intérieur d'une autre fonction. Dans ce cas, la fonction n'est visible que dans le contexte local dans lequel elle a été déclarée.

```
>>> x = 1
>>> def f1():
...     x = 2
...     def f2():
...         print(x)
...     f2()
...
>>> f1()
2
>>> f2()
NameError: name 'f2' is not defined
```

D'un point de vue général, lorsqu'une variable est appelée, Python :
— regarde en premier si la variable a été déclarée localement ;
— si elle n'est pas déclarée localement, il regarde alors dans le contexte d'inclusion (fonction englobant la fonction par exemple) ;
— si la variable n'a toujours pas été trouvé, il la recherche alors dans les variables globales ;
— en dernier recours, il regarde dans les objets intégrés au langage.
On parle de **règle LEGB** pour local, enclosing, global, built-in.

18.3 Paramètres par défaut

Lorsqu'une fonction est utilisée de nombreuses fois dans un programme avec les mêmes paramètres, il peut être intéressant de donner une valeur par défaut aux paramètres pour ne

pas avoir à les spécifier à chaque fois que la fonction est utilisée.

Dans l'exemple précédent, si on utilise souvent la fonction pour calculer la puissance 2 d'un nombre, on écrira :

```
>>> def puissance(nombre, facteur = 2):
...     resultat = nombre ** facteur
...     return resultat
```

Pour appeler la fonction, il suffira alors d'écrire :

```
>>> x = puissance(2)
>>> x
4
```

Et si on souhaite calculer une puissance différente de 2, on ajoutera le 2ème paramètre lors de l'appel :

```
>>> x = puissance(3, 3)
>>> x
27
```

Ajoutons que la position des paramètres lors de la définition de la fonction a de l'importance : les paramètres avec des valeurs par défaut doivent toujours être placés après les autres paramètres :

```
>>> def puissance(nombre, facteur = 2): #syntaxe correcte
>>> def puissance(facteur = 2, nombre): #syntaxe incorrecte
```

18.4 Fonctions et types modifiables

Ce paragraphe vise à attirer l'attention sur quelques comportements particuliers des fonctions lorsque leurs paramètres sont des données de type modifiable.

La fonction ci-dessous, ré-initialise à zero les n premiers terme d'une liste.

```
>>> def init_liste(n, liste):
...     for i in range(n):
...         liste[i] = 0
...
>>> n = 2
>>> liste = [1, 2, 3, 4, 5]
>>> init_liste(n, liste)
>>> n
2
>>> liste
[0, 0, 3, 4, 5]
```

Le paramètre `n` n'a pas été modifié. On dit qu'il a été **passé en valeur**. C'est le cas de toutes les données de type immuable.

La liste a en revanche bien été modifiée par la fonction. Tout se passe comme si ce paramètre avait été **passé en paramètre**. C'est le cas des données de type modifiable.

L'autre remarque que nous soulèverons dans ce paragraphe concerne l'utilisation de valeurs

de types modifiables comme paramètre par défaut d'une fonction. Cela peut introduire des résultats inattendus, comme le montre l'exemple ci-dessous :

```
>>> def ajout_5(liste = [0, 0]):
...     for i in range(len(liste)):
...         liste[i] += 5
...     return liste
...
>>> ajout_5()
[5, 5]
>>> ajout_5()
[10, 10]    # la liste par défaut a été modifiée
>>> ajout_5([0, 0])
[5, 5]
```

L'explication provient du fait que la liste renseignée comme valeur par défaut est toujours la même, mais la valeur des données qu'elle contient change à chaque appel.

18.5 Passage des paramètres

Les paramètres sont les objets passés aux fonctions. Avant d'aller plus loin dans les particularités des paramètres, rappelons les points déjà abordés les concernant :

— passer un paramètre à une fonction revient à affecter une valeur à une variable locale ;
— l'attribution des noms de paramètre dans une fonction n'a pas d'effet en dehors de la fonction ;
— modifier les valeurs d'un paramètre de type modifiable dans une fonction a un impact sur l'objet également en dehors de la fonction.

Nous avons également déjà souligné le fait que l'ordre des paramètres à une importance : ils doivent être passés à la fonction dans l'ordre dans lequel ils apparaissent lors de la définition de la fonction, ou être nommés lors de l'appel.

Le langage autorise la création de fonction acceptant un nombre indéterminé de paramètres. On utilise pour cela la syntaxe `*args`. On parle d'**argument positionnels variables** (ou *arguments dynamiques*).

La fonction suivante calcul la somme des chiffres passés en paramètre, leur nombre étant indéterminé :

```
>>> def somme(*n):
...     s = 0
...     for valeur in n:
...         s += valeur
...     print(s)
...
>>> somme(1, 4, 5, 8)
18
>>> somme()
0
```

Les arguments positionnels variables sont passées à la fonction sous forme de tuples :

```
>>> def func(*args):
...     print(args)
...     print(type(args))
...
```

```
>>> func("d", 4, 5, "a")
("d", 4, 5, "a")
<class 'tuple'>
```

Il est également possible d'utiliser des **arguments nommés variables** (ou *arguments dynamiques nommés*), c'est à dire de créer des fonctions acceptant un nombre indéterminé d'arguments nommés. La syntaxe est alors `**kwargs`. Ce type d'arguments est passé à la fonction sous forme de dictionnaire.

```
>>> def func(**kwargs):
...     print(kwargs)
...
>>> func(a=1, b=2, c="ok")
{'a': 1, 'b': 2, 'c': 'ok'}
```

Les dictionnaires d'arguments nommés variables peuvent par exemple être utilisés dans une fonction de connexion à une base de données, pour laquelle les options peuvent varier d'un appel à l'autre. Ci-dessous, le dictionnaire peut être utilisé pour spécifier une valeur particulière qui remplacera la valeur par défaut :

```
>>> def connect(**options):
...     conn_params = {
...         'host': options.get('host', '127.0.0.1'),
...         'port': options.get('port', 5432),
...         'user': options.get('user', ''),
...         'pwd': options.get('pwd', ''),
...     }
...     return conn_params
...
>>> connect()
>>> connect(host='127.0.0.42')
>>> connect(user='postgres', pwd='postgres')
```

Si l'on souhaite combiner les différents types de paramètres, on respectera l'ordre suivant :
— paramètres positionnels normaux (et obligatoires) ;
— paramètres facultatifs avec valeurs par défaut ;
— paramètres dynamiques ;
— paramètres dynamiques nommés.
Par exemple :

```
>>> def func(a, b, c=7, *args, **kwargs):
...     print('a, b, c = ', a, b, c)
...     print('args = ', args)
...     print('kwargs = ', kwargs)
...
>>> func(1, 2, 3, 4, 5, 6, d=7, e=8)
a, b, c =  1 2 3
args =  (4, 5, 6)
kwargs =  {'d': 7, 'e': 8}
>>> func(1, 2)
a, b, c =  1 2 7
args =  ()
kwargs =  {}
```

Remarque : on notera au passage que, plus généralement en Python, l'opérateur $*$ peut être utilisé pour récupérer un nombre indéterminé de valeurs. Ainsi :

```
>>> a, *b, c = 1, 2, 3, 4, 5
```

```
>>> a
1
>>> b
[2, 3, 4]
>>> c
5
```

Alors que sans le *, l'affectation ne fonctionne pas, car il n'y a pas le même nombre de valeurs que de variables :

```
>>> a, b, c = 1, 2, 3, 4, 5
ValueError: too many values to unpack (expected 3)
```

18.6 Fonctions récursives

Une fonction récursive est une fonction qui s'appelle elle-même.

Python permet d'utiliser ce type de fonction. L'exemple le plus fréquent est celui de la factorielle :

```
def factorielle(n):
    if n == 0:
        return 1
    else:
        return n * factorielle(n-1)
```

Python n'autorise par défaut pas plus de 1000 appels récursifs. Aussi la fonction `factorielle()` précédente ne sera pas capable de calculer les factorielles pour n supérieur à 999 (l'utilisation d'une fonction non récursive est alors nécessaire).

```
>>> factorielle(10)
3628800
>>> factorielle(999)
RuntimeError: maximum recursion depth exceeded in comparison
```

18.7 Fonctions anonymes

Lorsque la valeur de sortie d'une fonction est calculable à l'aide d'une seule expression, Python permet d'employer une syntaxe simplifiée utilisant un nouveau mot clé`lambda`.

```
lambda argument_1, ... argument_n: instruction
```

Par exemple :

```
produit = lambda a, b: a * b
```

Est équivalent à :

```
def produit(a, b):
    return a * b
```

A la différence des fonctions classiques, les fonctions anonymes sont de simples expressions. Elles peuvent directement employées dans une instruction sans avoir besoin d'être définies au préalable.

18.8 Générateurs

Plus tôt dans ce cours (cf. paragraphe 17.4), nous avons mentionné que l'itération est très utilisée en Python. Nous allons introduire dans ce paragraphe un nouveau type de fonction qui permet de retourner un itérateur : les **générateurs**.

La syntaxe sera la même que pour une fonction classique, à l'exception du mot clé `return` qui sera remplacé par le mot clé `yield`.

```
def mon_generateur(arguments):
    instructions
    yield valeurs
```

A la différence d'une fonction classique, un générateur pourra comporter plusieurs `yield`. L'itération portera sur chacune des valeurs retournées. Par exemple :

```
>>> def mon_generateur():
...     yield "valeur 1"
...     yield "valeur 2"
...
>>> for elem in mon_generateur():
...     print(elem)
valeur 1
valeur 2
```

On notera que les générateurs n'apportent pas de nouvelles possibilités : tout ce qui peut être fait avec un générateur peut l'être également avec une fonction classique. Par exemple, le code suivant définit un générateur qui génère le carré d'une suite de nombre :

```
>>> def mon_generateur(liste):
...     for n in liste:
...         yield n**2
...
>>> for elem in mon_generateur([1, 2, 3]):
...     print(elem)
1
4
9
```

La version avec une fonction est également réalisable :

```
>>> def fonct_carre(liste):
...     liste_c = []
...     for n in liste:
...         liste_c.append(n**2)
...     return liste_c
...
>>> for elem in fonct_carre([1, 2, 3]):
...     print(elem)
1
4
9
```

Il existe cependant une différence majeure entre ces deux versions. Dans le cas de la fonction classique, la liste des carrés est calculée en entier puis une itération est lancée sur cette liste. Dans la version avec le générateur, la liste des carrés n'est jamais calculée en entier : le code de la fonction s'exécute à chaque appel du générateur pour calculer la valeur suivante de l'itération.

Pour une liste comportant peu de valeurs, comme dans l'exemple ci-dessus, cela n'a pas trop d'importance. En revanche, s'il s'agissait d'un liste avec des milliers d'enregistrements, la version avec le générateur serait tout de suite beaucoup plus performante car elle utiliserait beaucoup moins d'espace mémoire.

19 Imports et modules

Lors du chapitre précédent, nous avons appris à créer des fonctions en précisant que l'intérêt d'utiliser des fonctions est de pouvoir réutiliser des morceaux de code plusieurs fois. Pour l'instant nous sommes capables d'appeler une fonction créée uniquement depuis le fichier dans lequel elle est définit. Mais avouons qu'il serait très utile de pouvoir utiliser une fonction écrite depuis n'importe quel programme.

Typiquement chaque fichier .py est un **module** Python. Nous verrons dans ce chapitre que Python permet d'**importer** des modules pour réutiliser les fonctions qu'ils définissent dans d'autres modules ou scripts.

19.1 Espaces de noms

Lorsque l'on programme et que l'on souhaite que notre code soit réutilisable, il devient nécessaire de disposer d'une organisation permettant de retrouver facilement les outils et données manipulées. Le langage Python utilise ainsi un système de classement où les objets sont rangés dans des sortes de conteneurs qui permettent d'identifier de manière certaine chaque donnée utilisée.

Nous avons déjà vu qu'un **nom** est associé à chaque variable (paragraphe 6) ou fonction (paragraphe 18) manipulée dans un programme. C'est la partie nommage des outils. En Python, ce qui nous permettra de retrouver un outil sera son **espace de noms** (**namespace**). Il s'agit d'un moyen de regrouper des fonctions et variables à une adresse commune. Les espace de noms classiques seront les "endroits" où les fonctions et variables auront été définies. Par exemple : le coeur du langage pour les structures de données de bases intégrées au langage, le contexte global pour les variables d'un programme, le contexte local pour les variables d'une fonction.

Ce système permet également d'utiliser plusieurs variables qui portent le même nom, à condition qu'elles ne soient pas définies dans le même espace de noms. Nous avons par exemple rencontré ce cas avec les fonctions, où nous avons vu qu'une variable du contexte local de la fonction pouvait porter le même nom qu'une variable globale du programme, et ce sans risque de conflit.

Concrètement, l'identification complète d'une variable se présente sous la forme suivante, en séparant les espaces de noms et les noms avec des points . :

```
espace_de_nom.sous_espace_de_nom.nom_de_variable
```

Remarque : la notion d'espace de noms est intimement liée à la notion de portée des variables introduite dans le chapitre sur les fonctions (paragraphe 18). D'après la documentation de

Python, la portée est une région d'un programme Python où l'espace de noms est directement accessible. Directement accessible signifiant que pour un objet dont la portée n'a pas encore été déterminée, la recherche du nom s'effectuera dans cet espace de noms.

19.2 Architecture d'un programme Python

Un programme Python complet peut être constitué de centaines de milliers de lignes de codes. Pour s'y retrouver, il va être nécessaire de découper ce programme en **modules**. Un module est un fichier .py.

Mais cela ne sera parfois pas suffisant. Nous utiliserons alors une autre structure Python appelée un **package**, qui permet de regrouper des modules entre eux. Un package est un répertoire qui doit contenir un fichier spécial, `__init__.py`, qui ne doit pas nécessairement contenir de code, mais doit impérativement être présent pour que le répertoire soit reconnu comme un package par Python. Un package peut lui-même être découpé en packages (ou sous-packages).

Admettons par exemple qu'un projet soit structuré de la manière suivante :

```
code/
    geometrie/
        __init__.py
        systeme_coordonnees.py
        primitive/
            __init__.py
            point.py
            ligne.py
            surface.py
```

`geometry` et `primitive` sont des packages (`primitive` est un sous-package de `geometrie`). En revanche, `code` n'en est pas un car il ne contient pas de fichier `__init__.py`. `point`, `ligne` et `surface` sont des modules du sous-package `primitive`. `systeme_coordonnees` en est un du package `geometrie`.

19.3 Utiliser un module

Dans les deux paragraphes précédents, nous avons détaillé l'organisation du code dans un projet Python et nous avons introduit la notion d'espace de noms qui nous permet d'identifier de manière certaine des objets distinct portant le même nom.

Il nous reste à étudier le plus important : comment utiliser dans notre programme les données et fonctions définies dans d'autres modules. Pour charger un module externe dans notre programme, on va utiliser le mot clé `import` suivi du nom du module.

```
import module
```

Une fois la commande d'import exécutée, l'interpréteur Python charge dans notre programme l'espace de noms du module. Il devient dès lors possible d'utiliser l'ensemble des fonctions que celui-ci a définit.

Par exemple, après l'import du module `math`, on peut utiliser la fonction racine carré (`sqrt()`)

```
>>> import math
```

```
>>> math.sqrt(4)
2
```

Dans le cas où l'on ne souhaiterait utiliser qu'une seule fonction d'un module externe, une autre syntaxe peut être utilisée :

```
from module import fonction
```

Le processus est similaire mais, au lieu de charger dans notre programme l'espace de noms complet du module, cette forme d'import ne charge dans l'espace de noms global du programme que le seul nom de la fonction à utiliser. On appelle alors directement la fonction :

```
>>> from math import sqrt
>>> sqrt(4)
2
```

La syntaxe d'import d'un package ou d'un module d'un package est très similaire à celle d'un module :

```
import package
from package.sous-package import module
```

En repartant de l'exemple de package du paragraphe précédent, les imports suivant seraient possibles :

```
import geometrie
from geometrie import systeme_coordonnees
from geometrie import primitive
from geometrie.primitive import point
```

Remarque : il est d'usage de placer tous les imports de modules au début d'un programme, même s'il n'interviennent que dans une partie du code.

19.4 Les imports en détail

Pour la plupart des imports standards, lorsqu'il s'agit d'utiliser une librairie de base ou installée proprement (cf. annexe VIII), ce que nous avons expliqué précédemment fonctionnera à merveille. Mais lorsque vous commencerez à vouloir importer vos propres modules, cela ne se passera pas toujours aussi bien.

Dans ce paragraphe, nous allons revenir sur le fonctionnement de la commande `import` pour comprendre comment l'utiliser correctement, même avec nos propres modules.

Lorsque l'on utilise la commande `import` pour charger, par exemple, le module `os`, l'interpréteur Python appelle la fonction `__import__()` pour créer un objet module associée à une variable `os` :

```
>>> os = __import__('os')
>>> type(os)
<class 'module'>
>>> os.getcwd()   # fonction pour récupérer le répertoire courant
'C:\Python34'
```

Après l'import, `os` est une variable comme une autre. On remarquera d'ailleurs qu'il est possible de l'écraser :

```
>>> import os
>>> os = "Je mange un os"
>>> os.getcwd()
AttributeError: 'str' object has no attribute 'getcwd'
```

Si l'on revient à la fonction `__import__()`, elle effectue les actions suivantes :

1. recherche du module ;
2. compilation du module en bytecode si nécessaire ;
3. exécution du module pour construire les objets qu'il définit.

Pour *trouver le module*, Python a besoin de savoir dans quel répertoire le rechercher. En premier, Python regarde dans le répertoire courant (celui depuis lequel la commande python est exécutée). Puis il utilise la variable d'environnement PYTHONPATH. Par défaut, le PYTHONPATH contient les sous-répertoires *site-package* et *dist-package* du répertoire d'installation de Python. Ce sont les répertoires dans lesquels les paquets Python sont installés par les outils standards (cf. annexe VIII).

Remarque : pour personnaliser les répertoires de recherche, il est donc possible de modifier la variable PYTHONPATH. Il est aussi possible d'ajouter des répertoires de recherche en créant un fichier .pth dans le répertoire site-package de l'installation (le fichier .pth, pour path, doit contenir un chemin par ligne). Depuis un programme Python, la variable `sys.path` contient la liste des répertoires de recherche. Il est possible de jouer également sur cette variable.

Lorsque le module a été trouvé, Python regarde s'il a déjà été *compilé en bytecode* (donc si un fichier .pyc est présent ; cf. paragraphe 4) :

— si ce n'est pas le cas, le fichier .py est compilé en .pyc ;
— si un fichier .pyc est déjà présent, Python compare les date des deux fichiers pour déterminer si la version compilée correspond bien à la dernière version du module. Si besoin, le .py est compilé en .pyc

La dernière étape de l'import consiste à *exécuter le bytecode du module*. Le résultat de l'exécution est affecté à la variable portant le nom du module.

On notera que cette dernière étape est bien une exécution du bytecode. Aussi, si le module contient des instructions au niveau du module lui-même, les résultats apparaîtrons lors de l'import (par exemple, si un module contient une unique instructions `print("Mon beau module")`, l'import provoquera l'affichage de "Mon beau module" dans la console).

Remarque 1 : depuis la version 3.2 de Python, les fichier .pyc sont stockées dans un dossier `__pycache__` situé dans le même répertoire que les fichiers source. Les fichiers compilés contiennent, dans leur nom, les informations sur le nom du module et la version de Python utilisée pour la compilation.

Remarque 2 : lors d'un import, Python utilise le premier fichier trouvé correspondant au nom du module recherché, sans tenir compte des extensions. Aussi, `import a` peut fonctionner entre autres :

— avec un fichier source a.py
— avec un bytecode a.pyc uniquement (fichier source absent)
— avec un bytecode optimisé a.pyo (fichier source absent)
— avec un package a sans extension
— etc.

19.5 Quelques modules courants

Nous listons ici quelques uns des modules standards qui peuvent être couramment utilisés. Nous entendons par "module standard" un module qu'il n'est pas nécessaire d'installer (il est présent avec la version par défaut de Python).

Module	Utilisation
os	Manipulation de chemins et fichiers
glob	Lister des fichiers avec une syntaxe Unix
subprocess	Exécuter un programme externe
sys	Informations sur l'environnement
time	Heure courante, fuseaux horaires
datetime	Gestion des dates et durées
collections	Etend les données de listes, dictionnaires, tuples...
itertools	Manipulations sur les itérables
math	Fonctions mathématiques
decimal	Manipulation des chiffres à virgule
fractions	Manipulation des fractions
random	Nombres alétoires
multiprocessing	Paralléliser le code
urllib2	Requêtes sur une URL
doctest	Documentation et tests unitaires depuis les docstrings
unitest	Tests unitaires
sqlite3	Gérer des BDD SQLite3
Tkinter	Interfaces graphiques

TABLE 16 – Quelques modules courants

Pour trouver d'autres modules, on consultera le site https://pypi.python.org qui référence des milliers de modules. L'annexe VIII indique comment installer un module externe sur un ordinateur.

Partie V

Documentation et tests

Cette partie n'est pas en lien direct avec les précédentes, ni les suivantes. Il s'agit d'apporter des techniques permettant au développeur de faciliter l'exploitation de son travail par d'autres personnes.

Même si le langage Python est conçu pour être facilement lisible, des commentaires en langage humain, insérés aux bons endroits dans le code et/ou facilement accessibles aident grandement les autres personnes à comprendre la logique de votre programme. L'écriture de tests permet par ailleurs de s'assurer que les différents modules de l'application répondent bien aux spécifications initiales, et ce même après plusieurs mises à jour du code.

L'objectif de cette partie et de vous présenter une méthode de rédaction de la documentation d'un programme Python, ainsi que d'écriture de tests unitaires. L'une des grandes forces de Python est la facilité d'écriture et d'exploitation de documentation du code via le mécanisme des docstrings. Nous utiliserons au cours de cette partie le module `doctests` qui exploite ce mécanisme.

D'autres modules Python d'aide à la rédaction de documentation et/ou de tests unitaires existent et seront plus adaptés aux gros projets informatiques (pytest, unitest, sphinx par exemple).

20 Documentation

20.1 Rédaction de la documentation

Pour rédiger la documentation de nos programme, nous utiliserons un mécanisme très pratique en Python : les **docstring**. Il s'agit de chaînes de caractères qui ne sont assignées à aucune variable et sont placées à des endroits spécifiques du programme. Il est important de faire la distinction entre les doctrings et les commentaires (commençant après les caractères `#`) : les commentaires sont complètement ignorés par l'interpréteur Python tandis que les docstrings sont chargées et gardées en mémoire.

Pour une fonction, la docstring est placée juste sous la signature de la fonction. Exemple d'une fonction sans docstring :

```
def ma_fonction(arguments):
    return quelquechose
```

La même fonction avec une docstring :

```
def ma_fonction(arguments):
  """ma docstring"""
  return quelquechose
```

La docstring d'un module se place en haut du fichier. Cela doit être la première expression du module qui n'est pas un commentaire.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
    Description du contenu du module, de ce qu'il fait, à quoi il
       sert...
"""

import plein_de_choses

def ma_fonction(arguments):
    """ma docstring"""
    return quelquechose
```

De même, pour les classes que nous définirons dans la prochaine partie (partie VI), la docstring se place juste sous la signature de la classe.

Il est d'usage de respecter quelques conventions pour la rédaction des docstrings. En premier, pour des raisons de lisibilité, on limitera la taille d'une ligne à 80 caractères. La forme générale de la docstring sera la suivante :

```
"""
    Résumé sur un seule ligne.

    Contenu détaillé
    sur
    plusieurs lignes.
"""
```

Pour faciliter la lecture des docstrings longues (il n'est pas rare d'avoir des docstrings plus longues que les fonctions qu'elles décrivent), on utilise des = et - pour souligner les titres et sous-titres.

Des balises sont également disponibles pour préciser les entrées et sorties d'une fonction :

```
"""
    :param arg1: description de l'argument 1
    :param arg2: description de l'argument 2
    :type arg1: type de l'argument 1
    :type arg1: type de l'argument 2
    :return: description de la valeur de retour
    :rtype: type de la valeur de retour
"""
```

Un exemple reprenant ce que l'on à introduit plus haut :

```
def addition(a, b):
    """
        Addition de deux nombres.

        Cette méthode permet d'additionner deux nombres réels et
           retourne
        également un nombre réel. Elle est équivalente à l'opé
           rateur '+'.

        :param a: premier nombre à additionner
```

```
        :param b: deuxième nombre à additionner
        :type a: float
        :type b: float
        :return: résultat de l'addition
        :rtype: float
    """
    return a + b
```

20.2 Accès à la documentation

L'écriture de la documentation de nos programmes est essentiel car nous l'utilisons pour savoir comment ils fonctionnent. Vous avez d'ailleurs peut-être déjà recherché de l'aide sur une fonction ou un module que vous vouliez utiliser. Les moyens les plus utiles pour rechercher une information sur un programme sont les suivants :

— la fonction `dir` : elle retourne une liste de l'ensemble des attributs de l'objet passé en paramètre (une fonction, un module, etc.) :

```
>>> dir(math)
['__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
 'acosh',
 'asin',

...

 'tanh',
 'trunc']
```

— la fonction `help()` qui affiche l'aide spécifique à une fonction ou à l'ensemble des fonctions d'un module. La fonction `help()` affiche en fait le contenu de la doctring de l'objet passé en paramètre :

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

cos(...)
    cos(x)

    Return the cosine of x (measured in radians).
```

— l'attribut `__doc__` : qui affiche les mêmes informations que la fonction `help()`, mais est accessible partout dans le programme :

```
>>> math.__doc__
'This module is always available.  It provides access to the\n
    '
'mathematical functions defined by the C standard.'
>>> math.cos.__doc__
'cos(x)\n\nReturn the cosine of x (measured in radians).'
```

21 Tests unitaires

21.1 Les différents types de tests

Il existe différents types de tests en programmation. Chaque type de tests répond à un objectif précis. En fonction du projet, certaines catégories de tests ne sont pas toujours nécessaires.

Les **tests unitaires** permettent de tester de petites portions de code (une fonction, une classe, un module) indépendamment du reste du programme. Ils sont codés par le développeur afin de cadrer les évolutions dans le code. Ils permettent en effet de s'assurer que l'exécution d'un bout de code fournit toujours le même résultat.

Les **tests fonctionnels** permettent de valider les fonctionnalités d'une application d'un point de vu utilisateur. Une application peut répondre parfaitement aux tests unitaires (toutes les briques élémentaires sont correctes) mais n'être pas conforme d'un point de vu fonctionnel (mauvais enchaînement des briques élémentaires).

Les **tests de performances** sont constitués de tout un ensemble de tests liés à la performance de l'application : *test de charge* où l'on simule un nombre important d'utilisateurs, *tests de performance* où l'on mesure les temps de réponse de l'application face à différents niveaux de charge, *tests aux limites* où l'on simule une activité bien supérieure à l'activité attendue.

Nous nous intéresserons dans ce cours uniquement à la mise en oeuvre de tests unitaires qui est de la responsabilité du développeur.

21.2 Des tests unitaires grace aux docstrings

Pour écrire les tests unitaires en Python, nous utiliserons une propriété des docstrings : les **doctests**. Il s'agit de la possibilité d'écrire des tests à l'intérieur même des docstrings.

Concrètement, nous utiliserons un triple chevron >>> pour indiquer que nous écrivons un test. La ligne suivante, avec la même indentation, est considérée comme le résultat du test.

Cela donne par exemple :

```
def addition(a, b):
    """
        Addition de deux nombres.

        :Example:

        >>> addition(2, 3)
        5

    """
    return a + b
```

Ensuite pour vérifer que le test unitaire est respecté, nous utiliserons le module `doctest` et la fonction `testmod()` :

```
import doctest
```

```
doctest.testmod(nom_module)    # on passe le nom du module en
    argument
```

S'il n'y a pas d'échec, l'interpréteur nous affiche simplement le nombre de tests réussit :

```
TestResults(failed=0, attempted=1)
```

En revanche, si un test n'est pas réussit, l'interpréteur nous indique l'appel ayant causé l'erreur. Par exemple, si on ajoute le test :

```
>>> addition(0, 1)
0
```

On obtient :

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

File "__main__", line 4, in __main__.addition
Failed example:
    addition(0, 1)
Expected:
    0
Got:
    1
**********************************************************************

1 items had failures:
   1 of   2 in __main__.addition
***Test Failed*** 1 failures.

TestResults(failed=1, attempted=2)
```

Attention : doctest compare les résultats qui s'affiche est pas les valeurs. Ainsi si on teste :

```
>>> str(addition(1, 1))
2
```

Le test sera un échec car le résultat de `str(addition(1, 1))` est `'2'`. Il aurait fallut écrire :

```
>>> str(addition(1, 1))
'2'
```

On utilisera donc les doctests avec vigilence, surtout lorsqu'il s'agit de tester résultats composés de textes longs ou de listes avec beaucoup de valeurs (l'option ELLIPSIS du module peut alors être utile). De même, avec les dictionnaires, l'ordre des éléments en sortie n'est pas garantie et il faudra donc être très prudent.

Partie VI

Programmation orientée objet

Ce chapitre aborde un type de programmation différent de ce qui a été fait dans ce cours jusqu'à présent : la programmation orientée objet. Après avoir présenté brièvement ce type de programmation, nous montrerons comment le mettre en oeuvre en Python.

Avant d'aborder cette partie, il convient d'être à l'aise avec ce qui a été présenté jusque là : les structures de données de bases avec leurs opérations et méthodes, les fonctions, la portée des variables et le passage de paramètres, l'itération dans ses moindres détails (listes en intention, générateurs).

22 Introduction

La programmation orientée objet est un type de programmation. Il existe en effet plusieurs types de programmation qui diffèrent par la méthodologie générale employée pour résoudre les problèmes posés. On parle de **paradigme** de programmation.

Chaque style de programmation apporte ses spécificités qui le rende plus adapté pour traiter certains problèmes. Mais tous les paradigmes permettent de résoudre les mêmes problèmes.

Un langage de programmation peut par ailleurs supporter différents styles de programmation. Ainsi le langage Python permet de mettre en oeuvre de la programmation dite procédurale et/ou orientée objet.

La **programmation procédurale** est le type de programmation que nous avons employé naturellement depuis le début de ce cours. Elle consiste à découper un problème en sous-problèmes et à résoudre individuellement chacun de ces sous-problèmes (éventuellement en le redécoupant jusqu'à arriver à des briques maîtrisables). La programmation procédurale est ainsi centré sur la description des traitements à mettre en oeuvre pour résoudre un problème.

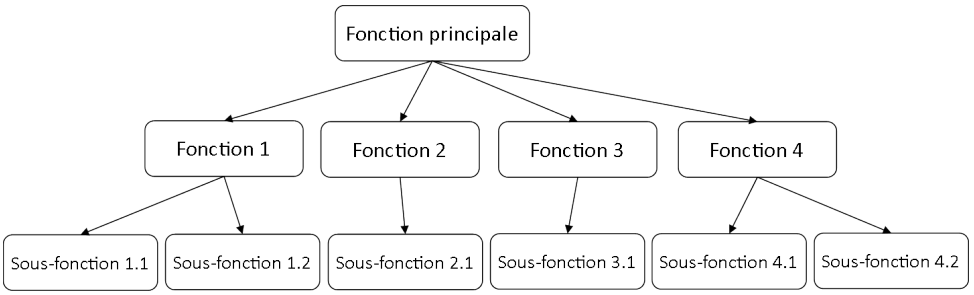

FIGURE 13 – Représentation de l'approche fonctionnelle

La **programmation orientés objet** est un style de programmation qui vise à identifier les structures impliquées dans un problème et à les faire interagir pour le résoudre. Elle considère

un système comme un ensemble d'objets autonomes possédant des caractéristiques propres et un comportement. Ce type de programmation cherche à rassembler au même endroit les données (ou structures de données) et fonctions qui sont faites pour aller ensemble.

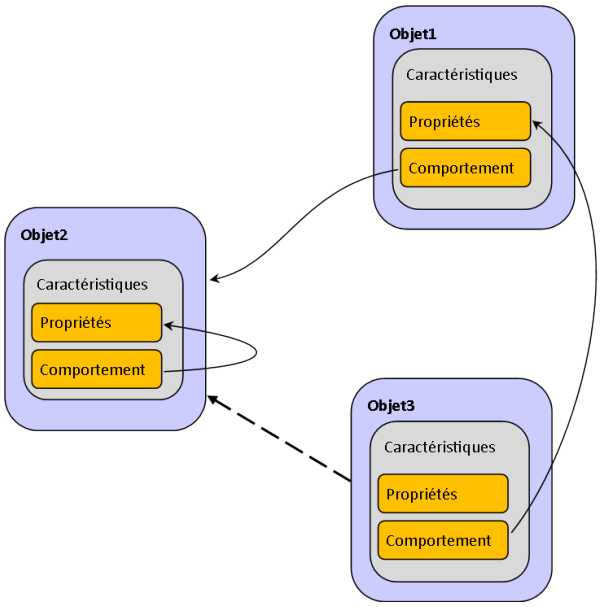

FIGURE 14 – Représentation de l'approche orientée objet

Lors des parties précédentes de ce cours, nous avons appris à utiliser des structures de données plus ou moins complexes (nombres, listes, dictionnaires, etc.). Nous avons stocké des valeurs dans ces structures, puis manipulé ces structures à l'aide d'opérateurs ou de de fonctions. En sommes, les structures manipulées possèdent des attributs (pour y stocker des valeurs) et des propriétés (les opérateurs et fonctions utilisés). Nous avons donc, sans le formaliser explicitement, manipulé des objets. Il nous reste maintenant à voir comment définir nos propres objets.

23 Objets, classes

23.1 Définitions

Un **objet** est une entité autonome (entité du monde physique, concept, idée) décrit par une collection de propriétés et de traitements associés. L'objet est au coeur de la modélisation orientée objet.

En Python, tout est objet : une chaîne de caractères, une liste, un dictionaire, etc. Pour créer un nouvel objet, il faut dire à l'ordinateur à quoi devra ressembler cet objet. Il s'agit concrètement de choisir un modèle suivant lequel créer l'objet. En programmation orientée objet, ce modèle d'objet est appelé une **classe**.

La syntaxe Python de création d'une classe utilise le mot clé `class` :

```
>>> class ModeleDObjet(object):
```

```
...     pass
```

Le (object) est obligatoire, nous y reviendrons ultérieurement. Ensuite, on crée l'objet à partir de ce modèle en appellant la classe :

```
>>> mon_objet = ModeleDObjet()
```

Remarque : par convention, on nomme les classes avec la première lettre de chaque mot en majuscule (NomDeLaClasse), et les objets avec des minuscules et des underscores (nom_de_l_objet).

On dit que l'objet est une **instance** de classe. Il est possible de créer autant d'instances que l'on veut à partir d'une même classe (la fonction id(), déjà utilisée dans ce cours, retourne l'adresse mémoire d'un objet : elle permet d'identifier de manière unique un objet) :

```
>>> id(ModeleDObjet())
49253616
>>> id(ModeleDObjet())
48768592
>>> id(ModeleDObjet())
49253584
```

23.2 Méthodes

La classe doit donc permettre de décrire à quoi va ressembler un objet :
— les attributs qui le caractérisent ;
— les opérations qui définissent son comportement.
Les opérations qui définissent le comportement d'un objet sont appelées des **méthodes**. Il s'agit simplement de fonctions définies à l'intéreur de la classe. On utilise donc le mot clé def :

```
>>> class ModeleDObjet(object):
...     def une_methode(param):
...         print("Je passe dans la méthode")
...
>>> mon_objet = ModeleDObjet()
>>> mon_objet.une_methode()
Je passe dans la méthode
```

Une méthode n'existe pas en dehors de la classe :

```
>>> une_methode()
NameError: name 'une_methode' is not defined
```

Une particularité de Python fait que, lors de l'appel d'une méthode depuis un objet, cet objet est automatiquement passé en premier paramètre de la méthode : tout se passe comme si l'interpréteur transformait l'écriture mon_objet.une_methode() en une_methode(mon_objet).

La présence de cet argument dans la définition de la méthode est obligatoire. Son absence provoque une erreur lors de l'appel de la fonction :

```
>>> class ModeleDObjet(object):
...     def une_autre_methode():
...         print("Je passe dans l'autre méthode")
...
>>> mon_objet = ModeleDObjet()
>>> mon_objet.une_autre_methode()
TypeError: une_autre_methode() takes no arguments (1 given)
```

La convention veut que l'on appelle cet argument `self` pour faire référence à l'objet lui-même. Il s'agit d'une convention très forte en Python et ne pas la respecter risque de rendre votre programme incompréhensible pour un autre développeur.

23.3 Attributs

De même que pour les méthodes, qui ne sont que des fonctions attachées aux objets, les attributs sont des variables attachées aux objets. Comme pour les méthodes, ils sont accessibles en faisant précéder le nom de l'attribut du nom de l'objet :

```
>>> mon_objet.un_attribut = 4
>>> mon_objet.un_attribut
4
>>> un_attribut   # l'attribut n'existe pas en dehors de l'objet
NameError: name 'un_attribut' is not defined
```

Deux objets peuvent avoir des valeurs d'attributs différentes :

```
>>> mon_objet = ModeleDObjet()
>>> mon_objet.un_attribut = 4
>>> ton_objet = ModeleDObjet()
>>> ton_objet.un_attribut = "toto"
>>> mon_objet.un_attribut
4
>>> ton_objet.un_attribut
"toto"
```

Il est également possible d'accéder à un attribut depuis l'intérieur d'un objet, pour lui affecter une valeur, le modifier, l'afficher... En respectant la convention, on utilisera alors `self` pour faire référence à l'objet lui-même :

```
>>> class ModeleDObjet(object):
...     def affecter_attribut(self, valeur):
...         self.un_attribut = valeur
...     def modifier_attribut(self):
...         self.un_attribut += 2
...     def afficher_attribut(self):
...         print(self.un_attribut)
...
>>> mon_objet = ModeleDObjet()
>>> mon_objet.affecter_attribut(4)  # on passe la valeur 4 en param
   ètre de la fonction
>>> mon_objet.afficher_attribut()
4
>>> mon_objet.modifier_attribut()
>>> mon_objet.afficher_attribut
6
```

23.4 Initialisation d'un objet

Il est assez fréquent de vouloir créer des objets avec un état de départ contrôlé. Mettre à jour l'ensemble des attributs de tous les objets ne serait pas des plus pratiques. Aussi, afin de faciliter cette tache, Python propose une méthode spéciale pour initialiser l'état d'un objet : la

méthode `__init__()`. Lorsqu'elle est présente, cette méthode est exécutée automatiquement à la création d'un objet.

```
>>> class ModeleDObjet(object):
...     def __init__(self):
...         self.un_attribut = "valeur initiale"
...         print("Bonjour")
...
>>> mon_objet = ModeleDObjet()
"Bonjour"
>>> mon_objet.un_attribut
"valeur initiale"
```

Il est possible de passer des paramètres à la méthode `__init__()` lors de l'instanciation pour préciser certaines conditions initiales :

```
>>> class ModeleDObjet(object):
...     def __init__(self, valeur):
...         self.un_attribut = valeur
...         print("Bonjour")
...
>>> mon_objet = ModeleDObjet("je suis vivant")
"Bonjour"
>>> mon_objet.un_attribut
"je suis vivant"
```

24 Principes de l'orienté objet en Python

Faire de la programmation orientée objet suppose manipuler des classes et des objets pour résoudre les problèmes. Pour pouvoir appliquer tous les concepts de ce type de programmation, cela nécessite aussi de maîtriser trois grands principes : l'**encapsulation**, l'**héritage** et le **polymorphisme**. Nous nous attarderons dans un premier temps sur la mise en oeuvre en Python des deux principes que sont l'héritage et le polymorphisme. Puis nous aborderons en détail le concept d'encapsulation en Python.

24.1 Héritage et polymorphisme

L'héritage c'est la capacité de pouvoir créer une classe dérivée à partir d'une classe existante, les attributs et méthodes de la classe existante étant automatiquement transmis à la classe dérivée. On parle également de classes mères et filles (pour respectivement classes existantes et dérivées). L'héritage est un principe qui encourage la réutilisation du code.

En Python, l'héritage est indiqué dans la signature de la classe, en remplaçant le `object` par le nom de la classe mère :

```
>>> class ClasseMere(object):
...     pass
...
>>> class ClasseFille(ClasseMere):  # on indique que ClasseFille hé
    rite de ClasseMere
```

```
...     pass
...
```

Les méthodes définies dans la classe mère sont directement utilisables depuis la classe fille. Il n'y a pas besoin de les redéfinir.

```
>>> class Bavard(object):
...     def blablabla(self):
...         print("Je ne m'arrête jamais de parler...")
...
>>> class Copieur(Bavard):
...     pass
...
>>> Copieur().blablabla()
Je ne m'arrête jamais de parler...
```

Remarque : la syntaxe générale `class ModeleDObjet(object)` nous indique donc que toutes les classes héritent d'une classe `object`. Il s'agit d'un type de base en Python, au même titre que les entiers, les chaînes de caractères ou les dictionnaires.

Parfois, on souhaite hériter seulement d'une partie d'une classe, et redéfinir certaines méthodes ou attributs. Il suffira pour cela de ré-écrire la méthode dans la classe fille.

```
>>> class Professeur(object):
...     def methode(self):
...         print("Il faut apprendre ses leçons")
...
>>> class BonEleve(Professeur):
...     pass
...
>>> class MauvaisEleve(Professeur):
...     def methode(self):
...         print("Je n'apprend pas mes leçons")
...
>>> BonEleve().methode()
Il faut apprendre ses leçons
>>> MauvaisEleve().methode()
Je n'apprend pas mes leçons
```

Dans notre exemple précédent, BonEleve et MauvaisEleve ont les mêmes attributs et méthodes, même si elles ne font pas la même chose. On dit que les deux classes partagent une **interface** commune. La capacité à redéfinir le comportement des méthodes est le **polymorphisme**.

Avant de poursuivre, on notera que l'héritage concerne toutes les méthodes de la classe mère, y compris le constructeur (`__init__()`) et méthodes spéciales (cf. paragraphe 25). Il est possible de redéfinir le comportement de toutes ces méthodes également.

```
>>> class Professeur(object):
...     def methode(self):
...         print("Il faut apprendre ses leçons")
...
>>> class BonEleve(Professeur):
...     pass
...
>>> class MauvaisEleve(Professeur):
...     def methode(self):
...         print("Je n'apprend pas mes leçons")
```

```
...
>>> BonEleve().methode()
Il faut apprendre ses leçons
>>> MauvaisEleve().methode()
Je n'apprend pas mes leçons
```

Nous avons appris à redéfinir une méthode héritée en la ré-écrivant complètement. Parfois le comportement de la classe fille ne différera que très légèrement de celui de la classe mère. Dans ce cas, il serait opportun de pouvoir récupérer le code du parent pour l'appeler dans la méthode de l'enfant en y ajoutant les précisions nécessaires. Cela est possible en Python à l'aide de la fonction `super()` :

```
>>> class Article(object):
...     def __init__(self, prix):
...         self.prix = prix
...
>>> class ArticleEnPromotion(Article):
...     def __init__(self, prix, rabais):
...         self.prix = super(ArticleEnPromotion, self).__init__(
   prix) * rabais
...
>>> Article(10).prix
10
>>> ArticleEnPromotion(10, 0.8).prix
8
```

24.2 Classe abstraite

Une classe abstraite est une classe qui ne peut pas être instanciée. Une telle classe est utile uniquement pour définir par héritage d'autres classes.

Il n'est pas possible, en raison du typage dynamique en Python, de définir réellement une classe abstraite. Certaines astuces consistant à lever une exception dans le constructeur de la classe abstraite permettent de s'affranchir de cette limitation du langage.

24.3 Héritage multiple

Il arrive parfois que l'on veuille construire une classe en réutilisant les attributs et méthodes définies dans plusieurs autres classes. Il s'agit d'un **héritage multiple** : une classe fille hérite de plusieurs classes mères. Il faut dans ce cas appeler explicitement le constructeur des classes mères.

```
>>> class Carre(object):
...     __init__(self, cote):
...         self.cote = cote
>>> class Couleur(object):
...     __init__(self, r, v, b):
...         self.rvb = (r, v, b)
>>> class CarreColore(Carre, Couleur):
...     def __init__(self, cote, r, v, b):
...         Carre.__init__(self, cote)
...         Couleur.__init__(self, r, v, b)
...
```

```
>>> CarreColore(4, 255, 0, 0).rvb
(255, 0, 0)
>>> CarreColore(4, 255, 0, 0).cote
4
```

24.4 Encapsulation

L'encapsulation est le fait de rassembler les données et méthodes au sein d'une structure en cachant les détails de l'implémentation. Ce concept permet de modifier les méthodes ou d'ajouter des attributs sans avoir à redéfinir complètement les objets.

Contrairement à d'autres langages comme le Java ou le C++, la structure du langage Python ne force pas l'encapsulation des données. Par défaut, il est toujours possible d'avoir accès aux attributs et méthodes d'une classe, même depuis l'extérieur de celle-ci (c'est ce que nous avons fait depuis le début de ce chapitre). En revanche, les programmeurs Python ont adopté un consensus sur l'accès aux attributs qui consiste :

— à préfixer d'un _ les attributs privés : `_mon_attribut_prive`

— à écrire normalement les attributs publics : `mon_attribut_public`

Utiliser un attribut privé ne signifie pas qu'il ne sera jamais possible d'y accéder depuis l'extérieur de l'objet. Lorsque cela sera nécessaire le développeur utilisera des **accesseurs** et **mutateurs**. Ces noms désignent juste des fonctions qui permettrons respectivement de lire et modifier la valeur de l'attribut d'un objet. Naturellement, en fonction des besoins, on écrira l'une ou l'autre (ou les deux ou aucune) de ces deux fonctions.

Pour écrire un accesseur ou un mutateur, on utilise des **décorateurs** de fonction. Un décorateur est une inscription que l'on place juste avant la signature de la fonction et qui permet de préciser l'utilisation de la fonction. Ils prennent la forme suivante :

```
@decorateur
def ma_fonction(args):
  instructions
```

Pour créer un accesseur, le décorateur sera `@property`. Cela transforme la méthode en propriété. Il n'est alors plus nécessaire d'utiliser les parenthèses pour appeler la méthode.

```
>>> class Carre(object):
...     def __init__(self, cote):
...         self._cote = cote
...
...     @property
...     def cote(self):
...         return self._cote
...
>>> Carre(4).cote
4
```

On utilisera également le décorateur `@property` lorsque l'on souhaitera créer des attributs dérivés.

```
>>> class Carre(object):
...     def __init__(self, cote):
...         self._cote = cote
...
...     @property
```

```
...     def surface(self):
...         return self._cote ** 2
...
>>> Carre(4).surface
16
```

Pour un mutateur, il prendra la valeur `@nom.setter` où nom est le nom de l'attribut (celui visible depuis l'extérieur, donc le nom de la propriété définit grâce à l'accesseur).

```
>>> class Carre(object):
...     def __init__(self, cote):
...         self._cote = cote
...
...     @property
...     def cote(self):
...         return self._cote
...
...     @cote.setter
...     def cote(self, valeur):
...         self._cote = valeur
...
>>> mon_carre = Carre(4)
>>> mon_carre.cote
4
>>> mon_carre.cote = 2
>>> mon_carre.cote
2
```

Nous utiliserons également un dernier décorateur nous permettra de supprimer un attribut : `@nom.deleter`.

L'utilisation d'accesseurs et mutateurs (on parle aussi de getter / setter), au delà de respecter la convention ayant cours dans le milieu des développeurs Python, nous permet de contrôler le type et éventuellement d'effectuer des opérations sur les valeurs affectées dans les attributs. En effet le typage dynamique fort en Python ne nous permet pas de verrouiller le type d'un attribut.

Voici un exemple complet :

```
>>> class Personne(object):
...     def __init__(self, nom):
...         self._nom = nom
...
...     @property
...     def nom(self):
...         return self._nom
...
...     @nom.setter
...     def nom(self, valeur):
...         if not isinstance(valeur, str):
...             raise ValueError("La valeur attendue pour le nom
   doit être un string")
...         self._nom = valeur.capitalize()
...
...     @nom.deleter
...     def nom(self):
...         del(self._nom)
...
>>> toto = Personne("Papa")
```

```
>>> toto.nom
Papa
>>> toto.nom = 45
ValueError: La valeur attendue pour le nom doit être un string
>>> toto.nom = "maman"
>>> toto.nom
Maman     # remarquez la majuscule automatique sur la 1ère lettre
>>> del(toto.nom)
>>> toto.nom
AttributeError: 'Personne' object has no attribute '_nom'
```

25 Concepts avancés avec les classes

25.1 Méthodes spéciales

On appelle **méthode spéciale** une méthode qui est exécutée automatiquement lorsque des conditions particulières sont observées. C'était par exemple le cas du constructeur de l'objet `__init__()` qui est lancé automatiquement à la création d'un objet. Les méthodes spéciales sont préfixés et suffixés avec deux underscores.

Nous présentons ici quelques unes des méthodes que nous pouvons rencontrer :

— la méthode `__str__()` : elle est déclanchée lorsque la commande `print` est appelée sur un objet. Elle permet de personnaliser ce qui est affiché.

```
>>> class ModeleDObjetStandard(object):
...     pass
>>> print(ModeleDObjet())
<__main__.ModeleDObjetStandard object at 0x059F0190>
>>> class ModeleDObjetPersonnalise(object):
...     def __str__(self):
...         return "Objet personnalisé (identifiant {})".
   format(id(self))
>>> print(ModeleDObjetPersonnalise())
Objet personnalisé (identifiant 6912457)
```

— la méthode `__repr__()` est appelée lorsque l'on tape directement le nom d'un objet. Elle permet de remplacer l'affichage par défaut qui n'est pas très parlant (du type `<__main__.ModeleDObjet object at 0x0342C390>`). Elle est aussi appelé lorsque l'on affiche une liste.

— méthodes de comparaison : `__eq__()`, `__lt__()`, `__le__()`,`__gt__()`, `__ge__()`, `__ne__()` pour respectivement $=$, $<$, $<=$, $>$, $>=$ et $!=$
Dans l'exemple suivant, on définit une classe Etudiant avec deux attributs : une note en informatique et une note en géodésie. On utilise ensuite la méthode `__le__()` permettant de comparer les étudiants : un étudiant est supérieur à un autre s'il possède une meilleure note en informatique).

```
>>> class Etudiant(object):
...     def __init__(self, informatique, geodesie):
...         self.note_info = informatique
...         self.note_geodes = geodesie
...
...     def __le__(self, other):
...         if self.note_info > other.note_info:
```

```
...             return True
...         else:
...             return False
>>> albert = Etudiant(15, 0)
>>> alfred = Edutiant(12, 20)
>>> albert > alfred
True
```

— les opérateur mathématiques ($+$, $-$, $*$, $**$, etc.) sont également couvert par des méthodes spéciales : `__add__()`, `__sub__()`, `__mul__()`, `__truediv__()`, etc. (cf.table 17).

Opérateur	Méthode
+	__add__
-	__sub__
*	__mul__
**	__pow__
/	__div__
//	__truediv__
%	Reste d'une division entière (modulo)

TABLE 17 – Méthodes spéciales pour surcharger les opérateurs

— les méthodes `__getitem__()`, `__setitem__()` et `__delitem__()` qui permettent de définir ce que l'on obtiendra lorsque l'on écrira respectivement `object[i]`, `object[i] = valeur` et `del object[i]`.
— les méthodes `__float__()`, `__int__()`, `__str__()` pour convertir un objet dans un format donné.
— la méthode `__len__()` est appelée lorsque l'on souhaite connaître la taille d'un objet avec la méthode `len(objet)`.
— ...

Les méthodes spéciales couvrent ainsi la grande majorité des opérations classiques que l'on est amené à effectuer sur les objets en Python.

25.2 Attributs et méthodes de classe

Un **attribut de classe** est un attribut qui appartient à la classe et non plus à l'objet. Sa valeur est partagée par toute les instances de la classe.

```
class ModeleDObjet:
    attribut_de_classe = 'meme valeur pour tous les objets'

    def __init__(self):
        self.attribut_d_objet = "valable seulement pour l'objet en
            cours"
```

Il n'est pas nécessaire de créer une instance de classe pour avoir accès à l'attribut de classe.

```
>>> print(ModeleDObjet.attribut_de_classe)
meme valeur pour tous les objets
>>> print(ModeleDObjet.attribut_d_objet)
AttributeError: type object 'ModeleDObjet' has no attribute '
    attribut_d_objet'
```

En revanche, une instance de la classe à bien accès aux deux attributs :

```
>>> print(ModeleDObjet().attribut_de_classe)
meme valeur pour pour tous les objets
>>> print(ModeleDObjet().attribut_d_objet)
valable seulement pour l'objet en cours
```

Sur le même principe, on peut créer des **méthodes de classe**. On utilise pour cela un décorateur de fonction : `@classmethod`. La convention dans ce cas est d'utiliser `cls` à la place de `self`. En effet, avec une méthode de classe c'est la classe elle-même qui est passée en premier paramètre de la fonction et non plus l'objet courant.

```
class ModeleDObjet:

    @classmethod
    def methode_de_classe(cls):
        print("Je suis une méthode de classe")
```

L'intérêt des méthodes et attributs de classes et de regrouper au même endroit les choses qui traitent de la même chose. Leur usage permet généralement d'éviter le recours à des attributs globaux dont le comportement n'est pas toujours maîtrisable.

25.3 Les gestionnaires de contexte

Les **gestionnaires de contexte** sont des objets qui nous permettent d'employer une syntaxe du type `with objet as alias:`. Sans en donner le nom, nous en avons déjà utilisé, par exemple pour manipuler un fichier (cf. paragraphe 13).

Une manière de créer un gestionnaire de contexte en Python est de définir une classe possédant deux méthodes spéciales : `__enter__` et `__exit__`. La fonction `__enter__` est déclenchée à l'entrée dans le bloc et permet de définir ce qui sera récupéré par le mot clé `as`, tandis que la fonction `__exit__` est déclenchée à la sortie du bloc `with`.

Dans l'exemple suivant, on définit un gestionnaire de contexte qui sert uniquement à afficher un temps d'exécution : le temps est enregistré à l'entrée dans le bloc et à la sortie, où on affiche alors la différence :

```
import time
class Compteur(object):
    def __init__(self, nom):
        self.nom = nom
    def __enter__(self):
        self.t_debut = time.time()
    def __exit__(self, exc_ty, exc_val, exc_tb):
        t_fin = time.time()
        print('{}: {}'.format(self.nom, t_fin - self.t_debut))
```

On peut l'utiliser pour mesurer le temps d'exécution de n'importe quel bloc d'instructions. Par exemple, pour comparer les performances de boucles `while` et `for` :

```
>>> n = 1000000
>>> with Compteur('Boucle for'):
...     for i in range(n):
...         pass
...
Boucle for: 0.42704200744628906
>>> with Compteur('Boucle while'):
```

```
...     while n > 0:
...         n -= 1
...
Boucle while: 1.0651071071624756
```

Partie VII

Les exceptions

26 Généralités

Réaliser certaines opérations est parfois impossible, comme par exemple diviser par zéro. Python fournit un mécanisme pour gérer ce genre d'erreurs survenues lors de l'exécution d'un programme : les **exceptions**.

Le traitement des exceptions permet d'apporter des solutions à des situations qui auraient entraîné une interruption du programme. Il se découpe en trois étapes :

1. détection de l'erreur, réalisation d'une condition exceptionnelle ;
2. interruption du déroulement normal et lancement d'une exception ;
3. traitement de l'erreur dans le contexte courant et/ou propagation de cette erreur vers un niveau supérieur.

Les exceptions fournissent une moyen efficace de répondre à un événement inhabituel qui est en train de se produire (erreur, cas particulier, événement inattendu, etc.). En effet, une exception ne fait pas qu'interrompre le cours normal d'un programme, elle apporte également des informations sur la source de l'erreur.

A ce stade du cours, vous avez certainement déjà rencontré des exceptions. Elles déclenchent l'affichage dans la console d'un message ressemblant à celui-ci :

```
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    import et-patatras
ImportError: No module named 'et-patatras'
```

On y retrouve toujours deux éléments importants :
— sur la dernière ligne, le *type de l'exception* ainsi que la description la cause de l'erreur ;
— au dessus, un à plusieurs blocs de trois lignes, appelé la *stack trace*, et qui indique les différents appels ayant conduit à l'erreur.
Une erreur se lit donc de bas en haut : le type de l'exception, l'instruction qui a causé l'erreur, et la pile d'appels permettant de retrouver l'origine de l'erreur.

Le type de l'exception se réfère à des catégories bien précises. Les exceptions sont des classes qui dérivent toutes d'une classe mère `BaseException`. La hiérarchie des exceptions les plus fréquemment rencontrées est présentée ci-dessous :

BaseException : regroupe l'ensemble des exceptions
- SystemExit : arrêt du programme
- KeyboardInterupt : l'utilisateur appuie sur des touches interruption du programme
- Exception : regroupe les exceptions qui ne se rapportent pas à la fermeture du programme
  - StopIteration : survient lorsque la fonction next d'un itérateur ne retourne plus rien
  - ArithmeticError : classe mère des exceptions sur les opérations mathématiques
    - OverflowError : résultat d'une expression trop grand pour être représenté
    - ZeroDivisionError : division par zéro
  - AssertionError : échec d'une instruction assert
  - AttributeError : utilisation d'un attribut inexistant d'un objet
  - EOFError :
  - ImportError : import d'un module inexistant
  - LookupError : classe mère des exceptions sur les index ou clés
    - IndexError : accès à un élément d'une séquence avec un indice qui n'existe pas
    - KeyError : accès à un élément d'un dictionnaire avec une clé qui n'existe pas
  - NameError : la variable ou fonction manipulée n'est pas déclarée
  - OSError : classe mère pour toutes les erreurs de manipulation de fichier
    - ConnectionError :
    - FileNotFoundError : le fichier n'existe pas
    - PermissionError : pas les droit pour effectuer l'opération sur un fichier
  - ReferenceError
  - RuntimeError
    - NotImplementedError
  - SyntaxError : erreur de syntaxe générale
    - IndentationError : erreur d'indentation
  - SystemError
  - TypeError : le type de la variable ne permet pas d'effectuer l'opération demandée
  - ValueError : le type est correct, mais pas la valeur

On notera que les exceptions sont, comme toutes choses en Python, des objets. Il est donc possible de définir ses propres types d'exception pour gérer des erreurs spécifiques à un programme. On créera pour cela des classes héritant de `BaseException` ou d'une exception qui en dérive.

27 Traitement d'une exception

Lorsqu'une exception est activée, on dit qu'elle est levée. Pour **lever une exception** soit même, on utilise la syntaxe suivante :

```
raise NomDeLExceptionALever("Message d'explication")
```

Une exception peut être levée lorsque l'on est dans une configuration que l'on sait non prévue par le programme. Par exemple le type d'un paramètre n'est pas celui attendu :

```
def ma_fonction(param):
    if not isinstance(param, int):
      raise TypeError("Le paramètre doit être un entier")
raise NomDeLExceptionALever("Message d'explication")
```

Si le paramètre n'est pas un entier, une exception est soulevée :

```
>>> ma_fonction(1.5)
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
```

```
    ma_fonction(1.5)
  File "<pyshell#36>", line 2, in ma_fonction
    if not isinstance(param, int):raise TypeError("Le paramètre
       doit être un entier")
TypeError: Le paramètre doit être un entier
```

Mais l'intérêt des exceptions ne se limite pas à arrêter le programme en affichant de jolis messages d'erreurs. Elles permettent avant tout de gérer des cas particuliers pour justement éviter que le programme ne fasse que s'arrêter. Lorsqu'un bloc de code est susceptible de soulever une exception, on utilise des instructions `try/except` :

```
try:
    instructions succeptibles de soulever une exception
except NomDeLException:
    instructions à exécuter si l'exception est levée
```

Par exemple :

```
try:
    return 1 / x
except ZeroDivisionError:
    return float('inf')
```

Si x vaut 0, Python va lever une exception `ZeroDivisionError`, le bloc `except` sera appelé et la valeur `float('inf')` sera retournée.

Cela suppose de savoir quelle exception on souhaite intercepter (en cas de doute, le plus simple est généralement de provoquer volontairement l'exception).

Il est possible de gérer plusieurs types d'exceptions pour le même bout de code.

```
liste = ['poule', 'vache', 'cochon']
try:
    return liste[10 % i]
except ZeroDivisionError, IndexError:
    return None
except TypeError:
    print("i doit être un nombre")
```

Remarque : il est aussi possible de ne pas spécifier le type d'exception à attraper, pour que Python intercepte toutes les erreurs possibles.

```
liste = ['poule', 'vache', 'cochon']
try:
    return liste[10 % i]
except:
    print("Problème...")
```

C'est généralement une mauvaise idée, car si une erreur arrive, on sera incapable de savoir ce qui l'a causée.

La structure `try/else` peut être complétée par des clauses `else` et/ou `finally`.

La clause `else` s'exécute lorsqu'aucune erreur n'a été interceptée à l'exécution du bloc de code `try`. La clause `finally` permet d'exécuter des instructions dans tous les cas, qu'une exception ai été levée ou pas. Elle est parcourue après toutes les autres clauses.

Exemple avec l'ouverture d'un fichier

```
chemin = "D:\..."
try:
    fichier = open(chemin, 'r')  # tentative d'ouverture du fichier
except FileNotFoundError:
    print("Le fichier n'existe pas")
except IOError:
    print("Erreur à la lecture du fichier")
else:
    print("Fichier ouvert")
    print(fichier.read())  # affichage du contenu
finally:
    fichier.close()  # on referme le fichier
```

Partie VIII

Annexes

Installer des librairies Python

Une des grandes forces de Python, c'est la multitude de libraires disponibles. On est donc souvent amené à installer de nouvelles librairies (on par le aussi de paquets). Pour installer un paquet Python, la première méthode est de télécharger les sources, qui contiennent un fichier `setup.py`, puis de saisir dans une invite de commande :

```
python setup.py install
```

Ce n'est pas des plus pratique, notamment l'étape du téléchargement (où trouver le paquet ? comment savoir si c'est bien la dernière version ?). De plus, il arrive fréquemment que des paquets nécessitent la présence d'autres paquets pour pouvoir fonctionner (on parle de dépendances). On doit donc trouver tous les paquets, puis les télécharger et installer individuellement...

Python nous facilite la vie avec un système de gestion de paquets : **pip**. Depuis les dernières version de Python, pip est installé par défaut. pip permet d'installer simplement un paquet en saisissant [10] :

```
pip install paquet
```

La syntaxe n'apporte pas grand chose, mais...
— il n'est plus nécessaire de télécharger le paquet : pip s'en charge tout seul
— il n'est plus nécessaire de récupérer les dépendances : pip s'en charge tout seul
En arrière plan, pip se connecte au site http://pypi.python.org/pypi où chaque développeur Python est libre d'enregistrer ses paquets, en respectant quelques conditions, dont le renseignement de dépendances.

pip permet de choisir la version que l'on souhaite installer. Par exemple, la version 1.8 de Django :

```
pip install Django==1.8
```

Il est également possible de mettre à jour un paquet :

```
pip install Django --upgrade
```

Mais également de désinstaller les paquets devenus inutiles :

```
pip uninstall Django
```

Bref, la méthode pip ne présente que des avantages. Elle atteindra ses limites lorsque l'on cherchera à installer de grosses librairies types PyQt ou avec des extensions en C type numpy.

10. Selon la configuration du poste, il peut être nécessaire de se placer dans le répertoire Script de l'installation de Python.

Lorsque l'on commence à beaucoup développer, il arrive que l'on soit amené à travailler sur des projet basées sur des versions différentes d'une même librairie. Parfois on souhaite aussi juste tester une librairie ou une version sans modifier tous notre environnement. **virtualenv** est la solution à ce genre de problèmes.

Il s'agit d'un paquet Python qui permet de créer des espaces de travail Python isolés les uns des autres. Pour l'installer, on fait :

```
pip install virtualenv
```

On crée ensuite un environnement virtuel avec la commande :

```
virtualenv /chemin/de/l/espace/de/travail
```

Pour travailler dans cet environnement virtuel, on lance le fichier activate.bat contenu dans le répertoire. Une fois dans l'environnement virtuel, on peut installer nos paquets Python avec classiquement avec pip. Ils ne seront visibles que depuis l'environnement virtuel.

Choix d'une implémentation de Python

En introduction de ce cours (chapitre 1.5. Exécution d'un programme en Python), nous avons définit l'interpréteur Python comme un programme exécutant des programmes écrit en Python. Nous avons également précisé que cet interpréteur est lui-même écrit dans un langage différent de Python, mais sans donner plus de détail sur ce langage. En réalité, il existe plusieurs interpréteurs Python répondant à différentes problématiques et écrits dans différents langages.

CPython est l'interpréteur par défaut. C'est le standard d'implémentation, celui qui est utilisé par une majorité de personnes. L'interpréteur Python est écrit en C. Il est performant pour l'utilisation d'outils en C depuis Python.

Pour rendre performante l'utilisation d'outils Java ou .Net avec Python, d'autres implémentation de Python ont été développées : respectivement Jython [11] et IronPython [12]. Le bytecode produit par ces interpréteur n'est pas compatible avec celui d'une machine CPython.

Stackless [13] est une autre implémentation de Python conçue pour une utilisation très ciblée : l'amélioration des performances lors des traitements multi-tâches.

PyPy [14] est une mise en oeuvre du langage Python qui présente la particularité d'être écrite en elle-même en Python. L'objectif de PyPy est d'aboutir à une version plus rapide que l'implémentation en C classique. Les porteurs du projet rapportent des gains de performance compris entre *5 et *100. Ils estiment également que leur interpréteur n'est pas encore assez stable pour être déployée massivement.

Cython [15] permet de transformer du Python en C, afin de l'embarquer dans un autre code

11. http://www.jython.org/
12. http://ironpython.net/
13. http://www.stackless.com
14. http://pypy.org/
15. http://cython.org/

C, et inversement de permettre des appels de C en Python plus faciles. Cet interpréteur écrit en C permet ainsi de compiler un programme autonome écrit en Python.

D'autres implémentation du langage Python existent pour répondre à d'autres besoins spécifiques. Le développeur n'aura pratiquement jamais à se poser la question de l'implémentation de Python à choisir pour un projet. Elles sont à considérer comme des bonus qui peuvent aider un développeur chevronné à résoudre un problème particulier.

Bibliographie

Un peu de lecture pour en apprendre plus sur le langage Python :

Downez Allen, *Think Python (How to think like a computer scientist)*, 2012, Green Tea Press

Bersini Hugues, *La programmation orientée objet*, 2013, Eyrolles

Beazley David and Jones Brian, *Python Cookbook 3rd edition*, 2013, O'Reilly

Lambert Kenneth A., *Fundamentals of Python*, 2010, Cengage Learning

Lutz Mark, *Programming Python 4th edition*, 2010, O'Reilly

Lutz Mark, *Learning Python 5th edition*, 2013,O'Reilly

Romano Fabrizio, *Learning Python*, 2015, Packt Publishing

Dupré Xavier, *Programmation avec le langage Python*, 2010, ENSAE, http://www.xavierdupre.fr/enseignement/initiation/initiation_via_python_ellipse_mai_2010.pdf

Swinnen Gérard, *Apprendre à programmer avec Python 3*, 2012, http://inforef.be/swi/download/apprendre_python3_5.pdf

Site Sam&Max : http://sametmax.com/cours-et-tutos/