Langage Python

Repères

La table des matières est détaillée à la fin du document.

Chapitre 1

Introduction

1.1 Ordinateur et langages

L'informatique présentée dans ce cours est abordée d'un point de vue utilitaire. C'et outil permet d'automatiser des tâches répétitives, d'accélérer des calculs complexes comme l'approximation d'un calcul intégral (voir figure 1.1). Le langage *Python*, sans être le plus rapide des langages de programmation, permet dans la plupart des cas de concevoir plus rapidement un programme informatique répondant à un objectif donné. Mais avant de rentrer dans le vif du sujet, voici quelques détails techniques et quelques termes fréquemment employés en informatique.

1.1.1 L'ordinateur

On peut considérer simplement qu'un ordinateur est composé de trois ensembles :

- le microprocesseur
- la mémoire
- les périphériques (écran, imprimantes, disque dur, ...)

Le microprocesseur est le cœur de l'ordinateur, il suit les instructions qu'on lui donne et ne peut travailler qu'avec un très petit nombre d'informations.

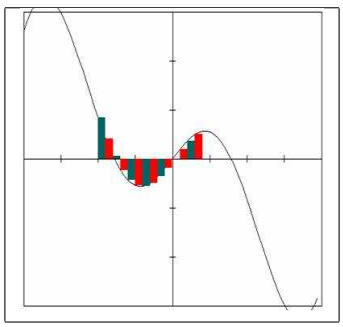

FIG. 1.1 – *Illustration du calcul approché d'une intégrale à l'aide de la méthode des rectangles.*

C'est pourquoi on lui adjoint une mémoire avec laquelle il échange sans arrêt des données. Sa capacité se mesure en octets (kilo-octets, mégaoctets, gigaoctets ou leurs abréviations Ko, Mo, Go). Ces échanges entre processeur et mémoire sont rapides.

Les périphériques regroupent tout le reste (écran, clavier, souris, disque dur, imprimante...), tout ce qui nous permet de dialoguer avec l'ordinateur et tout ce qui nous permet de conserver des informations une fois que celui-ci est éteint.

1.1.2 Termes informatiques

Définition 1.1.1 : algorithme
Un algorithme est une suite finie de règles à appliquer dans un ordre déterminé à un nombre fini de données pour arriver avec certitude, en un nombre fini d'étapes, à un certain résultat et cela, indépendamment des données. Leur écriture est indépendante du langage choisi.

Qu'il soit écrit en Basic, en Pascal, en C, en *Python*, en français, un algorithme reste le même, ce qui favorise l'apprentissage d'un nouveau langage informatique lorsqu'on en connaît déjà un.

Définition 1.1.2 : programme
Un programme informatique est une suite d'instructions ou séquence d'instructions. C'est la réalisation informatique d'un algorithme. Il dépend du langage.

Définition 1.1.3 : compilateur et compilation
Le compilateur est un programme qui traduit un code écrit dans un langage de programmation (ici le C) en langage dit "machine", compréhensible par l'ordinateur. La compilation est le fait de traduire un programme afin que l'ordinateur le comprenne.

Par la suite, certaines erreurs de compilations courantes seront citées, celles-ci dépendant du compilateur choisi qui est pour ce cours celui du langage *Python*.

Définition 1.1.4 : mot clé
Un mot clé est une composante du langage et fait partie de sa grammaire.

La table 3.1 (page 33) regroupe les mots-clé du langage *Python*.

1.2 Présentation du langage *Python*

1.2.1 Histoire

*Python*est un langage objet interprété de haut niveau, créé au début des années quatre-vingt-dix par Guido Van Rossum au Centrum voor Wiskunde à *Informatica*, Amsterdam. . En 1995, Rossum poursuivit le développement de *Python*à la *Corporation for National Research Initiatives* de Reston (Virginie). Et en 2000, Rossum créa l'équipe BeOpen PythonLabs qui, en octobre de la même année est incorporée à Zope Corporation puis à la société Digital Creations. En 2001, la PSF (Python Software Foundation) est créée. Il s'agit d'une organisation à but non lucratif détenant les droits de propriété intellectuelle de Python. Elle est sponsorisée en particulier par Zope Corporation. *Python*est distribué sous forme de logiciel libre.

*Python*est couvert par sa propre licence (voir le site [www-PyLicence] ou le chapitre 6.2).Toutes les versions depuis la 2.0.1 sont compatibles avec la licence GPL (GNU Public Licence [1])..

1.2.2 Description sommaire

On distingue plusieurs classes parmi les langages informatiques selon la syntaxe qu'ils proposent ou les possibilités qu'ils offrent. *Python*est un langage :

1. interprété
2. orienté objet
3. de haut niveau
4. modulaire
5. à syntaxe positionnelle

Le langage *Python*est dit *interprété* car il est directement exécuté sans passer par une phase de compilation qui traduit le programme en langage machine, comme c'est le cas pour le langage C. En quelque sorte, il fonctionne autant comme une calculatrice que comme un langage de programmation. Afin d'accélérer l'exécution d'un programme *Python*, il est néanmoins possible de traduire un programme dans un langage (bytecode) qui est ensuite interprété par une machine virtuelle *Python*. Ce mécanisme est semblable à celui propre au langage Java.

On considère que le langage *Python*est de haut niveau car il propose des fonctionnalités avancées et automatiques telle le *garbage collecting*. Cette tâche correspond à la destruction automatique des objets créés lorsqu'ils ne sont plus utilisés. Il propose également des structures de données complexes telles que des dictionnaires, éloignées des types numériques standards.

Le langage *Python*est modulaire. La définition du langage est très succinte et autour de ce noyau concis, de nombreuses librairies ou modules ont été développées. *Python*est assez intuitif, être à l'aise avec ce langage revient à connaître tout autant sa syntaxe que les nombreux modules disponibles, eux-mêmes écrits en *Python*.

Le langage *Python*est à syntaxe positionnelle en ce sens que l'indentation fait partie du langage. Le point virgule permet de séparer les instruction en langage C, l'accolade permet de commencer un bloc d'instruction. En *Python*, seule l'indentation permet de marquer le début et la fin d'un tel bloc, ce procédé consiste à décaler les lignes vers la droite pour montrer qu'elles appartiennent au même bloc d'instructions.

1.2.3 Avantages et inconvénients du langage *Python*

Alors qu'il y a quelques années, le langage C puis le langage C++ s'imposaient souvent comme langage de programmation, il existe dorénavant une profusion de langages (Java, PHP, Visual Basic, Perl, ...). Il est souvent possible de transposer les mêmes algorithmes d'un langage à un autre. Le choix approprié est alors celui qui offre la plus grande simplicité lors de la mise en œuvre d'un programme et aussi lors de son utilisation (vitesse d'exécution notamment).

Comme la plupart des langages, le langage *Python*est tout d'abord portable puisqu'un même programme peut être exécuté sur un grand nombre de systèmes d'exploitation comme Linux, Windows, Mac Os... *Python*possède également l'avantage d'être entièrement gratuit tout en proposant la possibilité de pouvoir réaliser des applications commerciales à l'aide de ce langage. Les paragraphes qui suivent présentent les avantages et les inconvénients de *Python*face à d'autres langages.

1. voir le site [www-GPL].

1.2.3.1 *Python*et Java

La syntaxe de *Python*est beaucoup plus simple que celle de Java (proche du C), ce qui améliore de façon très significative les temps de développement. Le programmeur ne perd pas de temps en déclaration de types, de variables, ... *Python*intègre des types de données très puissants, comme les listes et dictionnaires polymorphiques qui simplifient considérablement le travail de programmation. Enfin, *Python*est un langage totalement ouvert et libre, qui ne dépend d'aucune entreprise particulière.

1.2.3.2 *Python*et Perl

*Python*et Perl partagent un certain nombre de concepts mais leurs philosophies sont totalement différentes. Perl est plutôt destiné à programmer des tâches de bas niveau, avec son système d'expressions régulières, d'analyse de fichier et de génération de rapport. *Python*est plus orienté vers le développement d'applications, nécessitant des structures de données plus complexes et encourage le programmeur à produire du code facile à maintenir.

1.2.3.3 *Python*et C++

La plupart des remarques concernant Java s'appliquent à C++. Ajoutons encore que si le code *Python*est typiquement trois à cinq fois plus court que le code Java équivalent, il est de cinq à dix fois plus court que le code C++ correspondant. C'est un gain de temps notable lors des phases de développement et de maintenance des programmes. Un programme C++ nécessite une recompilation chaque fois que l'on change d'environnement, un programme compilé sur une plate-forme ne pouvant en aucun cas être exécuté sur une autre. A l'inverse, un programme *Python*s'exécutera sur toute plate-forme disposant de la machine virtuelle Python. Son principal inconvénient face au langage C++ est sa vitesse d'exécution, plus lente.

1.2.3.4 Conclusion

Si le langage C reste le langage de prédilection pour l'implémentation d'algorithmes complexes et gourmands en temps de calcul ou en capacités de stockage, un langage tel que *Python*suffit dans la plupart des cas. De plus, lorsque ce dernier ne convient pas, il offre toujours la possibilité, pour une grande exigence de rapidité, d'intégrer un code écrit dans un autre langage tel que le C ou Java, et ce, d'une manière assez simple.

1.3 Installation sur Windows

1.3.1 Installation du langage *Python*

*Python*a l'avantage d'être disponible sur de nombreuses plate-formes comme *Windows*, *Linux* ou encore *Macintosh*. L'installation présentée ici concerne le système d'exploitation Windows uniquement. Toutefois, excepté ce paragraphe, les exemples décrits par la suite ne dépendront pas d'un quelconque système.

Sous *Windows*, il suffit de décompresser le fichier *Python-2.3.4.exe* disponible à l'adresse [www-PyDownload] et dont la documentation associée est fournie à l'adresse [www-PyDoc]. Il est conseillé de télécharger la dernière version stable du langage. Les options d'installation choisies sont celles par défaut, le répertoire d'installation est *C:/Python23*. A la fin de cette installation apparaît un menu supplémentaire dans le menu *Démarrer* (ou *Start*) de Windows comme le montre la figure 1.2.

FIG. 1.2 – *Menu ajouté à Windows*

Ce menu contient les intitulés suivant :

IDLE (Python GUI)	éditeur de texte, pour programmer
Module Docs	pour rechercher des informations dans la documentation
Python (command line)	ligne de commande *Python*
Python Manuals	documentation à propos du langage *Python*
Uninstall Python	pour désinstaller *Python*

La documentation décrit en détail le langage *Python*, elle inclut également un tutoriel qui permet de le découvrir. La ligne de commande (voir figure 1.4) permet d'exécuter des instructions en langage *Python*. Elle est pratique pour effectuer des calculs mais reste contre-indiqué lorsqu'il s'agit d'écrire un programme. C'est pourquoi il est nécessaire d'utiliser un éditeur de texte qui permet d'écrire un programme, de le sauvegarder, et de ne l'exécuter qu'une fois terminé au lieu que chaque ligne de celui-ci ne soit interprétée immédiatement après qu'elle a été écrite.

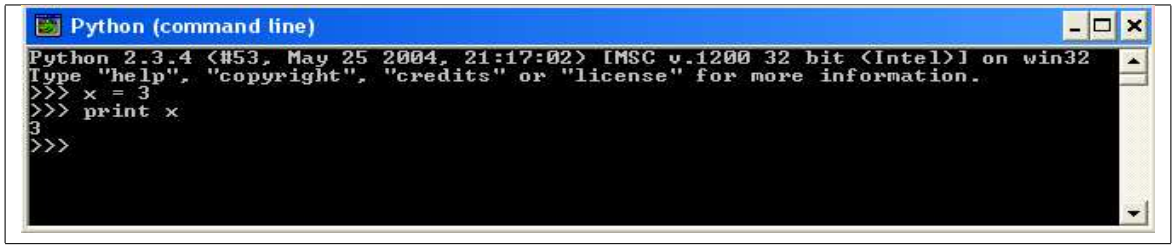

FIG. 1.3 – *Ligne de commande, la première ligne affecte la valeur 3 à la variable x, la seconde ligne l'affiche.*

1.3.2 Utilisation de l'éditeur de texte

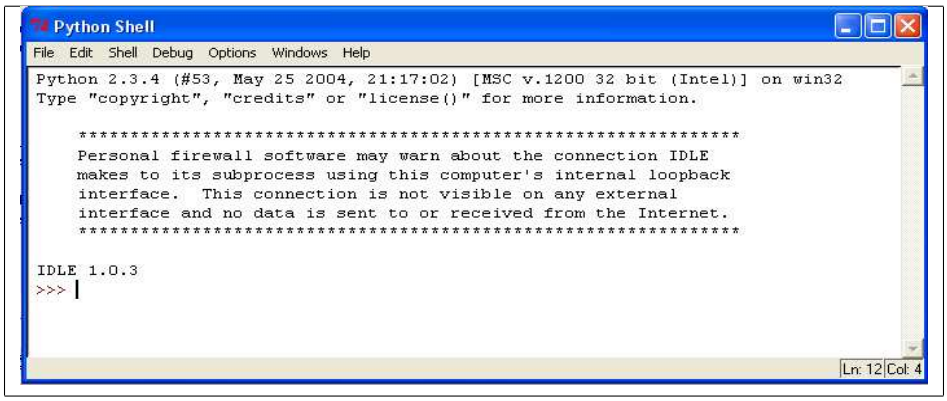

FIG. 1.4 – *Fenêtre de commande fournie avec le langage Python.*

La figure 1.2 montre le menu installé par *Python*dans le menu "Démarrer" de Windows. En choisissant l'intitulé "IDLE (Python GUI)", on active la fenêtre de commande de *Python*(voir figure 1.4).Les instructions sont interprétées au fur et à mesure qu'elles sont tapées au clavier. Après chaque ligne, cette fenêtre de commande conserve la mémoire de tout ce qui a été exécuté. Par exemple :

```
>>> x = 3
>>> y = 6
>>> z = x * y
>>> print z
18
>>>
```

Après l'exécution de ces quelques lignes, les variables `x` , `y` , `z` existent toujours. Pour effacer toutes les variables créées, il suffit de redémarrer l'interpréteur par l'intermédiaire du menu `Shell` $-->$ `RestartShell` . Les trois variables précédentes auront disparu.

Il n'est pas possible de conserver le texte qui a été saisi au clavier, ce programme conserve seulement les résultats de son interprétation. Il est possible toutefois de rappeler une instruction déjà exécutée par l'intermédiaire de la combinaison de touches `ALT` $+$ `p` , pressée une ou plusieurs fois. La combinaison `ALT` $+$ `n` permet de revenir à l'instruction suivante. Pour écrire un programme et ainsi conserver toutes les instructions, il suffit d'actionner le menu `File` $-->$ `NewWindow` qui ouvre une seconde fenêtre qui fonctionne comme un éditeur de texte (voir figure 1.5).

FIG. *1.5 – Fenêtre de programme, le menu* `Run` $-->$ `RunModule` *permet de lancer l'exécution du programme. L'interpréteur Pythonest réinitialisé au début de l'exécution et ne conserve aucune trace des travaux précédents.*

Après que le programme a été entré, le menu `Run` $-->$ `RunModule` exécute le programme. Il demande au préalable s'il faut enregistrer puis réinitialise l'interpréteur *Python*pour effacer les traces des exécutions précédentes. Le résultat apparaît dans la première fenêtre (celle de la figure 1.4). La pression des touches "Ctrl + C" permet d'arrêter le programme avant qu'il n'arrive à sa fin.

Remarque 1.3.1: fenêtre intempestive
Si on veut de débarasser de cette fenêtre intempestive qui demande confirmation pour enregistrer les dernières modifications, il suffit d'aller dans le menu `Options` $-->$ `ConfigureIDLE` et de choisir `No prompt` sur la quatrième ligne dans la rubrique `General` (voir figure 1.6).

Cette description succinte permet néanmoins de réaliser puis d'exécuter des programmes simples. Les autres fonctionnalités sont celles d'un éditeur de texte classique, notamment la touche `F1` qui débouche sur l'aide associée au langage *Python*. Il est possible d'ouvrir autant de fenêtres qu'il y a de fichiers à modifier simultanément. Néanmoins, il existe d'autres éditeurs plus riches comme celui proposé au paragraphe 1.3.3. Ils sont souvent conçus de manière à s'adapter à plusieurs autres langages tels que C,

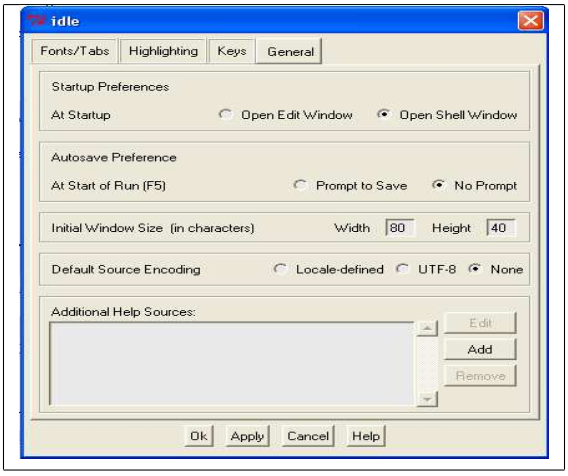

FIG. 1.6 – *Cliquer sur* `General` *puis sur* `Noprompt` *pour se débarrasser de la fenêtre intempestive qui demande à l'utilisateur de confirmer la sauvegarde des dernières modifications.*

Perl, PHP, ou HTML. Il suffit pour cela de leur indiquer l'emplacement du compilateur associé au langage choisi. Cette configuration est souvent manuelle mais semblable d'un éditeur à l'autre.

1.3.3 Installation d'un autre éditeur de texte

Il existe de nombreux éditeurs de texte, payant ou gratuit. Celui qui est proposé est *Syn Text Editor*, il est gratuit et fonctionne avec de nombreux compilateurs de langages différents (voir figure 1.8). Il existe d'autres éditeurs, dont la plupart sont référencés par le site officiel de *Python*à l'adresse [www-PyEdit]. Pour installer cet éditeur, il suffit d'exécuter le fichier *synsetup-2.1.0.43.exe* téléchargeable depuis l'adresse [www-Syn] dans la rubrique *Download*.

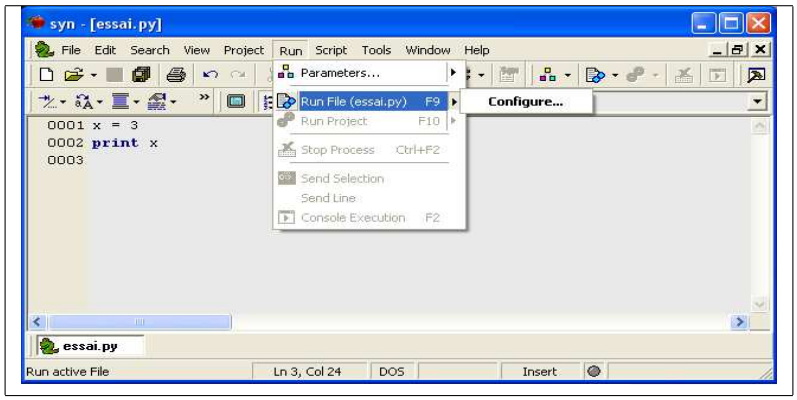

FIG. 1.7 – *Même programme que celui de la figure 1.4, ce programme est enregistré sous le nom "essai.py", et on tente de l'exécuter par l'intermédiaire du menu "Run -> Run File".*

La fenêtre de la figure 1.9a apparaît où aucun champ n'est renseigné. Il faut alors ajouter un profile qu'on appellera *python*. La fenêtre 1.9b apparaît. Une fois cette étape terminée, il faut cliquer sur l'icône cerclé de rouge de l'image 1.9a pour insérer un nom de programme, ici "essai". Une fois ce nom ajouté, on clique dessus pour enfin renseigner les derniers champs comme indiqué dans la figure 1.9a afin de préciser l'emplacement du compilateur *Python*.

Après cette étape, il ne reste plus qu'à exécuter le programme en appuyant sur la touche F9. Le résultat

s'affiche dans la fenêtre "Output" en bas de l'écran comme le montre la figure 1.10. Une erreur de syntaxe comme dans le programme suivant :

```
x = 3
print xy
```

aboutira à une erreur de compilation à la seconde ligne indiquant que la variable `xy` n'existe pas :

```
=============== Running Profile: python ===============

------------------------------------------------------------
Commandline: C:/Python23/python.exe c:/temp/essai.py
Workingdirectory: c:/temp
Timeout: 10 sec, Delay: 0 sec, Started: 08/14/04 13:45:13
------------------------------------------------------------

Traceback (most recent call last):
  File "c:/temp/essai.py", line 2, in ?
    print xy
NameError: name 'xy' is not defined

Process "essai" completed with Exit Code 1, at 08/14/04 13:45:13.
```

Cet éditeur possède néanmoins un inconvénient qui apparaît lors de l'utilisation de la fonction `raw_input` qui permet de saisir une réponse au clavier. Cette fonction bloque le programme. Dans ce cas, il est préférable d'utiliser l'éditeur de texte fourni avec le langage *Python*.

1.4 Notes à propos de ce document

Ce document décrit les points essentiels du langage *Python*. Une description plus complète est disponible dans le livre [Martelli2004] qui s'intéresse sommairement aux nombreuses extensions disponibles recouvrant des thèmes comme les calculs numériques (matriciels, ...), le traitement des images, la communication via un réseau ou encore l'utilisation du langage C via *Python*et l'utilisation réciproque, la lecture et l'écriture de données au format XML. Le site officiel du langage *Python*([www-Python]) abrite la version la plus

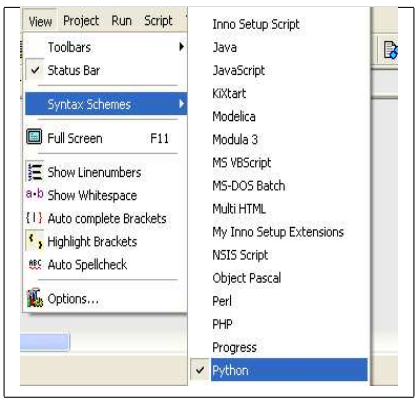

FIG. 1.8 – *Configuration de l'éditeur, il est possible de configurer l'éditeur pour de nombreux langages.*

récente ainsi que de nombreux modules à l'adresse ([www-PyModule]). Les fonctionnalités offertes par ces modules sont très variées. Le module *pyXLWriter* permet par exemple d'écrire des données au format Excel. Le module FSA (Finite State Automate) implémente quelques algorithmes concernant les automates à états finis. Le module *MySQL-python* offre quant à lui un accès à *MySQL* via *Python*.

Etant donné que *Python*est récent, il existe encore peu de livres consacrés à ce langage. La plupart des documents sont disponibles via internet en anglais principalement comme sur le site officiel voire en français (voir le site [www-DocF]). Vieux d'une dizaine d'années, *Python*est un langage en expansion dont la richesse réside surtout dans la liste des modules disponibles. Cette liste couvre un large panel de besoins susceptible de s'accroître avec le temps. Il peut être parfois utile de jeter un coup d'œil aux modules recensés sur internet ([www-Python]) avant de se lancer dans la programmation pour découvrir peut-être qu'un module recouvre une partie du programme sur le point d'être développé.

L'intérêt d'un tel langage est de parvenir à construire le programme répondant à votre problème le plus rapidement et le plus simplement possible. Il n'est pas aussi rapide qu'un langage comme le C mais il propose une approche simple qu'il est possible d'étoffer lorsque cela est nécessaire à l'aide d'autres langages plus rapides. Ce document n'a pas pour objectif d'étudier le langage *Python*de manière approfondie mais de proposer un outil informatique simple d'utilisation au service de travaux statistiques, économiques, financiers...

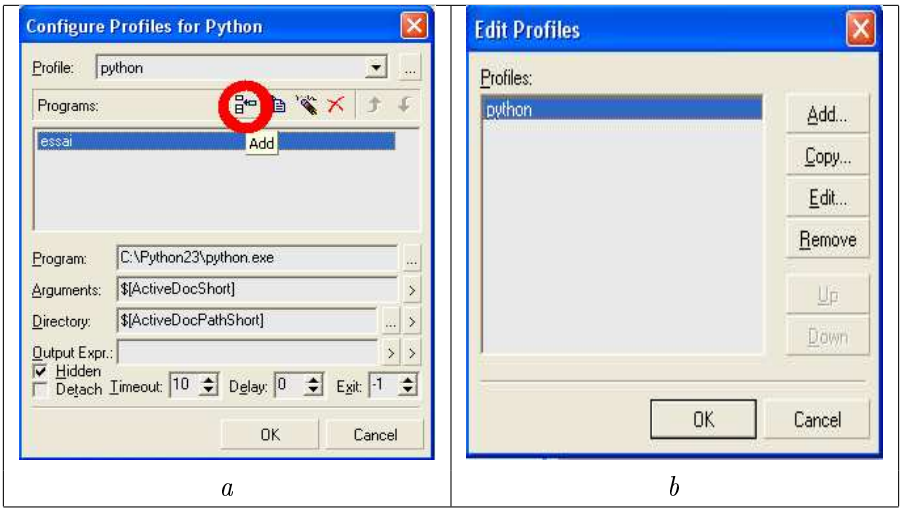

FIG. 1.9 – *Configuration de l'éditeur, ajout d'un profil qui permet d'exécuter le programme avec le compilateur Python.*

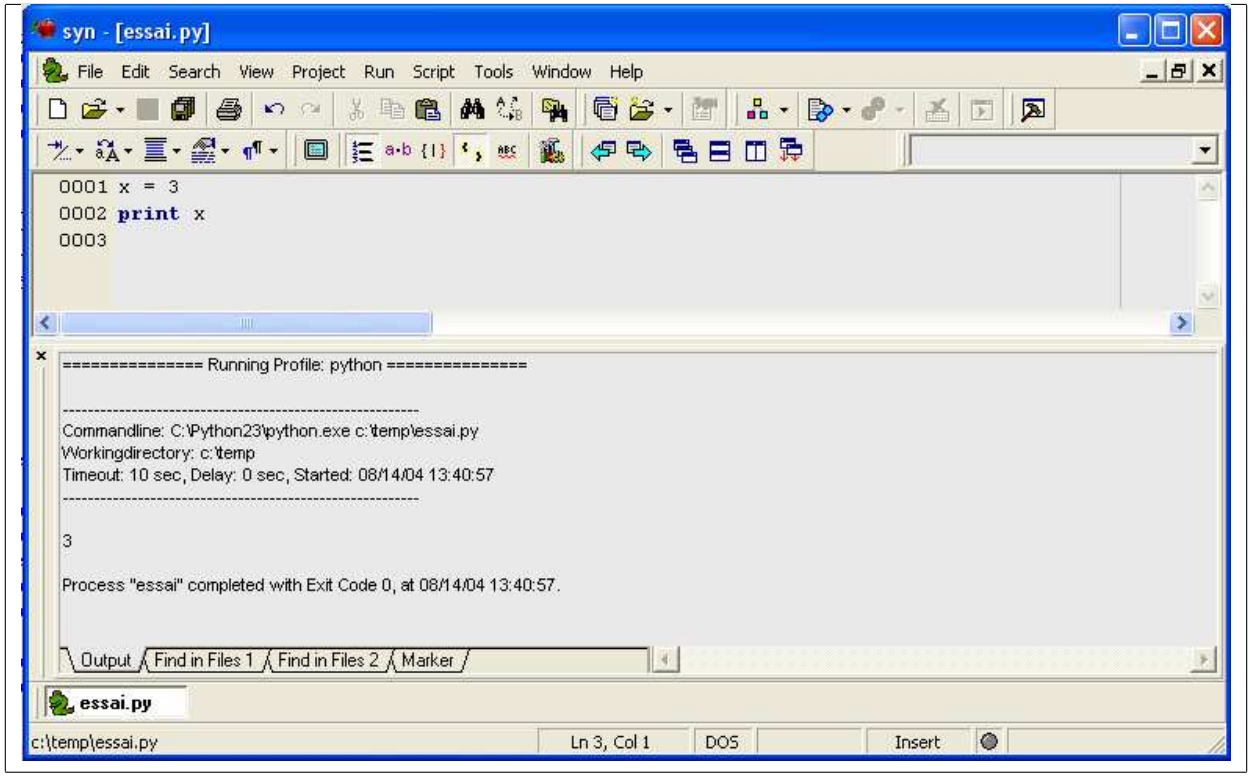

FIG. 1.10 – *Exécution du programme "essai.py" dans la fenêtre "Output".*

Chapitre 2

Type de variables du langage *Python*

2.1 Variables

2.1.1 Définition

Les variables permettent de manipuler des informations en les nommant. Elles jouent un rôle semblable aux inconnues dans une équation mathématique. Par exemple, on désire calculer la somme de n nombres $(u_1,\ldots,u_n)$. Or l'ordinateur ne sait bien souvent manipuler que deux nombres à la fois. Par conséquent, il est nécessaire de créer une variable intermédiaire qu'on appellera par exemple `somme` de manière à conserver le résultat des sommes intermédiaires.

```
somme = 0
for i in range(1,n) :       # pour tous les indices de 1 à n
    somme = somme + u [i]   # on ajoute le ième élément à somme
```

Définition 2.1.1 : variable
Une variable est caractérisée par :

un identificateur : il peut contenir des lettres, des chiffres, des blancs soulignés mais il ne peut commencer par un chiffre. Minuscules et majuscules sont différenciées.

un type : c'est une information sur le contenu de la variable qui indique au compilateur *Python*la manière de manipuler cette information.

Et avec les variables, leur contraire, les constantes :

Définition 2.1.2 : constante
Les constantes sont le contraire des variables, ce sont toutes les valeurs numériques, chaînes de caractères, ..., tout ce qui n'est pas désigné par un nom. Les constantes possède un type mais pas d'identificateur.

Le langage *Python*possède un certain nombre de types de variables déjà définis ou types fondamentaux à partir desquels il sera possible de définir ses propres types (voir chapitre 4). Au contraire de langages tels que le C, il n'est pas besoin de déclarer une variable pour signifier qu'elle existe, il suffit de lui affecter une valeur. Le type de la variable sera défini par le type de la valeur qui lui est affectée. Le type d'une

variable peut changer, il correspond toujours au type de la dernière affectation.

```
x   = 3.5         # création d'une variable nombre réel appelée x initialisée à 3.5
str = "chaîne"    # création d'une variable chaîne de caractères appelée str
                  # initialisée à "chaîne"
```

Remarque 2.1.3: commentaires
Pour tous les exemples qui suivront, le symbol # apparaîtra à maintes reprises. Il marque le début d'un commentaire que la fin de la ligne termine. Autrement dit, toute information aidant à la compréhension du programme mais n'en faisant pas partie comme dans l'exemple qui suit.

```
x = 3 # affectation de la valeur 3 à la variable x
```

Remarque 2.1.4: instruction sur plusieurs lignes
Le *Python*impose une instruction par ligne. Il est donc impossible de créer deux variables sur la même ligne, de réaliser deux affectations sur une même ligne. La réciproque est aussi vraie, il n'est pas non plus possible d'utiliser deux lignes pour écrire une affectation à moins de conclure chaque ligne qui n'est pas la dernière par le symbole \ . L'exemple suivant est impossible.

```
x =
    5.5
```

Il devrait être rédigé comme suit :

```
x =  \
    5.5
```

Avec ce symbole, les longues lignes peuvent être écrites sur plusieurs de manière plus lisibles, de sorte qu'elles apparaissent en entier.

2.2 Types immuables (ou *immutables*)

> **Définition 2.2.1 : type immuable (ou immutable)**
> Une variable de type immuable ne peut être modifiée. Une opération sur une variable de ce type entraîne nécessairement la création d'une autre variable du même type, même si cette dernière est temporaire.

Autrement dit, la simple instruction `x+ = 3` qui consiste à ajouter à `x` la valeur `3` crée une seconde variable dont la valeur est celle de `x` augmentée de `3` puis à en recopier le contenu dans celui de la variable `x` . Les types immuables sont de taille fixe à l'exception des T-uples et des chaînes de caractères. Tous les autres types ne sont qu'un assemblage de types immuables.

2.2.1 Type "rien"

*Python*propose un type `None` pour signifier qu'une variable ne contient rien. La variable est de type `None` et est égale à `None` .

```
s = None
print s    # affiche None
```

Certaines fonctions utilise cette convention lorsqu'il leur est impossible de retourner un résultat. Dans ce cas, il est possible de générer une erreur (ou exception, voir paragraphe 4.1), de retourner une valeur par défaut ou encore de retourner `None` . Il n'y a pas de choix meilleur que les autres, il suffit juste de préciser la convention choisie.

2.2.2 Nombres réels et entiers

Il existe deux types de nombres en *Python*, les nombres réels (`float`) et les nombres entiers (`int`). L'instruction `x = 3` crée une variable de type `int` initialisée à `3` tandis que `y = 3.5` crée une variable de type `float` initialisée à `3.5` . Le programme suivant permet de vérifier cela en affichant pour les variables `x` et `y` , leurs valeurs et leurs types respectifs.

```
x = 3
y = 3.5
print "x =", x, type(x)
print "y =", y, type(y)
```

Ce qui donne à l'écran :

```
x = 3 <type 'int'>
y = 3.5 <type 'float'>
```

Voici la liste des opérateurs qui s'appliquent aux nombres réels et entiers.

opérateur	signification	exemple
« »	décalage à gauche, à droite	`x = 8 << 1` (résultat = 16)
|	opérateur logique ou bit à bit	`x = 8|1` (résultat = 9)
&	opérateur logique et bit à bit	`x = 11&2` (résultat = 2)
+ -	addition, soustraction	`x = y + z`
+= -=	addition ou soustraction puis affectation	`x+ = 3`
* /	multiplication, division	`x = y * z`
//	division entière	`x = y//3`
%	reste d'une division entière (modulo)	`x = y%3`
*= /=	multiplication ou division puis affectation	`x* = 3`
**	puissance	`x = y * *3`

Les trois premiers résultats s'expliquent en utilisant la représentation en base deux. `8 << 1` s'écrit en base deux `100 << 1 = 1000` , ce qui vaut 16 en base décimale. De même, `7&2` s'écrit `1011&10 = 10` , qui vaut 2 en base décimale.

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

```
x = int (3.5)
y = float (3)
print type(x), type(y)
```

Le programme précédent donne comme résultat :

```
<type 'int'> <type 'float'>
```

Les opérateurs listés par le tableau ci-dessus ont des priorités différentes, triés par ordre croissant. Toutefois, il est conseillé d'avoir recours aux parenthèses pour enlever les doutes : $3 * 2 * *4 = 3 * (2 * *4)$.

Remarque 2.2.2: division entière

```
x = 11
y = 2
z = x / y # le résultat est 5 et non 5.5 car la division est entière
```

L'opérateur // permet d'effectuer une division entière lorsque les deux nombres à diviser sont réels. Il existe néanmoins un autre cas pour lequel cet opérateur est implicite : lorsque la division opère sur deux nombres entiers ainsi que le montre l'exemple précédent. Pour obtenir le résultat juste incluant la partie décimale, il faut convertir les entiers en réels.

```
x = float (11)
y = float (2)
z = x / y # le résultat est 5.5 car c'est une division entre deux réels
```

2.2.3 Booléen

Les booléens sont le résultat d'opérations logiques et ont deux valeurs possibles : `True` ou `False` .

```
x = 4 < 5
print x
print not x
```

Le programme précédent a pour résultat :

```
True
False
```

Voici la liste des opérateurs qui s'appliquent aux booléens.

opérateur	**signification**	**exemple**
and or	et, ou logique	`x = True or False` (résultat = True)
not	négation logique	`x = not x`

Voici la liste des opérateurs de comparaisons qui retournent des booléens.

opérateur	**signification**	**exemple**
< >	inférieur, supérieur	$x = 5 < 5$
<= >=	inférieur ou égal, supérieur ou égal	$x = 5 <= 5$
== !=	égal, différent	$x = 5 == 5$

A l'instar des nombres réels, il est préférable d'utiliser les parenthèses pour éviter les problèmes de priorités d'opérateurs dans des expressions comme : $3 < x$ `and` $x < 7$. Toutefois, pour cet exemple, *Python*accepte l'écriture résumée qui enchaîne des comparaisons : $3 < x$ `and` $x < 7$ est équivalent à $3 < x < 7$. Il existe deux autres mots-clé qui retournent un résultat de type booléens :

opérateur	**signification**
is	test d'identification
in	test d'appartenance

Ces deux opérateurs seront utilisés ultérieurement, `in` lors des boucles (paragraphe 3.3), `is` lors de l'étude des classes (chapitre 4). Bien souvent, les booléens sont utilisés de manière implicite lors de tests (paragraphe 3.2).

2.2.4 Chaîne de caractères

2.2.4.1 Création d'une chaîne de caractères

> **Définition 2.2.3 : chaîne de caractères**
> Le terme "chaîne de caractères" ou *string* en anglais signifie une suite finie de caractères, autrement dit, du texte. Une chaîne de caractères est une variable contenant du texte.

Ce texte est compris entre deux guillemets ou deux apostrophes, ces deux symboles sont interchangeables. L'exemple suivant montre comment créer une chaîne de caractères.

```
# -*- coding: cp1252 -*-
t = "string = texte"
print type (t), t
t = 'string = texte, initialisation avec apostrophes'
print type (t), t

t = "morceau 1" \
   "morceau 2" # second morceau ajouté au premier,
                        # il ne doit rien y avoir d'espace après le symbole \
print t

t = """première ligne
seconde ligne""" # chaîne de caractères qui s'étend sur deux lignes
print t
```

Le résultat de ce petit programme est le suivant :

```
<type 'str'> string = texte
<type 'str'> string = texte, initialisation avec apostrophes
morceau 1morceau 2
première ligne
seconde ligne
```

La troisième chaîne de caractères créée lors de ce programme s'étend sur deux lignes. Il est parfois plus commode d'écrire du texte sur deux lignes plutôt que de le laisser caché par les limites de fenêtres d'affichages. *Python*offre la possibilité de couper le texte en deux chaînes de caractères recollées à l'aide du symbole \ à condition que ce symbole soit le dernier de la ligne sur laquelle il apparaît. De même, lorsque le texte contient plusieurs lignes, il suffit de les encadrer entre deux symboles """ ou ''' pour que l'interpréteur *Python*considère l'ensemble comme une chaîne de caractères et non comme une série d'instructions.

Par défaut, le *Python*ne permet pas l'insertion de caractères tels que les accents dans les chaînes de caractères, il faut pour cela ajouter le commentaire suivant $\#-*-$ `coding : cp1252` $-*-$ à la première ligne du programme. De même, pour insérer un guillemet dans une chaîne de caractères encadrée elle-même par des guillemets, il faut le faire précéder du symbole \. La séquence \" est appelée un extra-caractère (voir table 2.1).

\"	guillemet
\'	apostrophe
\n	passage à la ligne
\\	insertion du symbole \
\%	pourcentage, ce symbole est aussi un caractère spécial
\t	tabulation

TAB. 2.1 – *Liste des des extra-caractères les plus couramment utilisés à l'intérieur d'une chaîne de caractères.*

2.2.4.2 Manipulation d'une chaîne de caractères

Une chaîne de caractères est semblable à un tableau et certains opérateurs qui s'appliquent aux tableaux s'appliquent également aux chaînes de caractères, celles-ci sont regroupées dans la table 2.2. La fonction `str` permet de convertir un nombre, un tableau, un objet (voir chapitre 4) en chaîne de caractères afin de pouvoir l'afficher. La fonction `len` retourne la longueur de la chaîne de caractères.

```
x = 5.567
s = str (x)
print type(s), s   # <type 'str'> 5.567
print len(s)  # affiche 5
```

opérateur	signification	exemple
+	concaténation de chaînes de caractères	t = "abc" + "def"
[n]	obtention du $n^{ième}$ caractères, le premier caractère a pour indice 0	t = "abc" print t[0] # donne a
[i : j]	obtention des caractères compris entre les indices i et j − 1 inclus, le premier caractère a pour indice 0	t = "abc" print t[0 : 1] # donne ab

TAB. 2.2 – *Opérations applicables aux chaînes de caractères.*

Il existe d'autres fonctions qui permettent de manipuler les chaînes de caractères. Elles suivent la syntaxe des classes (voir chapitre 4) :

$$\texttt{res} = \texttt{s.fonction}(...)$$

Où `s` est une chaîne de caractères, `fonction` est le nom de l'opération que l'on veut appliquer à `s` , `res` est le résultat de cette manipulation.

La table 2.3 présente une liste non exhaustive des fonctions disponibles dont un exemple d'utilisation suit.

```
st = "langage python"
st = st.upper () # mise en lettres majuscules
i  = st.find ("PYTHON") # on cherche "PYTHON" dans st
print i # affiche 8
print st.count ("PYTHON")  # affiche 1
print st.count ("PYTHON", 9)  # affiche 0
```

count(sub[,start[,end]])	Retourne le nombre d'occurences de la chaîne de caractères sub , les paramètres par défaut start et end permet de réduit la recherche entre les caractères d'indice start et end exclus. Par défaut, start est nul tandis que end corresond à la fin de la chaîne de caractères.
find(sub[,start[,end]])	Recherche une chaîne de caractères sub , les paramètres par défaut start et end ont la même signication que ceux de la fonction count . Cette fonction retourne -1 si la recherche n'a pas abouti.
isalpha()	Recherche True si tous les caractères sont des lettres, False sinon.
isdigit()	Recherche True si tous les caractères sont des chiffres, False sinon.
replace(old,new[,count])	Retourne une copie de la chaîne de caractères en remplaçant toutes les occurrences de la chaîne sub par new . Si le paramètre optionnel count est renseigné, alors seules les count premières occurrences seront remplacées.
split([sep[,maxsplit]])	Découpe la chaîne de caractères en se servant de la chaîne split comme délimiteur. Si le paramètre maxsplit est renseigné, au plus maxsplit coupures seront effectuées.
upper()	remplace les minuscules par des majuscules
lower()	remplace les majuscules par des minuscules

TAB. 2.3 – *Quelques fonctions s'appliquant aux chaînes de caractères, l'aide associée au langage Pythonfournira la liste complète. Certains des paramètres sont encadrés par des crochets, ceci signifie qu'ils sont facultatifs.*

2.2.4.3 Formatage d'une chaîne de caractères

*Python*offre une manière plus concise de former une chaîne de caractères à l'aide de plusieurs types d'informations en évitant la conversion explicite de ces informations (fonctions str) et leur concaténation. Le format est le suivant :

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

c1 est un code choisi parmi ceux de la table 2.4 (page 21). Il indique le format dans lequel la variable v1 devra être transcrite. Il en est de même pour le code c2 associé à la variable v2 . Les codes insérés dans la chaîne de caractères seront remplacés par les variables citées entre parenthèses après le symbole % suivant la fin de la chaîne de caractères. Il doit y avoir autant de codes que de variables, qui peuvent aussi être des constantes.

Voici concrètement l'utilisation de cette syntaxe :

```
# -*- coding: cp1252 -*-
x = 5.5
d = 7
s = "caractères"
res =   "un nombre réel %f et un entier %d, une chaîne de %s, \n" \
        "un réel d'abord converti en chaîne de caractères %s" % (x,d,s, str(x+4))
print res
res =   "un nombre réel " + str (x) + " et un entier " + str (d) + \
", une chaîne de " + s + \
```

```
        ",\n un réel d'abord converti en chaîne de caractères " + str(x+4)
print res
```

La seconde affectation de la variable `res` propose une solution équivalente à la première en utilisant l'opérateur de concaténation `+` . Les deux solutions sont équivalentes, tout dépend des préférences de celui qui écrit le programme.

```
un nombre réel 5.500000 et un entier 7, une chaîne de caractères,
un réel d'abord converti en chaîne de caractères 9.5
un nombre réel 5.5 et un entier 7, une chaîne de caractères,
un réel d'abord converti en chaîne de caractères 9.5
```

La première option permet néanmoins un formatage plus précis des nombres réels en imposant par exemple un nombre défini de décimal. Le format est le suivant :

"%`n.df`"%`x`

où `n` est le nombre de chiffres total et `d` est le nombre de décimales

Exemple :

```
x = 0.123456789
print x
print "%1.2f" % x
print "%6.2f" % x
```

Ce qui donne :

```
0.123456789
0.12
  0.12
```

Il existe d'autres formats regroupés dans la table 2.4. L'aide reste encore le meilleur réflexe pour le langage *Python*susceptible d'évoluer et d'ajouter de nouveaux formats.

d	entier relatif
e	nombre réel au format exponentiel
f	nombre réel au format décimal
g	nombre réel, format décimal ou exponentiel si la puissance est trop grande ou trop petite
s	chaîne de caractères

TAB. *2.4 – Liste non exhaustive des codes utilisés pour formater des informations dans une chaîne de caractères.*

2.2.5 T-uple

Définition 2.2.4 : T-uples
Les T-uples sont un tableau d'objets qui peuvent être de tout type. Ils ne sont pas modifiables.

Un T-uple apparaît comme une liste d'objets comprise entre parenthèses et séparées par des virgules. Leur création reprend le même format :

```
x = (4,5)                # création d'un T-uple composé de deux entiers
x = ("un",1,"deux",2)    # création d'un T-uple composé deux chaînes de caractères
                         # et de deux entiers, l'ordre d'écriture est important
x = (3,)                 # création d'un T-uple d'un élément, sans la virgule,
                         # le résultat est entier
```

Ces objets sont des vecteurs d'objets. Il est possible d'effectuer les opérations regroupées dans la table 2.5. Etant donnée que les chaînes de caractères sont également des tableaux, ces opérations reprennent en partie celles de la table 2.2. Quelques fonctionnalités sont néanmoins ajoutées par rapport à celles regroupées dans la table 2.5.

`x in s`	vrai si `x` est un des éléments de `s`
`x not in s`	réciproque de la ligne précédente
`s + t`	concaténation de `s` et `t`
`s * n` , `n * s`	concatène `n` copies de `s` les unes à la suite des autres
`s[i]`	retourne une copie du $i^{\text{ème}}$ élément de `s`
`s[i : j]`	retourne un T-uple contenant une copie des éléments de `s` d'indices i à j exclu.
`s[i : j : k]`	retourne un T-uple contenant une copie des éléments de `s` dont les indices sont compris entre i et j exclu, ces indices sont espacés de k : i,$i+k$,$i+2k$,$i+3k$,...
`len(s)`	nombre d'éléments de `s`
`min(s)`	plus petit élément de `s` , résultat difficile à prévoir lorsque les types des éléments sont différents
`max(s)`	plus grand élément de `s` , résultat difficile à prévoir lorsque les types des éléments sont différents
`sum(s)`	retourne la somme de tous les éléments

TAB. 2.5 – *Opérations disponibles sur les Tuples, on suppose que* `s` *et* `t` *sont des T-uples,* `x` *est quant à lui quelconque.*

Remarque 2.2.5: T-uple, opérateur []

Les T-uples ne sont pas modifiables, cela signifie qu'il est possible de modifier un de leurs éléments. Par conséquent, la ligne d'affectation suivante n'est pas correcte :

```
a = (4,5)
a [0] = 3   # déclenche une erreur d'exécution
```

Le messsage d'erreur suivant apparaît :

```
Traceback (most recent call last):
  File "<pyshell#78>", line 1, in -toplevel-
    a[0]=3
TypeError: object doesn't support item assignment
```

Pour changer cet élément, il est possible de s'y prendre de la manière suivante :

```
a = (4,5)
a = (3,) + a[1:2]  # crée un T-uple d'un élément concaténé avec la partie inchangée de a
```

2.2.6 Autres types

Il existe d'autres types comme le type `complex` permettant de représenter les nombres complexes. Ce type numérique suit les mêmes règles et fonctionne avec les mêmes opérateurs (excepté les opérateurs de comparaisons) que ceux présentés au paragraphe 2.2.2 et décrivant les nombres.

Il est impossible de dresser une liste exhaustive de ces types puisque le langage *Python*offre la possibilité de créer ses propres types (voir le chapitre 4, notamment le mot-clé `__slots__`). La plupart seront introduits par les différents modules qui seront intégrés au programme (voir chapitre 5).

2.3 Types modifiables (ou *mutable*)

2.3.1 Liste

2.3.1.1 Définition et fonctions

> **Définition 2.3.1 : liste**
> Les listes sont des collections d'objets qui peuvent être de tout type. Elles sont modifiables.

Une liste apparaît comme une succession d'objets comprise entre crochets et séparées par des virgules. Leur création reprend le même format :

```
x = [4,5]               # création d'une liste composée de deux entiers
x = ["un",1,"deux",2]   # création d'une liste composée deux chaînes de caractères
                        # et de deux entiers, l'ordre d'écriture est important
x = [3,]                # création d'une liste d'un élément, sans la virgule,
                        # le résultat reste une liste
x = [ ]                 # crée une liste vide
x = list ()             # crée une liste vide
```

Ces objets sont des listes chaînées d'objets. Il est possible d'effectuer les opérations regroupées dans la table 2.6. Ces opérations reprennent celle des T-uples (voir table 2.6) et incluent d'autres fonctionnalités puisque les listes sont modifiables (voir table 2.7). Il est donc possible d'insérer, de supprimer des éléments, de les trier. La syntaxe des opérations sur les listes que présentent la table 2.7 nécessitent la connaissance des classes présentées au chapitre 4.

2.3.1.2 Exemples

L'exemple suivant montre une utilisation de la méthode `sort` .

```
# -*- coding: cp1252 -*-
x = [9,0,3,5,4,7,8]   # définition d'une liste
print x               # affiche cette liste
x.sort ()             # trie la liste par ordre croissant
print x               # affiche la liste triée

def compare (x,y):           # crée une fonction
    if x > y : return -1     # qui retourne -1 si x<y,
```

`x in s`	vrai si `x` est un des éléments de `l`
`x not in s`	réciproque de la ligne précédente
`l + t`	concaténation de `l` et `t`
`l * n , n * s`	concatène `n` copies de `l` les unes à la suite des autres
`l[i]`	retourne l'élément $i^{\text{ème}}$ élément de `l` , à la différence des T-uples, l'instruction `l[i] = "3"` est valide, elle remplace l'élément `i` par 3.
`l[i : j]`	retourne une liste contenant les éléments de `l` d'indices i à j exclu. Il est possible de remplacer cette sous-liste par une autre en utilisant l'affectation `l[i : j] = l2` où `l2` est une autre liste (ou un T-uple) de dimension différente ou égale.
`l[i : j : k]`	retourne une liste contenant les éléments de `l` dont les indices sont compris entre i et j exclu, ces indices sont espacés de k : i,$i+k$,$i+2k$,$i+3k$,... Ici encore, il est possible d'écrire l'affectation suivante : `l[i : j : k] = l2` mais `l2` doit être une liste (ou un T-uple) de même dimension que `l[i : j : k]` .
`len(l)`	nombre d'éléments de `l`
`min(l)`	plus petit élément de `l` , résultat difficile à prévoir lorsque les types des éléments sont différents
`max(l)`	plus grand élément de `l` , résultat difficile à prévoir lorsque les types des éléments sont différents
`sum(l)`	retourne la somme de tous les éléments
`del l[i : j]`	supprime les éléments d'indices entre `i` et `j` exclu. Cette instruction est équivalente à `l[i : j] = []` .
`list(x)`	convertit `x` en une liste quand cela est possible

TAB. 2.6 – *Opérations disponibles sur les listes, identiques à celles des T-uples, on suppose que* `l` *et* `t` *sont des listes,* `i` *et* `j` *sont des entiers.* `x` *est quant à lui quelconque.*

```
    elif x == y : return 0   # 0 si x == y
    else: return 1           # 1 si x < y

x.sort (compare)      # trie la liste x à l'aide de la fonction compare
                      # cela revient à la trier par ordre décroissant
print x
```

Voici les trois listes affichées par cet exemple :

```
[9, 0, 3, 5, 4, 7, 8]
[0, 3, 4, 5, 7, 8, 9]
[9, 8, 7, 5, 4, 3, 0]
```

L'exemple suivant illustre un exemple dans lequel on essaye d'accéder à l'indice d'un élément qui n'existe pas dans la liste :

```
x = [9,0,3,5,0]
print x.index(1)
```

`l.count(x)`	Retourne le nombre d'occurrences de l'élément `x`. Cette notation suit la syntaxe des classes développée au chapitre `chap_classe`. `count` est une méthode de la classe `list`.
`l.index(x)`	Retourne l'indice de la première occurrence de l'élément `x` dans la liste `l`. Si celle-ci n'existe, une exception est déclenchée (voir le paragraphe 4.1).
`l.append(x)`	Ajoute l'élément `x` à la fin de la liste `l`. Si `x` est une liste, cette fonction ajoute la liste `x` en tant qu'élément, au final, la liste `l` ne contiendra qu'un élément de plus.
`l.extend(k)`	Ajoute tous les éléments de la liste `k` à la liste `l`. La liste `l` aura autant d'éléments supplémentaires qu'il y en a dans la liste `k`.
`l.insert(i,x)`	Insère l'élément `x` à la position `i` dans la liste `l`.
`l.remove(x)`	Supprime la première occurence de l'élément `x` dans la liste `l`. S'il n'y a aucune occurrence de `x`, cette méthode déclenche une exception.
`l.pop([i])`	Retourne l'élément `l[i]` et le supprime de la liste. Le paramètre `i` est facultatif, s'il n'est pas précisé, c'est le dernier élément dont la valeur est d'abord retournée puis il est supprimé de la liste.
`l.reverse(x)`	Retourne la liste, le premier et dernier élément échange leurs places, le second et l'avant dernier, et ainsi de suite.
`l.sort([f])`	Cette fonction trie la liste par ordre croissant. Le paramètre `f` est facultatif, il permet de préciser la fonction de comparaison qui doit être utilisée lors du tri. Cette fonction prend comme paramètre deux éléments `x` et `y` de la liste et retourne les valeurs -1,0,1 selon que $x < y$, $x == y$ ou $x > y$ (voir paragraphe 3.4).

TAB. 2.7 – *Opérations permettant de modifier une listes on suppose que* `l` *est une liste,* `x` *est quant à lui quelconque.*

Comme cet élément n'existe pas, il déclenche ce qu'on appelle une exception qui se traduit par l'affichage d'un message d'erreur. Le message indique le nom de l'exception générée (`ValueError`) ainsi qu'un message d'information permettant en règle générale de connaître l'événement qui en est la cause.

```
Traceback (most recent call last):
  File "c:/temp/temp", line 2, in -toplevel-
    print x.index(1)
ValueError: list.index(x): x not in list
```

Pour éviter cela, on choisit d'intercepter l'exception (voir paragraphe 4.1).

```
# -*- coding: cp1252 -*-
x = [9,0,3,5,0]
try:
    print x.index(1)
except ValueError:
    print "1 n'est pas présent dans la liste x"
else:
    print "ça marche"
```

Ce programme a pour résultat :

```
1 n'est pas présent dans la liste x
```

2.3.1.3 Boucles, fonction `range`

Les listes sont souvent utilisés dans des boucles ou notamment par l'intermédiaire de la fonction **range** . Cette fonction retourne une liste d'entiers.

range (**debut**, **fin** [,**marche**])

Retourne une liste incluant tous les entiers compris entre **debut** et **fin** inclus. Si le paramètres facultatifs **marche** est renseigné, la liste contient tous les entiers **n** compris **debut** et **fin** exclu et tels que $\mathbf{n} - \mathbf{debut}$ soit un multiple de **marche** .

Exemple :

```
print range(0,10,2)  # affiche [0, 2, 4, 6, 8]
```

La fonction **xrange** est équivalente à **range** à ceci près qu'elle utilise moins de mémoire. En contrepartie, cette opacité est reflétée par l'impossibilité d'afficher le résultat de la fonction **xrange** sous forme de liste.

```
print xrange(0,10,2)  # affiche xrange(0,10,2)
```

Cette fonction est souvent utilisée lors de boucles (voir paragraphe 3.3). Pour parcourir tous les éléments d'un T-uple, d'une liste, d'un dictionnaire... Le programme suivant permet par exemple de calculer la somme de tous les entiers compris entre 1 et 20 exclu.

```
s = 0
for n in range(1,20):      # ce programme est équivalent à
    s += n                 # s = sum (range(1,20))
```

Le programme suivant permet d'afficher tous les éléments d'une liste.

```
x = ["un", 1, "deux", 2, "trois", 3]
for n in range(0,len(x)):
    print "x [%d] = %s" % (n, x [n])
```

Affiche :

```
x [0] = un
x [1] = 1
x [2] = deux
x [3] = 2
x [4] = trois
x [5] = 3
```

Il est possible aussi de ne pas se servir des indices comme intermédiaires pour accéder aux éléments d'une liste. Il s'agit d'effectuer un même traitement pour tous les éléments de la liste `x` .

```
x = ["un", 1, "deux", 2, "trois", 3]
for el in x :
    print "la liste inclut : ", el
```

Ce programme a pour résultat :

```
la liste inclut :  un
la liste inclut :  1
la liste inclut :  deux
la liste inclut :  2
la liste inclut :  trois
la liste inclut :  3
```

2.3.2 Dictionnaire

2.3.2.1 Définition et fonctions

> **Définition 2.3.2 : dictionnaire**
> Les dictionnaires sont des listes de couples. Chaque couple contient une clé et une valeur. Chaque valeur est indicée par sa clé. La valeur peut-être de tout type, la clé doit être de type immuable, ce ne peut donc être ni une liste, ni un dictionnaire. Chaque clé comme chaque valeur peut avoir un type différent des autres clés ou valeurs.

Un dictionnaire apparaît comme une succession de couples d'objets comprise entre accolades et séparées par des virgules. La clé et sa valeur sont séparées par le symbole `:` . Leur création reprend le même format :

```
x = { "cle1":"valeur1", "cle2":"valeur2" }
y = { }         # crée un dictionnaire vide
z = dict ()     # crée aussi un dictionnaire vide
```

Les indices ne sont plus entiers mais pour cet exemple des chaînes de caractères. Pour associer la valeur associé à la clé "cle1", il suffit d'écrire :

```
print x ["cle1"]
```

La plupart des fonctions disponibles pour les listes sont interdites pour les dictionnaires comme la concaténation ou l'opération de multiplication ($*$). Il n'existe plus non plus d'indices pour repérer les éléments, le seul repère est leur clé. La table 2.8 dresse la liste des opérations simples sur les dictionnaires tandis que la table 2.9 dresse la liste des méthodes plus complexes associées aux dictionnaires.

Contrairement à une liste, un dictionnaire ne peut être trié car sa structure interne est optimisée pour effectuer des recherches rapides parmi les éléments. On peut aussi se demander quel est l'intérêt de la méthode `popitem` qui retourne un élément puis le supprime alors qu'il existe la fonction `del` . Cette méthode est simplement plus rapide car elle choisit à chaque fois l'élément le moins coûteux à supprimer, surtout lorsque le dictionnaire est volumineux.

Les itérateurs sont des objets qui permettent de parcourir rapidement un dictionnaire à condition que celui-ci ne soit pas modifié lors du parcours. Un exemple de leur utilisation est présenté dans le paragraphe suivant.

`x in d`	vrai si `x` est une des clés de `d`
`x not in d`	réciproque de la ligne précédente
`l[i]`	retourne l'élément associé à la clé `i`
`len(d)`	nombre d'éléments de `d`
`min(d)`	plus petite clé
`max(d)`	plus grande clé
`del l[i]`	supprime l'élément associée à la clé `i`
`list(d)`	retourne une liste contenant toutes les clés du dictionnaire `d` .
`dict(x)`	convertit `x` en un dictionnaire si cela est possible, particulier, `d` est égal à `dict(d.items())`

TAB. 2.8 – *Opérations disponibles sur les dictionnaires,* `d` *est un dictionnaire,* `x` *est quant à lui quelconque.*

`d.copy()`	Retourne une copie de `d` .
`d.has_key(x)`	Retourne `True` si `x` est une clé de `d` .
`d.items()`	Retourne une liste contenant tous les couples (clé, valeur) inclus dans le dictionnaire.
`d.keys()`	Retourne une liste contenant toutes les clés du dictionnaire `d` .
`d.values()`	Retourne une liste contenant toutes les valeurs du dictionnaire `d` .
`d.iteritems()`	Retourne un itérateur sur les couples (clé, valeur).
`d.iterkeys()`	Retourne un itérateur sur les clés.
`d.itervalues()`	Retourne un itérateur sur les valeurs.
`d.get(k[,x])`	Retourne `d[k]` , si la clé `k` est manquante, alors la valeur `None` est retourné à moins que le paramètre optionnel `x` soit renseigné, auquel cas, ce sera ce paramètres qui sera retourné.
`d.clear()`	Supprime tous les éléments du dictionnaire.
`d.update(d2)`	Pour chaque clé de `d1` , `d[k]` = `d1[k]`
`d.setdefault(k[,x])`	Retourne `d[k]` si la clé `k` existe, sinon, affecte `x` à `d[k]` .
`d.popitem()`	Retourne un éléments et le supprime du dictionnaire.

TAB. 2.9 – *Méthodes associées aux dictionnaires,* `d` , `d2` *sont des dictionnaires,* `x` *est quant à lui quelconque.*

2.3.2.2 Exemples

Il n'est pas possible de trier un dictionnaire. L'exemple suivant permet néanmoins d'afficher tous les éléments d'un dictionnaire selon un ordre croissant des clés. Ces exemples font appel aux paragraphes sur les boucles (voir chapitre 3).

```
# -*- coding: cp1252 -*-
x = { "un":1, "zéro":0, "deux":2, "trois":3, "quatre":4, "cinq":5, "six":6, "sept":1, \
       "huit":8, "neuf":9, "dix":10 }
key = d.keys ()
key.sort ()
for k in key:
    print k,d [k]
```

L'exemple suivant montre un exemple d'utilisation des itérateurs. Il s'agit de construire un dictionnaire inversé pour lequel les valeurs seront les clés et réciproquement.

```
# -*- coding: cp1252 -*-
d = { "un":1, "zéro":0, "deux":2, "trois":3, "quatre":4, "cinq":5, "six":6, "sept":1, \
       "huit":8, "neuf":9, "dix":10 }

dinv = dict ()
it = d.iteritems ()     # création d'un itérateur sur les couples (clé, valeur)
while True:             # ne change jamais lors d'un parcours avec itérateur
    try :               # ne change jamais lors d'un parcours avec itérateur,
                        # permet d'attraper une exception
        x = it.next ()  # affecte à x le couple (clé, valeur) désigné par l'itérateur
        dinv [ x [1] ] = x [0]
    except StopIteration :  # attrape l'exception qui signifie que tous les éléments
                            # du dictionnaire ont été visités
        break               # on sort donc de la boucle

print d                 # affiche le dictionnaire
print dinv              # affiche le dictionnaire inversé
```

2.4 Extensions

2.4.1 Mot-clé `print` , `repr` , conversion en chaîne de caractères

Le mot-clé **print** est déjà apparu dans les exemples présentés ci-dessus, il permet d'afficher une ou plusieurs variables préalablement définies, séparées par des virgules. Les paragraphes qui suivent donnent quelques exemples d'utilisation. La fonction **print** permet d'afficher n'importe quelle variable ou objet à l'écran, cet affichage suppose la conversion de cette variable ou objet en une chaîne de caractères. Deux fonctions permettent d'effectuer cette étape sans toutefois afficher le résultat à l'écran.

La fonction **str** (voir paragraphe 2.2.4.2) permet de convertir toute variable en chaîne de caractères. Il existe cependant une autre fonction **repr** , qui effectue cette conversion. Dans ce cas, le résultat est interprétable par la fonction **eval** (voir paragraphe 2.4.2) qui se charge de la conversion inverse. Pour les types simples comme ceux présentés dans ce chapitre, ces deux fonctions retournent des résultats identiques. Pour l'exemple, **x** désigne n'importe quelle variable.

```
x == eval (repr(x)) # est toujours vrai (True)
x == eval (str (x)) # n'est pas toujours vrai
```

Le module **pprint** inclut une fonction du même nom proposant des affichages plus lisibles des types comme les listes ou les dictionnaires. L'instruction **print** affiche les éléments d'une liste les uns à la suite des autres. Lorsque cette présentation devient illisible, la fonction **pprint** affiche un élément par ligne.

```
import pprint
x = ["un","deux","trois"]
x = x*4
print x
pprint.pprint (x)
```

Ce programme a pour résultat :

```
['un', 'deux', 'trois', 'un', 'deux', 'trois', 'un', 'deux
', 'trois', 'un', 'deux', 'trois']
['un',
 'deux',
 'trois',
 'un',
 'deux',
 'trois',
 'un',
 'deux',
 'trois',
 'un',
 'deux',
 'trois']
```

2.4.2 Informations fournies par *Python*

Bien que les fonctions ne soient définies que plus tard (paragraphe 3.7), il peut être intéressant de mentionner la fonction `dir` qui retourne la liste de toutes les variables créées et accessibles à cet instant du programme. L'exemple suivant :

```
x = 3
print dir ()
```

retourne le résultat suivant :

```
['__builtins__', '', '', '', 'x']
```

Certaines variables, des chaînes des caractères existent déjà avant même la première instruction :

`__builtins__`	utilisée lors de la compilation d'un code défini lors de l'exécution du programme
`__doc__`	chaîne commentant le fichier, c'est une chaîne de caractères insérée aux premières du fichiers et entourés des symboles """
`__file__`	contient le nom du fichier où est écrit ce programme
`__name__`	contient le nom du module

La fonction `dir` est également pratique pour afficher toutes les fonctions d'un module. L'instruction `dir(sys)` affiche la liste des fonctions du module `sys` (voir chapitre 5).

De la même manière, la fonction `type` retourne une information concernant le type d'une variable.

```
x = 3
print x, type(x)
x = 3.5
print x, type(x)
```

L'exemple précédent aboutit au résultat suivant :

```
3 <type 'int'>
3.5 <type 'float'>
```

Autre fonction intérassant, la fonction `eval` permet d'évaluer une chaîne de caractères. Le petit exemple suivant permet d'afficher le contenu de toutes les variables définies à cet instant du programme.

```
""" affiche toutes les variables """
x = 3
x = x ** eval ("x")
for s in dir ():
  print s, "  =  ", eval (s)
```

Ceci aboutit au résultat suivant :

```
__builtins__   =   <module '__builtin__' (built-in)>
__doc__   =    affiche toutes les variables
__file__   =   C:\temp\essai2.py
__name__   =   __main__
x   =   27
```

La première ligne du programme est une chaîne de caractères. Elle est facultative mais sa présence est considérée comme un commentaire associé au fichier contenant le programme. Cette convention est utile lors de la conception de modules (voir chapitre 5).

2.4.3 Affectations multiples

Il est possible d'effectuer en *Python*plusieurs affectations simultanément.

```
x = 5       # affecte 5 à x
y = 6       # affecte 6 à y
x,y = 5,6   # affecte en une seule instruction 5 à x et 6 à y
```

Cette particularité reviendra lorsque les fonctions seront décrites puisqu'il est possible qu'une fonction retourne plusieurs résultat comme la fonction `divmod` illustré par le programme suivant.

```
x,y = divmod (17,5)
print x,y                          # affiche 3 2
print "17 / 5 = 5 * ", x, " + ",y  # affiche 17 / 5 = 5 *  3  +  2
```

Chapitre 3

Syntaxe du langage *Python*

3.1 Les trois concepts des algorithmes

Les algorithmes sont composées d'instructions, ce sont des opérations élémentaires que le processeur exécute selon trois schémas :

la séquence	enchaînement des instructions les unes à la suite des autres : passage d'une instruction à la suivante
le saut	passage d'une instruction à une autre qui n'est pas forcément la suivante (c'est une rupture de séquence)
le test	choix entre deux instructions

Le saut n'apparaît plus de manière explicite dans les langages évolués car il est une source fréquente d'erreurs. Il intervient dorénavant de manière implicite au travers des boucles qui combinent un saut et un test comme le montre l'exemple suivant :

Version avec boucles :

```
initialisation de la variable moy à 0
faire pour i allant de 1 à N
      moy reçoit moy + ni
moy reçoit moy / N
```

Version équivalente avec sauts :

```
ligne 1 : initialisation de la variable moy à 0
ligne 2 : initialisation de la variable i à 1
ligne 3 : moy reçoit moy + ni
ligne 4 : i reçoit i + 1
ligne 5 : si i est inférieur ou égal à N alors aller à la ligne 3
ligne 6 : moy reçoit moy / N
```

Si tout programme peut se résumer à ces trois concepts, l'aisance que l'on peut acquérir dans tel ou tel langage dépend de la syntaxe qui les met en place. Cette syntaxe permet de transcrire des algorithmes rédigés en français comme celui de l'exemple ci-dessus pour aboutir au programme suivant, écrit en *Python*et interprétable par l'ordinateur via le compilateur associé à ce langage *Python*.

```
moy = 0
```

```
for i in range(1,N):
  moy += n [i]
moy /= N
```

Le premier élément de cette syntaxe est constitué de ses mots-clé (voir table 3.1) et des symboles (voir table 3.2) dont certains interviennent lors de l'écriture des tests, des boucles et des fonctions.

3.2 Tests

> **Définition 3.2.1 : test**
> Les tests permettent d'exécuter des instructions différentes selon la valeur d'une condition logique.

3.2.1 Syntaxe

```
if condition1 :
 instruction1
 instruction2
 ...
else :
 instruction3
 instruction4
 ...
```

S'il est nécessaire d'enchaîner plusieurs tests d'affilée, il est possible de condenser l'écriture :

```
if condition1 :
 instruction1
 instruction2
 ...
elif condition2 :
 instruction3
 instruction4
 ...
elif condition3 :
 instruction5
 instruction6
```

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

TAB. *3.1 – Mots-clé du langage Python.*

```
 ...
else :
 instruction7
 instruction8
 ...
```

3.2.2 Comparaisons possibles

Les comparaisons possibles entre deux entités sont avant tout numériques mais ces opérateurs peuvent être définis pour tout type (voir chapitre 4), notamment sur les chaînes de caractères pour lesquels les opérateurs de comparaison transcrivent l'ordre alphabétique.

< , >	inférieur, supérieur
<= , >=	inférieur ou égal, supérieur ou égal
== , !=	égal, différent
`is` , `is not`	`x is y` vérifie que `x` et `y` sont égaux, `isnot` , différents
`in` , `not it`	appartient, n'appartient pas

3.2.3 Opérateurs logiques

Il existe trois opérateurs logiques qui combinent entre eux les conditions.

`not`	négation
`and`	et logique
`or`	ou logique

3.2.4 Ecriture condensée

Il existe deux écritures de tests condensée. La première consiste à écrire un test et l'unique instruction qui en dépend sur une seule ligne.

```
if condition:
    instruction1
else:
    instruction2
```

=	+	-	*	/
:	+=	-=	*=	/=
#	**	"	'	//
%	~	"""	"'	\
[	]	(	)	,
<	>	<=	>=	.
^	!	==	!=	&
«	»	|	{	}

TAB. 3.2 – *Symbole du langage Python, certains ont plusieurs usages comme : qui est utilisé à chaque test ou boucle et qui permet aussi de déterminer une plage d'indices dans un tableau.*

Ce code est équivalent à :

```
if condition: instruction1
else: instruction2
```

Le second cas d'écriture condensée concerne les comparaisons enchaînées. Le test `if 3 < x and x < 5 : instruction` peut être condensé par `if 3 < x < 5 : instruction` . Il est ainsi possible de juxtaposer autant de comparaisons que nécessaire : `if 3 < x < y < 5 : instruction` .

3.2.5 Exemple

L'exemple suivant associe à la variable `signe` le signe de la variable `x` .

```
x = -5
if x < 0 :
   signe = -1
elif x == 0 :
   signe = 0
   print "pas de signe"
else :
   signe = 1
```

Son écriture condensée lorsqu'il n'y a qu'une instruction à exécuter :

```
x = -5
if x < 0 : signe = -1
elif x == 0 :
   signe = 0
   print "pas de signe"
else : signe = 1
```

Le programme suivant saisit une ligne au clavier et dit si c'est "oui" ou "non" qui a été saisi.

```
s = raw_input ("dit oui : ")
if s == "oui" or s [0:1] == "o" or s [0:1] == "O" or s == "1" :
    print "oui"
else :
    print "non"
```

3.3 Boucles

3.4 Fonctions

`help`

portée

passage de paramètres

help

3.4.1 Syntaxe

3.4.2 Nombre de paramètres variables

3.4.3 Ecriture simplifiée pour des fonctions simples

lambda

3.5 Commentaires

Lorsque les commentaires incluent des symboles exclusivement français tels que les accents, le compilateur génère l'erreur suivante :

```
sys:1: DeprecationWarning: Non-ASCII character '\xe9' in file C:\temp\essai2.py on line 1,
but no encoding declared; see http://www.python.org/peps/pep-0263.html for details
```

Il est néanmoins possible d'utiliser des accents dans les commentaires à condition d'insérer le commentaire suivant à la première ligne du programme.

```
# -*- coding: cp1252 -*-
```

3.5.1 Appelable

La fonction `callable` retourne un booléen permettant de savoir si un identificateur est une fonction ou une classe (voir chapitre 4), de savoir par conséquent si tel identificateur est appelable comme une fonction.

```
x = 5
def y () :
    return None
print callable (x)  # affiche False car x est une variable
print callable (y)  # affiche True car y est une fonction
```

3.6 Indentation

parler du symbole \ qui permet d'écrire une instruction sur plusieurs lignes

3.7 Fonctions

Chapitre 4

Classes et exceptions

4.1 Exceptions

Chapitre 5

Modules

5.1 Extensions

5.1.1 Modules disponibles

Le petit exemple suivant permet d'obtenir la liste des modules qu'il est possible d'importer.

```
import sys
import pprint
pprint.pprint (sys.modules)
```

Voici le début de la liste.

```
{'ConfigParser': <module 'ConfigParser' from 'C:\Python23\lib\ConfigParser.pyc'>,
 'FixTk': <module 'FixTk' from 'C:\Python23\lib\lib-tk\FixTk.pyc'>,
 'Queue': <module 'Queue' from 'C:\Python23\lib\Queue.pyc'>,
 'SimpleDialog': <module 'SimpleDialog' from 'C:\Python23\lib\lib-tk\SimpleDialog.pyc'>,
 'SocketServer': <module 'SocketServer' from 'C:\Python23\lib\SocketServer.pyc'>,
 'StringIO': <module 'StringIO' from 'C:\Python23\lib\StringIO.pyc'>,
 'Tkconstants': <module 'Tkconstants' from 'C:\Python23\lib\lib-tk\Tkconstants.pyc'>,
 'Tkinter': <module 'Tkinter' from 'C:\Python23\lib\lib-tk\Tkinter.pyc'>,
 'UserDict': <module 'UserDict' from 'C:\Python23\lib\UserDict.pyc'>,
 ... # liste tronquée
```

Mais il est plus facile de vérifier la présence d'un module grâce aux quelques lignes suivantes.

```
import sys
print "math" in sys.modules  # retourne True normalement puisque le module
                             # math est installé par défaut
```

Chapitre 6

Licence du langage *Python*

Voici le contenu de la page [www-PyLicence].

6.1 HISTORY OF THE SOFTWARE

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see http://www.cwi.nl) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see http://www.cnri.reston.va.us) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation, see http://www.zope.com). In 2001, the Python Software Foundation (PSF, see http://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see http://www.opensource.org for the Open Source Definition).

6.2 TERMS AND CONDITIONS FOR ACCESSING OR OTHERWISE USING PYTHON

6.2.1 PSF LICENSE AGREEMENT FOR PYTHON 2.3

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.3 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.3 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright (c) 2001, 2002, 2003 Python Software Foundation; All Rights Reserved" are retained in Python 2.3 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.3 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.3.
4. PSF is making Python 2.3 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.3 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.3 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.3, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 2.3, Licensee agrees to be bound by the terms and conditions of this License Agreement.

6.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed

to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

6.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright (c) 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013".
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to

use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

6.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

Bibliographie

[Martelli2004] Alex Martelli, *Python en concentré*, O'Reilly, édition française (2004)

[www-DocF] , *documents rédigés en langue française*, http://www.developpez.com/cours/ (2004)

[www-GPL] , *site officiel de la licence GPL*, http://www.gnu.org/copyleft/gpl.html (2004)

[www-Python] , *site officiel du langage Python*, http://www.python.org (2004)

[www-PyDoc] , *site officiel du langage Python, documentation*, http://docs.python.org/download.html (2004)

[www-PyEdit] , *éditeurs de texte pour le langage Python*, http://www.python.org/cgi-bin/moinmoin/PythonEditors (2004)

[www-PyDownload] , *site officiel du langage Python, téléchargement du programma d'installation*, http://www.python.org/download/ (2004)

[www-PyLicence] , *site officiel du langage Python, licence*, http://www.python.org/psf/license.html (2004)

[www-PyModule] , *site officiel du langage Python, liste des modules existant*, http://www.python.org/pypi (2004)

[www-Syn] , *éditeur de texte Syn Text Editor*, http://syn.sourceforge.net/ (2004)

Table des figures

Liste des tableaux

Table des matières

Index