9

Le langage Python

Python est un langage mordant.
— Thierry Becker

Nous avons rapidement examiné, tout au long de notre apprentissage de Zope, certains concepts de programmation. Nous avons en particulier vu que Zope était fortement couplé à Python de par sa manière de manipuler les expressions dans le DTML, ou de traiter les boucles au sein des balises `<dtml-in>`, par exemple.

Le présent chapitre propose une analyse plus approfondie du langage Python. Ce chapitre ne fait appel à aucune notion propre à Zope : Python est un langage à part entière et peut être utilisé en dehors du cadre de Zope. Nous verrons au chapitre suivant comment écrire des programmes Python destinés à Zope.

Ce chapitre suppose que le lecteur connaît les bases de la programmation traditionnelle : variables, boucles, etc.

Présentation

Python est un langage dynamique et interprété, orienté objet. Il est possible d'écrire des programmes complets et complexes entièrement en Python (Zope, par exemple, est en très grande partie écrit en Python).

Python est un langage interprété comme Java. Ainsi, contrairement au langage C, les programmes écrits en Python n'ont pas à être compilés avant d'être exécutés.

Comme tout outil, Python présente des avantages et des inconvénients, souvent liés :

- Python est simple, comme nous allons le découvrir dans ce chapitre.
- Python est interprété. Les programmes Python sont donc plus faciles à mettre au point, peuvent être conçus, écrits et testés modulairement plus aisément que dans le cas des langages compilés. En outre, Python est l'un des langages interprétés les plus rapides. Cela dit, un programme interprété est plus lent qu'un programme compilé. De plus, un programme Python a besoin d'un interpréteur pour fonctionner.
- Python est riche : outre la richesse de ses fonctionnalités en tant que langage de programmation, Python est livré avec une riche bibliothèque de modules et de fonctions, permettant par exemple de créer un serveur web en à peine dix lignes de code…
- Python est un logiciel libre : il peut donc être téléchargé et redistribué gratuitement, et bénéficie en outre d'une abondante documentation et de listes de diffusion actives sur l'Internet (voir *http://www.python.org* pour plus d'informations).

Python est un langage original qui peut résoudre autant de problèmes que les « grands » langages comme Java, le C, le C++ ou d'autres, à la différence que – à notre avis – les programmes en Python sont naturellement mieux structurés, mieux conçus et plus faciles à maintenir que dans les langages que nous venons de citer.

Python et Zope

Nous avons parlé d'un couplage fort entre Python et Zope. Et ce pour trois raisons :

- Zope est principalement écrit en Python ;
- le DTML supporte la syntaxe de Python ;
- les composants de Zope sont naturellement écrits en Python.

Il est toutefois possible d'utiliser d'autres langages : ainsi, il existe actuellement un projet consistant à utiliser du Perl au lieu de Python pour certains scripts. Néanmoins, Python reste le langage naturel de Zope.

Il n'est pas nécessaire de connaître Python pour utiliser Zope : nous n'avons jusqu'à présent que très peu parlé de Python dans cet ouvrage. La connaissance de ce langage ouvre cependant de nouveaux horizons aux programmeurs, qui pourront personnaliser et étendre Zope grâce à des fonctions ou des modules écrits en Python. En outre, Python s'avère parfois plus efficace que le DTML pour traiter certains problèmes (en particulier, pour effectuer des traitements algorithmiques).

Attention

À ce jour, la version de Zope (2.3) utilise la version 1.5.2 de Python. Or, la dernière version de Python est la 2.1. Le présent chapitre couvre la version 1.5.2. Il n'est pas possible, à l'heure actuelle, d'utiliser d'une manière simple la version 2.1 de Python avec Zope. Cela deviendra possible dans les versions ultérieures de Zope.

Où est Python ?

Nous avons souligné que Python était un langage interprété. Aussi, pour que Zope s'exécute, il faut que Python soit présent sur l'ordinateur. Suivant le mode d'installation utilisé pour Zope (voir chapitre 1), l'exécutable `python` va se trouver soit directement dans votre chemin d'accès (sous Unix, il s'agit généralement de `/usr/local/bin/` ou `/usr/bin/`), soit dans le répertoire de Zope (`$ZOPE/bin/`).

Python peut fonctionner en mode interactif ou pas. En mode interactif, on peut entrer ses instructions une par une et Python les traite au fur et à mesure. En mode non interactif (ou mode *batch*, traitement par lots), on donne à l'exécutable `python` un fichier qui contient le programme à exécuter. Zope fonctionne de cette manière : le script `start` ou `start.bat` démarre l'interpréteur en mode non interactif, en lui donnant en paramètre l'emplacement du code source de Zope.

Pour utiliser les exemples de ce chapitre, on doit tout d'abord lancer l'exécutable `python` situé dans un des répertoires mentionnés plus haut. Lors du lancement de l'interpréteur Python, l'invite suivante apparaît, avec un curseur clignotant sur la dernière ligne :

```
Python 1.5.2 (#0, Jul 30 1999, 09:52:18) [MSC 32 bit (Intel)] on win32
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>>
```

Python fonctionne ici en mode interpréteur et attend impatiemment vos instructions. Pour quitter l'interpréteur, utiliser soit `Ctrl+D` sous Unix, soit `Ctrl+Z` (suivi d'un retour chariot) sous Windows.

Programmer en Python

L'instruction `print`

Le premier exemple, indispensable à l'apprentissage de Python, va nous permettre d'afficher à l'écran la fameuse phrase « `Hello, World !` ». Voici comment procéder :

```
>>> print "Hello, World !"
Hello, World !
>>>
```

Remarque

Dans les exemples de code présentés dans ce chapitre, nous reproduisons l'invite de Python (>>>) au début des lignes qui doivent être recopiées. Les lignes ne commençant pas par une invite ne doivent donc pas être recopiées, mais sont produites par la ligne de code du dessus.

Cette simple ligne appelle plusieurs remarques :

- En Python, *`print` est une instruction et non une fonction* ; nous verrons ultérieurement comment on utilise les fonctions.
- Pour afficher une chaîne de caractères, nous avons choisi de la placer entre guillemets. Donc, en Python (comme dans beaucoup de langages), *les chaînes doivent être placées entre des guillemets simples ou doubles* (*quotes* ; nous reviendrons sur cet aspect par la suite).
- La ligne ne se termine pas par un point-virgule (comme en C ou en Java) : en Python, *les instructions sont séparées par des sauts de lignes* ; de plus, Python oblige le programmeur à être rigoureux en matière d'indentation – nous reviendrons sur cet aspect.

> **Remarque**
>
> *En Python, l'indentation est obligatoire. Cette contrainte oblige les programmeurs à indenter correctement leurs programmes et contribue à rendre n'importe quel code source Python agréable à lire.*
>
> *Python accepte les indentations faites avec des tabulations ou des espaces. Si les deux sont mélangés dans un même code source, les tabulations sont considérées comme équivalentes à 8 espaces. Aussi la pratique courante consiste-t-elle à utiliser uniquement des espaces dans le code source, et à indenter chaque bloc avec 4 espaces : c'est la convention que nous utiliserons ici.*

L'instruction `print` permet d'afficher des chaînes mais aussi des nombres :

```
>>> print 123
123
```

Expressions

Expressions arithmétiques

Un langage de programmation a avant tout pour objet d'effectuer des calculs. La syntaxe de Python pour écrire des expressions est naturelle et n'effraiera pas les habitués des langages de programmation traditionnels. Voici quelques exemples :

```
>>> 1 + 2
3
>>> 3 + 4
7
>>> 1 + 2
3
>>> 3 * 4
12
>>> (2 + 4) * (5 - 8)
-18
>>> 5 / 10
0
```

La syntaxe elle-même n'appelle pas de commentaires particuliers. En revanche, nous remarquons que le résultat est affiché par l'interpréteur, bien que l'instruction `print` n'ait pas été utilisée. D'où la remarque suivante :

> **Remarque**
> *En mode interpréteur, Python affiche implicitement le résultat d'une expression entrée par l'utilisateur. En revanche, lorsqu'il exécute un fichier, Python n'affiche jamais implicitement ces résultats.*

On peut en faire l'expérience en plaçant toutes nos instructions dans un fichier `exemple.py` et en appelant Python comme suit :

```
python exemple.py
```

Types élémentaires

Python est un langage dynamiquement typé. Il ne comprend en fait que quelques types de données élémentaires, facilement convertibles entre eux. Par exemple, un type *entier* et un type *flottant*. Dans les expressions données ci-après, observez le dernier exemple :

```
>>> 5 / 10
0
```

La division devrait donner 0,5 et pas 0 ! En réalité, Python considère que `5` et `10` sont des entiers et retourne donc un résultat sous forme d'entier. Pour obtenir un résultat sous forme de flottant, il faut exprimer au moins un des opérandes sous forme de flottant, par exemple :

```
>>> 5.0 / 10
0.5
```

Dans ce cas, le type flottant étant plus précis que le type entier, Python renvoie un résultat de type flottant.

> **Remarque**
> *Lorsqu'il évalue une expression, Python renvoie un résultat dans le type le plus précis de tous les opérandes.*

On utilise usuellement les types de données simples suivants :

- entier signé sur 32 bits (par exemple : `123`, `112220`, `2147483647`[1], `-5`…).
- entier long : `2132165465465131321654L`, `153L`…

1. Il s'agit du plus grand nombre pouvant être représenté sous forme d'entier (entier signé sur 32 bits).

- flottant : `123.456`, `.1110`, `112312.0`, `50.`, `5.12e-10`...
- nombre imaginaire : `51j`, `12.05J`, `5.12e-10j`...
- chaîne de caractères : `"abc"`, `'Hello'`

Nous verrons qu'il est possible de représenter un entier, un flottant ou une chaîne de plusieurs façons ; nous détaillerons ces points dans la section « Constantes littérales ».

Remarque

Une chaîne de caractères peut être considérée comme une séquence de caractères, comme nous le verrons ultérieurement.

Les valeurs booléennes

Il existe enfin une dernière notion de type en Python : la valeur booléenne (vrai/faux).

Définition

Toute expression Python peut être considérée comme une valeur booléenne. Toute expression en Python est donc soit vraie, soit fausse.

Voici un tableau où sont récapitulés les cas de valeurs fausses pour chacun des types usuels de Python :

Type	Valeur fausse
Numérique (entier, entier long, flottant ou imaginaire)	`0`
Chaîne	`Chaîne vide ("")`
Séquence ou dictionnaire	`Séquence vide`

Nous reviendrons dans ce chapitre sur la définition d'une séquence.

Constantes littérales

Maintenant que nous avons découvert les principaux types de Python, il nous reste à étudier la manière de les exprimer. Nous avons vu, par exemple, qu'un flottant s'exprime avec un « . » entre la partie entière et la partie décimale.

Le tableau suivant présente la liste exhaustive des moyens dont on dispose pour exprimer une valeur de chaque type élémentaire en Python (on parle de « constante littérale » lorsqu'une valeur est exprimée telle quelle dans un programme);

Le format des chaînes dans Python est suffisamment souple pour faire face à toutes les situations possibles.

Type	Expression	Exemple
Entier (décimal)	*Chiffres de la base 10, précédés éventuellement du signe –[a]. Le nombre ne doit pas commencer par un 0 et doit être compris entre* `-2147483638` *et* `2147483637`*.*	`123456` `-123546`
Entier (octal)	*Nombre en base 8 commençant par un 0 et éventuellement précédé du signe –.*	`0123 (83 en décimal)` `-012 (-10 en décimal)`
Entier (hexadécimal)	*Nombre en base 16 précédé du préfixe* `0x` *et éventuellement précédé de* `-`*.* *Les lettres peuvent être en majuscules ou en minuscules.*	`0xABCD (43981 en décimal)` `-0xABCD (-43981 en décimal)`

Entier long	*Entier (décimal, octal ou hexadécimal) suivi de la lettre* `L` *(minuscule ou majuscule).*	`12344545743534834L`
		`123l`
		`0xABCDEFL`

Flottant	*Nombre contenant soit un point, soit la lettre* `e` *(majuscule ou minuscule), suivi d'un entier, soit les deux.*	`123e-10`
		`123.456`
		`-1.432e-15`

Chaîne	*Suite de caractères ne contenant ni guillemets ni saut de ligne encadrée par des guillemets, ou suite de caractères ne contenant ni apostrophe droite ni saut de ligne encadrée par des apostrophes droites, ou chaîne pouvant contenir guillemets et/ou sauts de lignes encadrés par des triples guillemets ou des triples apostrophes droites.*	`"Boujour tout'l'monde !"`
		`'Je dis "Bonjour"'`
		`"""Bonjour`
		`Avec ", ' et sauts`
		`de ligne !"""`
		`'''Idem`
		`avec`
		`apostrophes !'''`

a. En réalité, Python ne considère les nombres entiers sous forme littérale que comme des nombres positifs, et traite le signe – comme un opérateur unaire de négation. Ce détail n'a aucune importance dans notre étude du langage.

Attention

Dans Zope, si on souhaite utiliser une balise `<dtml-var>` *contenant une expression sous forme de chaîne, les guillemets « " » sont interdits. On doit donc utiliser soit la syntaxe avec simples apostrophes, soit la syntaxe avec triples apostrophes si votre chaîne contient déjà des apostrophes simples.*

Types composés : séquences et dictionnaires

Python supporte enfin des types composés. Il ne s'agit pas, comme en C ou en Pascal, de structures de type enregistrements (`struct` en C ou `record` en Pascal), mais d'agrégations de valeurs. Python supporte, dans la définition du langage, trois types d'agrégations :

- **tuples** : il s'agit d'une liste ordonnée de valeurs *ne pouvant pas être modifiée* (immuable). De même, aucun élément de la liste ne peut être inséré ou supprimé.
- **listes** : il s'agit d'une liste ordonnée de valeurs pouvant être modifiée (mutable). Des éléments de la liste peuvent être ajoutés, insérés, supprimés.

- **dictionnaires** : liste *non ordonnée* de paires clé/valeur. Chaque valeur peut être retrouvée grâce à sa clé. Des éléments peuvent être ajoutés, supprimés, modifiés.

Séquences (tuples et listes)

Les tuples sont représentés par une liste de valeurs séparées par des virgules et encadrées par des parenthèses. Voici une séquence de nombres affectés à la variable `x` :

```
>>> x = (123, 654, 321, 654, 987)
```

Pour récupérer un élément de la liste, il suffit de connaître son indice (sachant que le premier élément de la liste a pour indice `0`) :

```
>>> x[0]
123
>>> x[3]
654
>>> x[10]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: tuple index out of range
```

Le dernier exemple montre que Python renvoie une erreur si on tente d'accéder à un indice qui n'existe pas. On peut également afficher la séquence entière à l'écran :

```
print x
(123, 654, 321, 654, 987)
```

Il est possible de mélanger des données de plusieurs types au sein d'un tuple. Il est même possible d'insérer des tuples dans un tuple : toute valeur peut devenir l'élément d'un tuple.

> **Remarque**
>
> *Un tuple vide s'exprime par une paire de parenthèses* `()`*. En revanche, un tuple ne contenant qu'un seul élément s'exprime obligatoirement avec une virgule après le premier élément :* `(12, )`*. Cette syntaxe permet de ne pas confondre un tuple qui ne comprend qu'un seul élément avec un groupage d'expressions par parenthèses.*
>
> *Nous verrons dans l'étude du langage que Python accepte toujours une virgule « en trop » à la fin d'une séquence.*

Les listes fonctionnent de la même manière mais doivent être encadrées par des crochets :

```
>>> y = ["Hello", ",", "World", "!"]
```

La variable `y` contient une liste de chaînes de caractères. Tout comme pour les tuples, il est possible de récupérer un élément par son indice :

```
>>> y[0]
'Hello'
```

En revanche, contrairement au cas des tuples, il est possible de remplacer un élément par un autre.

```
>>> y[0] = 'Bye bye'
>>> y
['Bye bye', ',', 'World', '!']
```

Il est possible de supprimer un élément avec la syntaxe suivante :

```
>>> del y[1]
>>> y
['Bye bye', 'World', '!']
```

Remarque

Une chaîne de caractères peut être vue comme une séquence de lettres (immuable).

Ainsi, nous pouvons écrire :

```
>>> chaine = "Hello, World !"
>>> chaine[2]
'l'
```

Dictionnaires (mapping)

Les dictionnaires contiennent eux aussi une liste d'éléments, mais chacun d'entre eux est identifié non pas par un indice mais par une *clé*. Chaque clé du dictionnaire choisie par le programmeur doit être unique, mais peut être quasiment de n'importe quel type.

Un dictionnaire est encadré par des accolades `{` et `}` ; les paires clés / valeurs sont séparées par des virgules et la clé, quant à elle, est séparée de la valeur par le signe « : ». Voici un dictionnaire qui contient notre liste et notre tuple, identifiés par les mots `Liste` et `Tuple` :

```
>>> z = {"Tuple": x, "Liste": y}
>>> z
{'Tuple': (123, 456, 321, 456), 'Liste': ['Hello', 'World', '!']}
>>> z["Tuple"]
(123, 456, 321, 456)
```

La clé de la valeur tuple est le mot `"Tuple"`.

Il est également possible de rajouter des éléments au dictionnaire :

```
>>> z["Entier"] = 123
>>> z
{'Tuple': (123, 456, 321, 456), 'Entier': 123, 'Liste': ['Hello', 'World', '!']}
```

Attention

Cet exemple montre bien que les dictionnaires ne sont pas ordonnés : lorsque nous parcourrons les dictionnaires, il ne faudra jamais présumer à l'avance de l'ordre dans lequel il est trié.

None

Il existe enfin un dernier type important en Python : `None`. Il s'agit d'un type signifiant « rien », qui ne peut prendre qu'une valeur, `None`, toujours fausse. `None` est donc à la fois un type et une variable.

Nous verrons dans quels cas elle peut être utile.

Opérateurs

Nous pouvons désormais dresser une liste exhaustive des opérateurs Python d'après les types qui s'y rapportent. Ce tableau montre la diversité des opérateurs proposés par Python. En

Opérateur	Type(s) applicable(s)	Signification	Exemple

`+`	*Nombres*	*Addition*	`5 + 5`
			`10.5 + 8`
			`+45`

	Chaînes	*Concaténation*	`"Hello" + " World"`
	Séquences (de même nature)*	*Concaténation*	`(1, "Hello") + (3, "World")`
`-`	*Nombres*	*Soustraction*	`4 - 8` `-6`
`*`	*Nombres*	*Multiplication*	`4 * 8`
	Une séquence et un entier	*Duplication*	`"Hello" * 3` *(retourne* `"HelloHello-Hello"`*)*
`/`	*Nombres entiers*	*Division entière*	`4 / 3` *(retourne* `1`*)*
	Au moins un nombre flottant	*Division réelle*	`4 / 3.` *(retourne* `1.3333333`*)*
`**`	*Nombres*	*Élévation à la puissance*	`5 ** 2` *(retourne* `25`*)* `5 ** 1.5` *(retourne* `11.1803398875`*)*
`%`	*Nombres*	*Modulo*	`5 %% 2` *(retourne* `1`*)*
`>>`	*Entiers*	*Décalage à droite*	`5 >> 2`
`<<`	*Entiers*	*Décalage à gauche*	`5 << 2`
`&` *ou* `and`	*Entiers*	*Opération « ET » binaire*	`456 & 123` *(retourne* `72`*)*
`|` *ou* `or`	*Entiers*	*Opération « OU » binaire*	`456 | 123` `456 or 123`

Opérateur	Type(s) applicable(s)	Signification	Exemple
`^`	*Entiers*	*Opération « OU EXCLUSIF » binaire*	`456 ^ 123`
`~`	*Entier*	*Complément à 1 binaire (opérateur unaire)*	`~123`
`<`	*Quelconques***	*Inférieur à… (retourne une valeur booléenne vraie**** si la valeur de gauche est inférieure à celle de droite)*	`45 < 12` *(renvoie* `0`*)* `12 < "Coucou"` *(renvoie* `1`*)*
`>`	*Quelconques*	*Supérieur à…*	`45 > (1, 2, 3)` `[0, 1, 2] > [1, 2, 3]`
`<=`	*Quelconques*	*Inférieur ou égal à…*	`12 <= 12`
`>=`	*Quelconques*	*Supérieur ou égal à…*	`"Hello" >= "World"`
`==`	*Quelconques*	*Égalité (retourne une valeur booléenne)*	`12 == 12.0` *(renvoie* `1`*)* `12 == "12"` *(renvoie* `0`*)*
`!=` *ou* `<>`	*Quelconques*	*Différence (retourne une valeur booléenne)*	`12 != 12.1` `"15" != 15`
`is`	*Quelconques*	*Identité (c'est-à-dire même variable)*	`12 is 12`
`is not`	*Quelconques*	*Non-identité*	`12 is not 12`
`not`	*Quelconque (traité comme une valeur booléenne)*	*Négation booléenne (unaire)*	`not 1` *(retourne* `0`*)* `not "x"` *(retourne* `0`*)*
`in`	*Quelconque et séquence*	*Appartenance : renvoie vrai si l'opérande de gauche appartient à la séquence de droite*	`1 in (1, 2, 3)` *(retourne* `1`*)* `"a" in ["x", "y", "z"]` *(retourne* `0`*)* `"a" in ["a", "b", "c"]` *(retourne* `1`*)* `"H" in "Hello, World !"` *(retourne* `1`*)*
`not in`	*Quelconque et séquence*	*Non-appartenance : inverse logique de l'opérateur* `in`	`"a" not in ["x", "y", "z"]` *(retourne* `1`*)*

* *Par nature, nous entendons « tuple » ou « liste » : cette remarque précise qu'un tuple ne peut pas être concaténé à une liste, et vice versa.*

** *En Python, deux valeurs de types différents sont quasiment toujours comparables. Il existe des règles strictes précisant dans quel cas une valeur d'un type est inférieure ou supérieure à une valeur d'un autre type – il n'est d'ailleurs pas nécessaire de connaître ces règles. Il est juste rassurant de savoir que deux valeurs dont les types sont différents ne seront jamais égales (sauf entre types numériques).*

*** *En général, la valeur booléenne « vrai » retournée par Python est l'entier* `1`*, tandis que celle de « faux » est l'entier* `0`*. Attention, cependant, on peut rencontrer des cas dans lesquels les valeurs booléennes retournées par Python sont différentes de* `0` *ou* `1`*. Il ne faut donc faire aucune assertion quant à la valeur numérique des booléens retournés par Python.*

**** *L'opérateur* `!=` *est généralement préféré à* `<>`*. Ce dernier est considéré comme obsolète mais a été préservé pour des raisons de compatibilité.*

***** *L'identité permet de tester si deux variables sont situées « dans la même case mémoire » en comparant leurs pointeurs. Il s'agit d'un opérateur qui est rarement utilisé.*

outre, la possibilité qui est offerte d'utiliser certains de ces opérateurs directement sur des séquences ou des dictionnaires permet de prendre la mesure de la simplicité et de la fluidité de la syntaxe de Python.

Variables

Nous avons dit plus haut que Python est un langage dynamiquement (ou faiblement) typé. Qui plus est, comme il est de nature interprétée, la déclaration des variables est inutile. Autrement dit, une expression comme

```
x = 123
```

est toujours valide quel que soit le contexte et quelles que soient les valeurs précédentes de `x` le cas échéant.

Identifiants

Les noms de variables (identifiants) en Python sont une suite de lettres et/ou de nombres et/ou de signes « _ », commençant obligatoirement par une lettre ou par « _ ».

> **Remarque**
> *Python fait la distinction entre majuscules et minuscules dans les noms de variables.*

En outre, aucun nom de variable ne doit correspondre à un mot-clé du langage Python. Voici la liste des mots-clés du langage :

and	del	for	is	raise
assert	elif	from	lambda	return
break	else	Global	not	try
class	except	If	or	while
continue	exec	Import	pass	
def	finally	In	print	

Par exemple, les noms de variables suivants sont valides et correspondent à des variables différentes :

```
x
x123
xXyYzZ
XxYyZz
_____x_____
```

En revanche, les identifiants suivants ne sont *pas* valides :

```
12x
1X
del
assert
pass
```

Conseil

D'une manière générale, les noms commençant par le signe « _» doivent être réservés aux symboles « utilitaires » définis par le programmeur qui ne doivent pas apparaître aux yeux des autres programmeurs. Il s'agit de la même convention que celle qui est utilisée en C.

Les symboles commençant et finissant par « __» sont généralement réservés pour les symboles intrinsèques de Python. Nous en verrons quelques exemples dans notre étude des classes.

Remarque

Sous Zope, en DTML ou dans un script Python, l'utilisation de variables commençant par « _ » est interdite (sauf pour la variable « _» représentant l'espace de noms).

Portée d'une variable

Tout comme Zope définit l'espace de noms, Python définit la portée d'une variable, qui indique l'étendue du programme dans lequel une variable est accessible.

Python ne connaît, dans sa version 1.5.2, que deux domaines de définition :

- le domaine global ;
- le domaine local.

Le domaine local correspond au domaine compris dans le corps d'une fonction. Le domaine global s'applique aux symboles définis en dehors de toute fonction, et qui sont donc accessibles à partir de quelque endroit que ce soit dans le programme.

Chaque fonction pouvant utiliser ses propres variables, il n'existe pas *un* mais *plusieurs* domaines locaux. Lorsque l'interpréteur entre dans le corps d'une fonction, il ne prend en considération que le domaine local à la fonction qu'il interprète. En revanche, le domaine global est le même dans tout le programme.

Nous aurons l'occasion de revenir sur ces définitions lors de l'étude des fonctions.

Référence ou valeur ?

Le langage Python ne permet pas au programmeur de manipuler des pointeurs comme en C ou en C++. Si les pointeurs sont masqués aux yeux des programmeurs, *les notions sous-jacentes n'en existent pas moins, notamment en ce qui concerne le passage d'une valeur à une fonction, ou le retour d'une valeur par une fonction.*

Remarque

Les variables représentant des types élémentaires (entier, flottant…) sont manipulées comme des valeurs. Les variables représentant des types composés (séquences, chaînes, dictionnaires et autres objets Python) sont pour leur part manipulées comme des pointeurs.

Voici comment le démontrer :

```
>>> c = [1, 2, 3, 4, 5]
>>> d = c
>>> c
[1, 2, 3, 4, 5]
>>> d
[1, 2, 3, 4, 5]
>>> c[0] = 999
>>> c
[999, 2, 3, 4, 5]
>>> d
[999, 2, 3, 4, 5]
```

En modifiant la liste contenue dans la variable `c`, nous avons également modifié celle contenue dans `d`.

Ce comportement nous permet d'ailleurs d'écrire l'instruction suivante :

```
>>> c[0] = c
```

Le premier élément de la liste `c` se contient lui-même… Si nous tentons d'afficher `c`, Python signale la récursivité de la manière suivante :

```
>>> print c
[[...], 2, 3, 4, 5]
```

En revanche, si nous modifions le contenu de la variable `c` pris dans sa globalité, `d` n'est pas affecté :

```
>>> c
[[...], 2, 3, 4, 5]
>>> c = "une chaîne"
>>> d
[[...], 2, 3, 4, 5]
>>> c
'une cha\214ne'
```

La valeur « pointée » par `c` est modifiée, tandis que celle de `d` est restée la même. Ce comportement est valide tant que la valeur est prise comme un tout. Nous pouvons également observer le fragment de code suivant :

```
>>> d = c
>>> c = c + " de caractères."
>>> c
'une cha\214ne de caract\212res.'
>>> d
'une cha\214ne'
```

Ici, l'opération `c = c + " de caractères."` a fait qu'une nouvelle valeur a été affectée à `c`. La valeur de `d` n'a donc pas été modifiée.

L'opérateur `is` prend ici tout son sens : deux expressions peuvent être identiques mais l'opérateur `is` vérifie l'identité des pointeurs entre les deux. Il est parfois préférable d'utiliser `is` plutôt que `==` dans certaines conditions (nous n'en ferons pas mention ici).

Ce comportement est finalement assez naturel et n'appelle pas de commentaires particuliers : le principe de la « moindre surprise » s'y applique facilement. Il est cependant important de le mettre en évidence pour ne pas être surpris par certains comportements du langage.

Vers un premier programme

Structure d'un programme Python

Comme beaucoup de langages de programmation, Python permet d'écrire des programmes structurés sous forme de fonctions.

Une fonction est définie dans le corps du programme et forme ce que l'on appelle un bloc. Un bloc peut lui-même être composé de fonctions (bien que ce soit plus rare).

L'ensemble des fonctions se trouve dans un fichier, appelé « script », s'il constitue un tout destiné à être exécuté directement par l'interpréteur, ou « module », s'il constitue un ensemble de fonctions utilitaires destinées à être utilisées par d'autres modules ou scripts. Les fichiers scripts et modules portent l'extension `.py`.

Un script Python est composé de fonctions. Mais, pour qu'il soit utilisable, il faut qu'au chargement de ce script par l'utilisateur, l'une de ces fonctions soit exécutée par l'interpréteur. En C, par exemple, il s'agit par convention de la fonction appelée `main`. En Python, il n'en est rien : Python exécute au chargement du script ou du module tous les traitements qui se trouvent *en dehors* du corps des fonctions. Le corollaire est qu'un script ou un module Python peut très bien ne contenir aucune fonction.

Voici à titre d'exemple un programme Python valide :

```
print "Hello, World !"
```

En plaçant cette ligne dans un fichier `hello.py` puis en appelant l'interpréteur depuis le shell comme suit :

```
python hello.py
```

la chaîne `Hello, World !` s'inscrit à l'écran.

Quand Python fonctionne en mode interactif, les commandes que vous tapez sont traitées comme s'il s'agissait de commandes contenues dans un fichier : Python les interprète sur-le-champ car elles se trouvent en dehors de toute fonction. Mais il est tout à fait possible de définir des fonctions en mode interactif.

Commentaires

Un langage de programmation se doit d'accepter les commentaires. En Python, le caractère # est utilisé pour annoncer un commentaire : tout ce qui suit ce caractère jusqu'au saut de ligne est considéré comme du commentaire.

```
>>> # Voici un commentaire !
... 1 + 1 # Un autre commentaire
2
```

Remarque

Il n'est pas possible, en Python, de commenter tout un bloc de code. Les utilisateurs d'Emacs en mode Python peuvent sélectionner la région, et utiliser respectivement `Ctrl+C+#` *pour commenter un bloc de code, et* `Ctrl+U+C+#` *pour ôter les commentaires.*

Fonction

Définition et appel

Pour définir une fonction, on utilise le mot-clé `def` suivi du nom de la fonction, de ses paramètres formels entre parenthèses, puis de deux points (`:`).

Ensuite, on écrit le corps de la fonction, sous une forme *indentée*.

Pour appeler une fonction, on utilise son identifiant suivi des paramètres éventuels entre parenthèses et séparés par une virgule.

Remarque

Dans la quasi totalité des constructions syntaxiques de Python composées d'éléments séparés par des virgules, il est possible d'ajouter une virgule à la fin de l'expression. Par exemple, dans le code ci-après, nous pourrions très bien remplacer la ligne `def Afficher(texte)` *par* `def Afficher(texte, )`.

Cette règle s'applique également pour la définition de tuples, listes et dictionnaires : l'expression `(1, 2, 3, )` *est valide et strictement équivalente à* `(1, 2, 3)`.

Ouvrez votre éditeur favori et créez le fichier `hello.py` contenant le code suivant :

```
# Hello.py
# Programme d'exemple d'une fonction Python

def Hello():
    print "Hello, World !"
```

```
def Afficher(texte):
    "Affichage d'une chaîne de caractères"
    print texte

Hello()
Afficher("Bonjour !")
```

Si vous interprétez ce fichier avec Python, voici le résultat :

```
$ python hello.py
Hello, World !
Bonjour !
```

Remarque

Nous avons dit plus haut que `print` *est une instruction et non une fonction : c'est la raison pour laquelle l'instruction* `print` *n'est pas suivie d'une parenthèse ouvrante.*

Domaine de définition d'une fonction

L'interpréteur Python lit le fichier source de manière séquentielle et rend disponibles les fonctions au fur et à mesure de son parcours. Il n'effectue donc qu'une seule passe d'interprétation (et ne revient donc pas en arrière).

Attention

Une fonction ne peut être utilisée qu'à partir de l'endroit où elle a été définie, pas avant.

Cette remarque concerne les lignes suivantes :

```
Hello()
Afficher("Bonjour !")
```

Ces lignes font référence à `Hello` et `Afficher` et doivent donc obligatoirement se situer *après* leur définition dans le fichier `hello.py`. C'est la raison pour laquelle la partie « exécutable » d'un script se situe presque toujours à la fin du script.

Remarque

Python est un langage très homogène. Le langage se décrit très bien par lui-même. Ainsi, les paramètres d'une fonction peuvent être vus comme un tuple de noms de variables (nous verrons ce point plus en détail dans les fonctions à nombre variable d'arguments). De même, une fonction est identifiée par un symbole qui est lui-même une variable. Dans notre programme `hello.py`*, une instruction telle que* `Test = Afficher` *a un sens et a pour effet d'indiquer que Test devient un synonyme de* `Afficher`*. Aussi pourrions-nous écrire* `Test("Bonjour")` *pour afficher un message à l'écran.*

Documentation

Python incite à la documentation. Dans le code source de `hello.py`, la fonction `Afficher` commence par une chaîne de caractères : il s'agit d'une chaîne de documentation. Cette chaîne peut être utilisée par la suite par le programmeur à titre de référence.

Ouvrez l'interpréteur Python et saisissez la fonction `Afficher` comme suit :

```
>>> def Afficher(texte):
...     "Affichage d'une chaîne de caractères"
...     print texte
...
```

Remarque

Lorsque l'interpréteur attend un bloc, l'invite `>>>` est remplacée par une invite `....` De même, il est important de saisir le saut de ligne à la dernière ligne (vide) pour retrouver l'invite `>>>`.

Puis, exécutez l'instruction suivante :

```
>>> print Afficher.__doc__
Affichage d'une chaîne de caractères
```

Cette technique, qui s'appuie sur la variable spéciale `__doc__`, peut être mise à profit pour n'importe quelle fonction pourvue de documentation – y compris les fonctions intégrées à Python. Par exemple, Python propose une fonction `list`, donc on peut lire la documentation comme suit :

```
>>> print list.__doc__
list(sequence) -> list

Return a new list whose items are the same as those of the argument sequence.
```

Remarques

Lorsque nous étudierons la programmation de produits Zope, nous verrons que Zope utilise cette technique pour rendre une méthode « publique » ou pas : si elle est documentée, alors elle est publique, sinon elle est privée. Voir le chapitre 12, consacré à la création de produits Python, pour plus d'informations.

Nous verrons dans la suite de ce chapitre que cette syntaxe (`list.__doc__`) correspond à l'appel d'une variable d'objet. Une fonction Python est donc une variable d'un type particulier, ce que l'on peut d'ailleurs vérifier au moyen de l'instruction `print Afficher`, qui retourne `<function Afficher at 795570>`. Python indique qu'il s'agit bien d'une fonction.

Retour de valeur

Une valeur de retour peut être spécifiée dans le corps d'une fonction avec l'instruction `return` suivie d'une expression.

Si une fonction ne comporte pas d'instruction `return`, elle retourne implicitement la valeur `None`. Voici comment le vérifier :

```
>>> print Afficher("")

None
```

Un exemple classique de programmation consiste à écrire la fonction factorielle sous forme récursive. Voici comme le faire en Python :

```
>>> def Factorielle(n):
...     if n == 0:
...         return 1
...     return n * Factorielle (n - 1)
...
>>> Factorielle (4)
24
```

Un bloc vide

Il arrive parfois que l'on ait besoin d'écrire une fonction qui ne fasse rien (c'est le cas par exemple dans certaines méthodes de conception qui stipulent que l'on doit écrire le squelette du programme avant d'écrire le contenu des fonctions). En Python, la plupart des structures en bloc (fonctions et, comme nous le verrons ci-dessous, branchements conditionnels, boucles ou encore classes) nécessitent un bloc. Ainsi, la définition suivante n'est pas valide :

```
def rien():
```

En effet, un bloc est attendu immédiatement.

Pour pallier cet inconvénient, on utilise l'instruction `pass`, qui ne fait rien. Ainsi le code suivant est-il valide :

```
def rien():
    pass
```

Branchements et boucles

Les instructions de branchements et de boucles permettent de dérouter le flot de contrôle d'un programme suivant certaines conditions. L'exemple le plus connu est le test (`if`). Si le test est vrai, alors un bloc est exécuté, sinon un autre bloc est éventuellement exécuté.

Attention

Il n'existe pas en Python d'instruction équivalente à `goto`.

Branchements conditionnels

L'instruction `if` permet de tester si une expression est vraie et d'agir en conséquence. La syntaxe en est simple (la syntaxe est présentée ici dans un format descriptif – les crochets indiquent des éléments optionnels, le + après un crochet signifie que l'élément peut se répéter plusieurs fois) :

```
if <expression>:
    <bloc>
[elif <expression>:
    <bloc>]+
[else:
    <bloc>]
```

Attention

En Python, les blocs doivent toujours être indentés correctement !

Voici un exemple dans une fonction (saisie dans l'interpréteur) :

```
>>> def Vrai(variable):
...     if variable:
...         print "Vrai"
...     else:
...         print "faux"
...
>>> Vrai(1)
Vrai
>>> Vrai("abc")
Vrai
>>> Vrai([])
faux
>>> Vrai(0)
faux
>>> Vrai(None)
faux
```

La bibliothèque de Python propose une fonction `type` qui permet de connaître le type d'une expression. Nous pourrions modifier la fonction `Afficher` comme suit :

```
>>> def Afficher(valeur):
...    if type(valeur) == type(0):
...        print "Entier:", valeur
```

```
...    elif type(valeur) == type(""):
...        print "Chaîne:", valeur
...    else:
...        print "Autre:", valeur
...
>>> Afficher("Hello")
Chaîne: Hello
>>> Afficher(123)
Entier: 123
>>> Afficher(type)
Autre: <built-in function type>
```

L'instruction `pass` peut être utilisée pour un bloc dans une instruction `if`.

Boucle `while`

Comme en C, en Java ou en Pascal, le langage met à disposition une boucle `while`, qui stipule que le bloc qui la suit est exécuté tant que l'expression suivant `while` est vraie. Voici la syntaxe de la boucle `while` :

```
while <expression>:
    <bloc>
```

Cette boucle est utile pour gérer des itérations. Voici comment effectuer une fonction factorielle :

```
>>> def Factorielle(n):
...     ret = 1
...     while n > 0:
...         ret = ret * n
...         n = n - 1
...     return ret
...
>>> Factorielle (4)
24
```

Comme en C, l'instruction `continue` au sein d'une boucle permet de retourner immédiatement à la ligne `while`. L'instruction `break` interrompt immédiatement le déroulement de la boucle.

La boucle `for`

La boucle `for` fonctionne différemment de ce qui se rencontre dans d'autres langages. Alors que, dans la plupart des cas (sauf en C ou en Java), la boucle `for` parcourt des nombres compris entre deux intervalles, en Python, la boucle `for` parcourt des séquences. En fait, à part dans des cas qui relèvent strictement des mathématiques comme la fonction `Exponentielle`, les boucles d'un programme servent surtout à parcourir des séquences d'éléments[1].

1. C'est d'ailleurs la raison d'être des objets itérateurs en programmation orientée objet.

La syntaxe générale de la boucle `for` en Python est la suivante :

```
for <identifiant> in <séquence>:
    bloc
```

Considérons par exemple une fonction `AfficherChaines` qui afficherait une séquence de chaînes ligne par ligne :

```
>>> def AfficherChaines(chaines):
...     for chaine in chaines:
...         print chaine
...
>>> AfficherChaines(['Hello', 'World', '!'])
Hello
World
!
```

Si nous avions dû nous contenter de la boucle `while`, la même fonction aurait été :

```
>>> def AfficherChaines(chaines):
...     index = 0
...     while index < len(chaines):
...         print chaines[index]
...         index = index + 1
...
>>> AfficherChaines(['Hello', 'World', '!'])
Hello
World
!
```

Cette version est nettement moins efficace que la première, et nettement plus difficile à lire et à comprendre. L'utilisation massive de la boucle `for` permet de résoudre la plupart des problèmes de boucles en Python.

Qui peut le plus peut le moins : Python propose un mécanisme pour reproduire le comportement des boucles `for` du Basic ou du Pascal, avec la fonction intégrée `range`.

```
>>> print range.__doc__
range([start,] stop[, step]) -> list of integers

Return a list containing an arithmetic progression of integers.
range(i, j) returns [i, i+1, i+2, ..., j-1]; start (!) defaults to 0.
When step is given, it specifies the increment (or decrement).
For example, range(4) returns [0, 1, 2, 3].  The end point is omitted!
These are exactly the valid indices for a list of 4 elements.
```

Les arguments entre crochets (`start` et `step`) sont optionnels. Par exemple, `range(10)` retourne la séquence `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`. Ainsi, pour écrire une boucle (ce qui en soi présente peu d'intérêt) qui compte de 2 en 2 de 0 à 50, on pourrait écrire :

```
>>> for n in range(0, 20, 2):
...     print n
...
0
2
4
6
8
10
12
14
16
18
```

Bien sûr, au sein d'une boucle `for`, les instructions `continue` et `break` peuvent être utilisées.

Remarque

La boucle `for` *de Python fait immanquablement penser à la balise* `<dtml-in>` *du DTML (à moins que ce ne soit l'inverse...). Il faut cependant relever quelques différences notables entre les deux. Le DTML définit à l'intérieur des boucles des variables particulières (*`sequence-start`*,* `sequence-end`*...) qui n'existent pas en Python. Par exemple, contrairement au DTML avec* `sequence-start` *et* `sequence-end`*, il n'est pas possible en Python de déterminer la dernière itération au sein d'une boucle. De même, les variables statistiques ne sont pas disponibles. Dans le même ordre d'idées,* `sequence-item`*,* `sequence-index` *et* `sequence-number` *n'existent pas en Python.*

En revanche, la boucle `<dtml-in>` *ne permet pas de définir le nom de la variable de conduite de boucle : il s'agit toujours de* `sequence-item`*. Ce point peut s'avérer gênant dans le cas de boucles imbriquées (voir le chapitre consacré au DTML pour plus d'informations).*

Modules (… et scripts)

Dans cette section, nous allons traiter de la manipulation des modules (c'est-à-dire de fichiers définissant des fonctions ou des classes utilitaires) en Python, et présenter la bibliothèque des modules fournie en standard avec Python.

Importer un module

Considérons un programme Python simple, qui va consister à demander à l'utilisateur une phrase et à l'afficher à l'écran, en séparant chaque mot par un saut de ligne. Le langage Python propose l'instruction `print` pour afficher un texte à l'écran suivi d'un saut de ligne, mais ne propose aucune instruction de type `input` pour accepter une saisie au clavier ou pour afficher,

ni aucune autre instruction pour séparer une phrase en mots. En laissant de côté ces éléments problématiques, nous pouvons tout de même écrire la fonction principale de notre programme, `AfficherMots`, comme suit :

```
>>> def AfficherMots():
...     chaine = DemanderChaine()
...     for mot in SeparerMots(chaine):
...         print mot
...
```

Pour simplifier, nous allons dans un premier temps écrire une fonction `DemanderChaine` qui renvoie toujours la chaîne « `Hello, World !` » :

```
>>> def DemanderChaine():
...    return "Hello, World !"
```

Nous modifierons ultérieurement cette fonction.

Importer un symbole

Nous allons nous concentrer maintenant sur l'écriture de la fonction `SeparerMots` qui prend une chaîne en entrée et retourne une séquence de mots en sortie. On dispose d'un module `string` qui contient un ensemble de fonctions très pratiques pour manipuler les chaînes. Ainsi, plutôt que de parcourir la chaîne caractère par caractère et de rechercher les espaces, la fonction `split` découpe précisément une chaîne de caractères en liste de mots. Lorsqu'on lance l'interpréteur, Python ne connaît pas la fonction `split` :

```
>>> split
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: split
```

Pour utiliser cette fonction, il faut l'*importer* depuis son module d'origine. Pour cela, on utilise la syntaxe suivante :

```
>>> from string import split
```

Littéralement, cette ligne se lit « Importe le symbole `split` depuis le module `string` ». La syntaxe générale en est donc la suivante :

```
from <module> import <symbole>[, <symbole]+
```

Cette syntaxe montre bien qu'il est possible d'importer plusieurs symboles (fonctions ou variables) depuis un même module. Un programme Python peut comporter autant de directives `import` que nécessaire. Le symbole `split` est défini dans le programme à partir du point où l'importation a été effectuée. Nous pouvons maintenant vérifier que `split` existe bien :

```
>>> split
<built-in function split>
```

```
>>> print split.__doc__
split(str [,sep [,maxsplit]]) -> list of strings
splitfields(str [,sep [,maxsplit]]) -> list of strings

Return a list of the words in the string s, using sep as the
delimiter string.  If maxsplit is nonzero, splits into at most
maxsplit words If sep is not specified, any whitespace string
is a separator.  Maxsplit defaults to 0.

(split and splitfields are synonymous)
```

Nous pouvons alors écrire la fonction `SeparerMots` comme suit :

```
      >>> def SeparerMots(chaine):
...     return split(chaine)
```

Et pour tester le programme :

```
>>> AfficherMots()
Hello,
World
!
```

Importer tout un module

Le module `string` est très pratique pour la manipulation des chaînes. Il arrive que l'on doive recourir à la plupart des fonctions et symboles définis par un module. Dans ce cas, on utilise une syntaxe particulière :

```
from string import *
```

Cette commande permet d'importer tous les symboles du module `string`. Ainsi, nous avons désormais accès aux fonctions `join` et `strip` et aux variables `lowercase` et `uppercase`. On se reportera à la documentation de la bibliothèque standard de Python pour disposer d'une liste exhaustive des modules et de leur contenu.

Attention

Les symboles du module commençant par _ ne sont pas importés par l'instruction `from <module> import *`*. Il est en revanche possible de les importer en utilisant* `from <module> import <symbole>`*.*

Conseil

La syntaxe `from module import *` *est en fait à proscrire : elle transforme les gros programmes en un énorme capharnaüm dans lequel on ne sait plus quel symbole est défini dans quel module. En effet, il est tout à fait possible que plusieurs modules définissent des symboles du même nom : le dernier importé prime alors sur les autres. En conséquence, si l'ordre des instructions* `import` *est modifié, le comportement du programme peut être altéré de manière dramatique…*

Importer le module lui-même

Notre programme est presque complet : il reste à réécrire la fonction `DemanderChaine` qui acceptera une saisie au clavier. Là encore, il nous faut recourir à un module pour que l'on obtienne une fonction permettant les saisies au clavier : le module `sys` définit l'objet `stdin` qui représente l'entrée standard du programme.

Plutôt que d'importer directement le symbole `stdin`, nous allons simplement importer le module `sys` lui-même avec l'instruction suivante :

```
import sys
```

Nous pouvons alors utiliser tous les symboles du module `sys` avec la syntaxe à point : `sys.<symbole>`. L'objet `stdin` définit la méthode[1] `readline()` qui permet d'accepter une ligne au clavier ; voici donc la fonction `DemanderChaine` récrite :

```
>>> def DemanderChaine():
...     return sys.stdin.readline()
...
```

La fonction `AfficherMots` rend désormais le service souhaité.

Remarque

Python étant un langage interprété, il est tout à fait possible de définir une fonction plusieurs fois – c'est ce que nous avons fait avec `DemanderChaine`*. La fonction déclarée en dernier dans le flot d'exécution du programme est alors celle qui est utilisée. Pour bien garder à l'esprit ce comportement, il suffit de se rappeler qu'une fonction est une variable…*

Conseil

Cette méthode d'importation, `import <module>`*, doit, et de loin, être préférée aux deux autres présentées précédemment. En effet, bien qu'en y recourant le programmeur doive mentionner le module à chaque utilisation d'une fonction, elle permet de ne jamais confondre deux fonctions qui appartiendraient à des modules différents, comme nous l'avons souligné plus haut. En outre, un module étant lui-même une variable (il s'agit d'un objet particulier), Python propose des outils pour le manipuler. Par exemple, la fonction* `dir(sys)` *permet de lister tous les symboles définis par le module* `sys`*.*

En outre, un module étant lui-même une variable (il s'agit d'un objet particulier), Python propose des outils pour le manipuler. Par exemple, la fonction `dir(sys)` *permet de lister tous les symboles définis par le module* `sys`*.*

1. Nous reviendrons sur la signification exacte du mot « méthode ». Pour l'instant, il suffit de savoir qu'il s'agit d'une fonction appliquée à un objet (ici la fonction `readline` est appliquée à `sys.stdin`).

Écrire un module

Code d'amorçage d'un module

Écrire un module est en soi très simple : nous avons vu plus haut qu'un module et un script sont très proches – sauf qu'un script possède une sorte de code d'amorçage qui lui permet d'être exécuté sans faire appel à une fonction particulière.

En fait, il est possible d'importer un script : le script sera alors exécuté par la directive `import`. Inversement, il est possible de placer dans le module du code en dehors de toute fonction, qui sera alors exécuté lors de la première importation du module. Certains modules utilisent cette stratégie pour exécuter du code d'initialisation, par exemple.

Bien sûr, cela n'est pas obligatoire, et un module peut très bien ne pas comporter de « code d'amorçage ».

Script ou module ?

En fait, il s'avère pratique qu'un module se comporte comme un script dans un cas précis : quand on met au point le module, on souhaite souvent disposer d'un jeu de tests permettant de tester globalement toutes les fonctions du fichier lorsque ce dernier est appelé comme un script ; lequel jeu, toutefois, ne sera pas exécuté quand le module est importé. Pour y parvenir, on peut placer, à la fin du fichier module, les lignes suivantes :

```
if __name__ == '__main__':
    <bloc de test>
```

Où `<bloc de test>` sont les instructions permettant de tester individuellement les fonctions du module, ou, d'une manière générale, les instructions qui doivent être exécutées si le fichier est interprété comme un script et non comme un module. Le test se justifie de la manière suivante :

- `__name__` est une variable particulière de Zope qui renvoie le nom du module en cours d'interprétation. Si, dans un module quelconque, on insère l'instruction `print __name__`, Python retournera le nom du module (soit le nom du fichier sans l'extension `.py`).
- Si, dans l'interpréteur ou dans un script, on entre `print __name__`, on obtiendra invariablement la chaîne `"__main__"`. Il s'agit du nom que donne automatiquement Python au fichier en cours d'interprétation s'il ne correspond à aucun module – s'il s'agit donc d'un script. Il suffit de tester que `__name__` est bien égal à `'__main__'` pour affirmer que l'on se trouve dans un script et pas dans un module.

La bibliothèque de Python

Python est fourni avec un nombre impressionnant de modules qui couvrent la plupart des besoins de base en programmation. Certains modules permettent la manipulation des chaînes de caractères ou des expressions rationnelles (*regular expressions*, en anglais). Certains autres

encapsulent les fonctions proposées par la librairie standard du langage C. D'autres encore permettent la gestion des *threads*[1] en Python.

Cette section présente certaines variables et fonctions importantes pour le programmeur.

> **Attention**
> *Bon nombre de ces modules ne sont pas accessibles directement sous Zope, pour des raisons de sécurité. Nous avons vu dans le chapitre 4 consacré au DTML qu'il était possible d'utiliser uniquement les modules* `string`, `math`, `random` *et* `whrandom`.

Le module `__builtin__`

Nous avons vu plus haut quelques fonctions qui peuvent être utilisées même si aucun module n'est importé. Par exemple, la fonction `dir` peut toujours l'être en Python et il n'y a pas lieu de l'importer depuis un module particulier.

Toutes ces fonctions sont regroupées dans un module particulier appelé `__builtins__` (il faut retenir que les variables de la forme `__x__` sont généralement des variables système de Python). Cette particularité n'a pas d'incidence sur la façon dont on peut se servir des fonctions en question, mais cela permet d'utiliser par exemple `dir(__builtins__)` pour obtenir la liste des symboles intégrés à Python.

> **Remarque**
> *En DTML sous Zope, certaines de ces fonctions sont accessibles au travers de la variable spéciale _ (par exemple, il est possible d'utiliser* `<dtml-var "_.chr(34)">` *pour afficher le caractère «* `"` *»).*

Voici la liste (non exhaustive) des symboles définis par le module :

Nom (et signature pour une fonction)	Description	Exemple
abs(nombre) -> nombre	*Renvoie la valeur absolue d'un nombre.*	`abs(-12) == 12`
`apply`*(fonction*, *arguments*, *[nommés]*)	*Appelle la fonction* `fonction` *en lui passant le tuple* `arguments` *comme arguments non nommés et le dictionnaire* `nommés` *en tant qu'arguments nommés.*	`apply(abs, (-12,))` *(équivaut au résultat de l'appel d'*`abs` *ci-dessus)*
`chr`*(entier) -> caractère*	*Renvoie le caractère à la position* `entier` *du code ASCII.*	`chr(34) == '"'`

1. La gestion des threads est une des façons d'écrire des programmes multitâches.

Nom (et signature pour une fonction)	Description	Exemple
`cmp`*(valeur1, valeur2)* `-> 0, 1` *ou* `-1`	*Compare* `valeur1` *et* `valeur2`*, retourne 0 si elles sont égales, 1 si la première est supérieure à la seconde, -1 sinon.*	`cmp(123, 465) == -1`
`dir([`*symbole*`]) ->` *liste de chaînes*	*Retourne une liste des symboles définis par l'objet* `symbole`*.*	`dir(__builtins__)`
`eval`*(chaîne)*	*Interprète* `chaîne` *comme s'il s'agissait d'une expression Python et renvoie le résultat.*	`eval(abs(-12))`
`float`*(nombre)* `->` *flottant*	*Renvoie* `nombre` *sous forme d'un flottant.*	`float(12) == 12.0`
`globals() ->` *dictionnaire*	*Retourne un dictionnaire dont les clés sont les noms des symboles définis dans l'espace de noms global de Python et dont les valeurs sont les valeurs réelles de ces symboles.*	`globals()`
`id`*(symbole)* `->` *entier*	*Retourne l'identifiant unique (à un instant donné) du symbole.*	`id(x)`
`input`*([invite])*	*Demande la saisie au clavier d'une expression Python, l'évalue et retourne le résultat.*	`input('Entrez une expression >')`
`int`*(nombre)* `->` *entier*	*Convertit* `nombre` *en un entier.*	`int(4.4) == 4`
`len`*(symbole)* `->` *entier*	*Retourne la longueur d'une variable, ou le nombre d'éléments d'une séquence ou d'un dictionnaire.*	`len([1, 2, 3]) == 3`
`list`*(séquence)* `->` *liste*	*Convertit une séquence en liste.*	`list((1, 2, 3)) == [1, 2, 3]`
`locals()` `->` *dictionnaire*	*Retourne un dictionnaire dont les clés sont les noms des symboles définis dans l'espace de noms local de Python et dont les valeurs sont les valeurs réelles de ces symboles.*	`locals()`
`long(`*nombre*`)` `->` *entier long*	*Retourne le nombre converti en un entier long.*	`long(12) == 12L`
`max(`*valeur1, valeur2*`)`	*Retourne la valeur maximale des deux valeurs.*	`max(12, 24) == 24`
`min`*(valeur1, valeur2)*	*Retourne la valeur minimale des deux valeurs.*	`min(12, 24) == 12`
`open(`*fichier[, mode]*`)`	*Retourne le fichier dont le nom est* `fichier` *conformément à la chaîne* `mode` *(*`r` *pour lecture seule,* `w` *pour écriture avec effacement,* `a` *pour écriture à la fin du fichier existant, etc.).*	`f = open('/tmp/test', 'w')`

Nom (et signature pour une fonction)	Description	Exemple
`range(`*[debut,]* *fin[, pas]*`) -> liste`	*Retourne une liste de nombres bornée par les valeurs.*	`range(5) == [0, 1, 2, 3, 4]`
`repr(`*valeur*`) ->` *chaîne*	*Retourne la valeur sous forme de chaîne.*	`repr(12.3000) == '12.3'`
`tuple(`*séquence*`) ->` *tuple*	*Convertit une séquence en tuple.*	`tuple([1, 2, 3]) == (1, 2, 3)`
`type(`*symbole*`) ->` *type*	*Retourne le type d'un symbole.*	`type('Hello, World !') == type('')`

Les modules `string`, `math`, `random` et `whrandom`

Ces modules sont définis dans le chapitre 4 consacré au DTML : il s'agit des mêmes modules que ceux qui sont accessibles respectivement *via* `_.string`, `_.math`, `_.random` et `_.whrandom`.

Programmation avancée en Python

Il est temps désormais de nous engager plus profondément dans les concepts forts de Python. Cette section propose tout d'abord un complément d'information sur les structures composées de Python, sur la gestion des chaînes de caractères et des fonctions. Puis, nous étudierons la gestion des erreurs par Zope et, enfin, l'orientation objet.

Retour sur les séquences et dictionnaires

Manipuler les séquences

Convertir une liste en tuple

Nous avons abordé quelques principes élémentaires des listes et des tuples. Nous avons en particulier vu qu'une liste est mutable mais qu'un tuple ne l'est pas. Il est possible de transformer l'un en l'autre, comme nous l'avons vu lors de l'étude du module `__builtins__`, avec les fonctions `list` et `tuple` :

```
>>> maliste = [1, 2, 3]
>>> tuple(maliste)
(1, 2, 3)
>>> list(tuple(maliste))
[1, 2, 3]
>>> montuple = (1, 2, 3,)
>>> list(montuple)
[1, 2, 3]
```

Manipulations sur les listes

Nous avons également étudié la fonction `dir` qui permet de voir les symboles définis par un module. Un module est lui-même un symbole : Python étant un langage très logique, cette

affirmation permet de supposer que l'on peut utiliser `dir` sur… un symbole, au sens large du terme. La preuve :

```
>>> x = 1
>>> dir(x)
[]
```

Python ne renvoie pas d'erreur ! La variable `x` est bien un symbole et ne contient elle-même aucun autre symbole. De même :

```
>>> dir(montuple)
[]
```

Mais, en revanche, oh surprise ! :

```
>>> dir(maliste)
['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

Le symbole `maliste` « contient » 9 symboles ! En effet, lors de notre étude des listes, nous avons affirmé qu'une liste était mutable, nous avons aussi vu comment remplacer un élément par un autre, mais nous n'avons pas vu, par exemple, comment ajouter un élément à la fin de la liste. On utilise pour cela la méthode `append`. Jusqu'ici, nous avons dit qu'une *méthode* était une fonction qui s'applique dans le contexte de la variable sur laquelle elle est appelée. La variable en question s'appelle usuellement un objet. Ainsi :

```
>>> maliste.append(0.4)
>>> maliste
[1, 2, 3, 0.4]
```

Les méthodes de liste ont, elles aussi, une documentation :

```
>>> print maliste.count.__doc__
L.count(value) -> integer -- return number of occurrences of value
>>> print maliste.insert.__doc__
L.insert(index, object) -- insert object before index
```

La méthode `count` permet de compter le nombre de fois qu'une valeur apparaît dans la liste. Avec la méthode `insert`, on peut insérer une valeur à une position précise de la liste. Par exemple :

```
>>> maliste.count(1)
1
>>> maliste.insert.__doc_
>>> maliste.insert(3, 1)
>>> maliste
[1, 2, 3, 1, 0.4]
>>> maliste.count(1)
2
```

La méthode `extend` sert à étendre une liste avec une autre liste. Par exemple :

```
>>> maliste.extend(list(montuple))
>>> maliste
[1, 2, 3, 1, 0.4, 1, 2, 3]
```

Nous aurions pu écrire un code similaire avec l'opérateur + :

```
>>> maliste = maliste + list(montuple)
```

L'utilisation de la méthode `extend` est cependant plus efficace que celle de l'opérateur +. Pour les puristes en algorithmique, la seconde solution oblige à créer une variable temporaire contenant la concaténation des deux listes, puis à l'affecter à la variable `maliste`. La première solution utilise moins de mémoire et elle est plus rapide.

Remarque

La plupart des modules standard de Python ainsi que des méthodes de gestion des types de données sont codées en C, et sont donc très efficaces. C'est aussi la raison pour laquelle il est déconseillé de vouloir « réinventer la poudre » avec Python : si un traitement élémentaire existe dans la librairie standard de Python, il y a de fortes chances qu'il soit nettement plus rapide que la version que vous pourriez écrire – à plus forte raison s'il est écrit en C.

La méthode `index` retourne le plus petit indice possible de l'élément passé entre parenthèses ; elle renvoie une erreur si l'élément n'est pas dans la liste.

```
>>> maliste
[1, 2, 3, 1, 0.4, 1, 2, 3]
>>> maliste.index(0.4)
4
>>> maliste[4]
0.4
>>> maliste.index('x')
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ValueError: list.index(x): x not in list
```

La méthode `pop` fonctionne comme en assembleur : elle ôte et renvoie le dernier élément de la liste. Il est également possible de préciser l'indice de l'élément à retirer, comme ceci :

```
>>> x = maliste.pop(3)
>>> print x
1
```

Pour supprimer un élément de la liste en connaissant son indice, on utilise l'instruction `del`, comme ceci :

```
>>> print maliste
[1, 2, 3, 0.4, 1, 2, 3]
>>> print maliste[5]
2
>>> del maliste[5]
>>> print maliste
[1, 2, 3, 0.4, 1, 3]
```

Attention

Le mot-clé `del` *est une instruction et non une méthode : on peut utiliser* `del` *pour détruire n'importe quelle variable Python.*

La méthode `remove` permet d'ôter un élément de la liste, non pas par son numéro d'indice, mais par sa valeur. On peut donc affirmer que `maliste.remove(valeur)` équivaut à `del maliste.index(valeur)`.

Enfin, les instructions `sort` et `reverse` permettent de trier une liste « sur place » (c'est-à-dire qu'elles ne retournent pas de valeur). `sort` effectue un tri, tandis que `reverse` inverse la liste (dernier élément en premier, et ainsi de suite). En voici un exemple concret :

```
>>> maliste
[1, 2, 3, 0.4, 1, 3]
>>> maliste.reverse()
>>> maliste
[3, 1, 0.4, 3, 2, 1]
>>> maliste.sort()
>>> maliste
[0.4, 1, 1, 2, 3, 3]
```

Attention

Les méthodes `sort` *et* `reverse` *ne renvoient rien ! Il s'agit de méthodes de tri « sur place », ce qui veut dire qu'elles modifient directement la liste. Si l'on souhaite conserver une liste non triée en état, on doit la copier : nous verrons dans cette section comment y parvenir.*

La méthode `sort` accepte en paramètre une fonction qui permet de modifier la manière dont les éléments sont triés. Par défaut, il s'agit de la fonction `cmp` que nous avons vue dans le module `__builtins__`. Mais il est possible de spécifier une autre fonction (acceptant la même signature que `cmp`), par exemple :

```
>>> def reverse_cmp(v1, v2):
...     return -cmp(v1, v2)
...
>>> maliste.sort(reverse_cmp)
```

```
>>> maliste
[3, 3, 2, 1, 1, 0.4]
```

Remarque

La version de `reverse_cmp` *que nous présentons ici pour « inverser » une liste s'avère paradoxalement moins efficace que si l'on utilise successivement* `maliste.sort()` *puis* `maliste.reverse()`*, pour les motifs que nous avons exposés plus haut.*

Les slices (tranches)

Les séquences recèlent une autre merveille dans leur capacité à supporter les « sous-séquences », appelées *slices*. Nous avons vu comment indexer un élément d'une liste avec la syntaxe à crochets. Il est possible de faire référence à une partie de la liste avec la syntaxe suivante :

```
sequence[index_inferieur:index_superieur]
```

Ainsi, pour notre liste :

```
>>> print maliste
[3, 3, 2, 1, 1, 0.4]
>>> print maliste[2:4]
[2, 1]
```

Il en va de même pour les tuples :

```
>>> print montuple
(1, 2, 3)
>>> print montuple[2:10]
(3,)
```

Remarque

Nous observons dans l'exemple précédent que les indices de début et de fin des sous-séquences slices ne sont pas obligatoirement des valeurs d'indice valides : le glissement comprend tous les éléments entre 2 et 10 (10 non inclus), et si aucun élément n'a de correspondance, alors Python renvoie un slice vide.

Il est possible d'omettre l'un des indices. Si l'indice inférieur est omis, le slice commence au premier élément de la liste. Si l'indice supérieur est omis, le slice se termine au dernier élément de la liste :

```
>>> maliste[:4]
[3, 3, 2, 1]
>>> maliste[2:]
[2, 1, 1, 0.4]
```

Il est également possible de ne préciser aucun des indices : logiquement, la liste entière est retournée.

```
>>> maliste[:]
[3, 3, 2, 1, 1, 0.4]
```

Cette syntaxe est très importante, car elle permet d'effectuer la copie d'une liste. Observons le fragment de code suivant :

```
>>> liste2 = maliste
>>> liste3 = maliste[:]
>>> maliste.append(10)
>>> print maliste
[3, 3, 2, 1, 1, 0.4, 10]
>>> print liste2
[3, 3, 2, 1, 1, 0.4, 10]
>>> print liste3
[3, 3, 2, 1, 1, 0.4]
```

La variable `liste2` contient la même valeur que `maliste` : si l'une est modifiée, l'autre l'est également. En revanche, `liste3` contient une nouvelle liste.

Manipuler les dictionnaires

Ajout et destruction de valeurs

Considérons un dictionnaire `notes` qui contient les notes de trois élèves, Pierre, Paul et Jacques (on notera au passage cette indentation correcte d'un dictionnaire sur plusieurs lignes).

```
>>> notes = {
...     "pierre": [10, 5, 16],
...     "paul": [14, 13, 15],
...     "jacques": [17, 16, 16],
...     }
```

Chaque élément du dictionnaire contient une liste. Pour observer les (bonnes) notes de `jacques`, on utilise, comme nous l'avons déjà vu :

```
>>> print notes['jacques']
[17, 16, 16]
```

Un dictionnaire est par essence mutable. Nous pouvons donc ajouter un nouvel élève :

```
notes['thierry'] = [4, 8, 7]
>>> notes
{'jacques': [17, 16, 16], 'pierre': [10, 5, 16], 'thierry': [4, 8, 7], 'paul': [
14, 13, 15]}
```

Remarque

L'affichage à l'écran de dictionnaires est assez désagréable. Pour pallier cet inconvénient, Python propose un module `pprint` *qui permet d'afficher d'une manière élégante (et triée par clé pour un dictionnaire) le contenu d'une variable Python. Le code suivant montre comment on peut utiliser* `pprint` *sur notre exemple :*

```
>>> import pprint
>>> pprint.pprint(notes)
{'jacques': [17, 16, 16],
 'paul': [14, 13, 15],
 'pierre': [10, 5, 16],
 'thierry': [4, 8, 7]}
```

Si `pierre` quitte la classe, on détruit simplement son entrée dans la table des notes, en utilisant l'instruction (et pas la méthode) `del` :

```
>>> del notes['pierre']
>>> print notes
{'jacques': [17, 16, 16], 'thierry': [4, 8, 7], 'paul': [14, 13, 15]}
```

Attention

Comme on peut le constater sur l'exemple précédent, un dictionnaire n'est jamais ordonné par clé.

Méthodes d'un dictionnaire

Les dictionnaires ont eux aussi des méthodes que nous pouvons lister par `la syntaxe dir(`*`dictionnaire`*`)`. Les plus utilisées sont `keys`, `values` et `items`, qui permettent de donner la liste, respectivement, des clés, des valeurs et des tuples (clé, valeur) du dictionnaire :

Remarque

Les méthodes `keys`, `values` *et* `items` *ont donné leur nom aux méthodes* `objectKeys`, `objectValues` *et* `objectItems` *des dossiers de Zope.*

```
>>> print notes.keys()
['jacques', 'thierry', 'paul']
>>> print notes.values()
[[17, 16, 16], [4, 8, 7], [14, 13, 15]]
>>> print notes.items()
[('jacques', [17, 16, 16]), ('thierry', [4, 8, 7]), ('paul', [14, 13, 15])]
```

Remarque

L'ordre des éléments retournés est le même entre `keys`, `values` *et* `items`.

Ainsi, pour calculer la moyenne de chaque élève, on pourrait écrire le code suivant :

```
>>> for eleve in notes.keys():
...     somme = 0
...     for note in notes[eleve]:
...         somme = somme + note
...     print eleve, "a pour moyenne :", somme / len(notes[eleve])
...
jacques a pour moyenne : 16
thierry a pour moyenne : 6
paul a pour moyenne : 14
```

Voici une autre manière (un peu plus efficace) d'obtenir le même résultat :

```
>>> for eleve in notes.items():
...     somme = 0
...     for note in eleve[1]:
...         somme = somme + note
...     print eleve[0], "a pour moyenne :", somme / len(eleve[1])
...
jacques a pour moyenne : 16
thierry a pour moyenne : 6
paul a pour moyenne : 14
```

Pour récupérer un élément du dictionnaire sans recourir à la syntaxe à crochets, on peut utiliser la méthode `get`. Cette méthode accepte un paramètre qui permet de définir une valeur par défaut si la clé n'existe pas (cette valeur est `None` si elle n'est pas spécifiée). Ainsi :

```
>>> print notes.get('alfred')
None
>>> print notes.get('alfred', [])
[]
```

En revanche :

```
>>> notes['alfred']
Traceback (innermost last):
  File "<stdin>", line 1, in ?
KeyError: alfred
```

Pour tester si une clé se trouve dans le dictionnaire, on utilise la méthode `has_key`, comme suit :

```
>>> if notes.has_key('alfred'):
...     print "Alfred est dans le dictionnaire"
... elif notes.has_key('thierry'):
...     print "Alfred n'y est pas mais Thierry y est."
...
Alfred n'y est pas mais Thierry y est.
```

La méthode `clear()` permet d'effacer tout le dictionnaire. La méthode `update` permet de concaténer un dictionnaire à un autre :

```
>>> notes2 = {
...     "jules": [6, 7, 8],
...     "claude": [12, 10, 12],
...     }
>>> notes.update(notes2)
>>> pprint.pprint(notes)
{'claude': [12, 10, 12],
 'jacques': [17, 16, 16],
 'jules': [6, 7, 8],
 'paul': [14, 13, 15],
 'thierry': [4, 8, 7]}
```

Si une clé avait existé dans les deux dictionnaires, la valeur de `notes2` aurait été conservée, d'où le nom de la méthode : `update` (mettre à jour).

Enfin, comme les dictionnaires n'ont pas de slices, la méthode `copy` est disponible pour effectuer malgré tout une copie de dictionnaire.

Cette méthode est en fait assez rarement utilisée.

Quand les chaînes se déchaînent

Nous l'avons mentionné plus haut, une chaîne de caractères peut être vue comme une séquence (non mutable) de caractères. Le module `string` offre des outils pour manipuler les chaînes de caractères de Python.

> **Remarque**
> *Dans la version 1.5.2 de Python, les chaînes ne sont pas véritablement considérées comme des objets, contrairement aux listes, tuples et dictionnaires. Cela n'est plus vrai dans la version 2.0 de Python qui définit une dizaine de méthodes accessibles depuis un objet chaîne. Comme la version actuelle de Zope utilise la version 1.5.2 de Python, nous n'en ferons pas mention ici.*

Séquences d'échappement

Les chaînes de caractères Python supportent à peu près les mêmes règles d'échappement que les chaînes de caractères du C. Par exemple, `\n` représente un saut de ligne :

```
>>> print "Hello\nWorld"
Hello
World
```

Voici la liste exhaustive des codes d'échappement supportés :

`\saut de ligne`	*Ignoré*
`\\`	`\`
`\'`	`'`
`\"`	`"`
`\a`	*Signal sonore*
`\b`	*Retour arrière*
`\f`	*Saut de page*
`\n`	*Saut de ligne*
`\r`	*Retour chariot*
`\t`	*Tabulation*
`\v`	*Tabulation verticale*
`\000`	*Code ASCII 000 (octal)*
`\xhh`	*Code ASCII hh (hexadécimal)*

Chaînes brutes (raw)

Contrairement au C, les séquences d'échappement sont directement interprétées par le langage. Il est possible d'empêcher cette interprétation en préfixant la chaîne de la lettre `r` (ou `R`), signifiant « raw ». Observez la différence :

```
>>> print r"Hello\nWorld\n!"
Hello\nWorld\n!
>>> print "\""
"
>>> print r"\""
\"
```

Attention

Même en mode « raw », une chaîne ne doit jamais se terminer par `\"`.

Voici le cas d'une chaîne « raw » non valide :

```
>>> print r"\"
  File "<stdin>", line 1
    print r"\"
              ^
SyntaxError: invalid token
```

Formatage des chaînes

Python propose enfin une syntaxe similaire à celle de la fonction `printf` du C pour mettre en forme les chaînes de caractères composées de plusieurs « arguments ». Reprenons notre bout de code d'affichage des moyennes : pour la ligne d'affichage, nous avons écrit :

```
...     print eleve[0], "a pour moyenne :", somme / len(eleve[1])
```

Pour insérer dans une chaîne la valeur d'une variable, on utilise le signe `%` suivi du type de la variable (voir tableau ci-après), puis on fait suivre la chaîne du signe `%`, suivi lui-même d'un tuple d'arguments. Dans notre exemple, nous aurions pu (et même dû) écrire :

```
...     print "%s a pour moyenne %d" % (eleve[0], somme / len(eleve[1]))
```

Lorsqu'il n'y a qu'une variable à afficher, les parenthèses sont facultatives : on peut écrire

```
>>> print "Voici un nombre : %d" % 15
```

Cette syntaxe possède d'autres subtilités (par exemple, la possibilité de spécifier la précision affichée ou d'utiliser un dictionnaire à la place des variables) : se reporter au manuel de référence de Python pour plus d'informations. Voici la liste des types supportés :

`%%`	*Le caractère %*
`%c`	*Entier ou caractère*
`%d`	*Entier*
`%u`	*Entier non signé*
`%i`	*Entier*
`%s`	*Chaîne*
`%o`	*Entier (octal)*
`%x` *ou* `%X`	*Entier (hexadécimal)*
`%e` *ou* `%E`	*Flottant (avec exposant)*
`%f`	*Flottant*
`%g` *ou* `%G`	*Format de flottant le plus lisible*

Fonctions, paramètres et arguments

Arguments par défaut et arguments nommés

Nous avons vu plus haut comment on définit une fonction. Nous avons également vu que certaines fonctions de la bibliothèque de Python acceptent des arguments par défaut. La valeur par défaut des arguments d'une fonction peut être spécifiée avec le signe « = » :

```
>>> def test(arg1, arg2=0, arg3="Hello"):
...     print "arg1", arg1
```

```
...     print "arg2", arg2
...     print "arg3", arg3
...
>>> test(1)
arg1 1
arg2 0
arg3 Hello
>>> test("Hello", 53, 12)
arg1 Hello
arg2 53
arg3 12
```

Attention

Les arguments pour lesquels une valeur par défaut est spécifiée doivent toujours n'être suivis que d'arguments pour lesquels une valeur est aussi définie par défaut. Il serait par exemple interdit d'écrire `def test(arg1, arg2=0, arg3)` *car* `arg2` *est suivi d'au moins un argument sans valeur par défaut.*

Lors de l'appel d'une fonction, il est possible d'affecter les arguments d'après leur nom. Par exemple, pour appeler `test` avec un argument `arg1`, un argument `arg3` et en conservant la valeur par défaut de `arg2`, on utilise l'instruction suivante :

```
>>> test("Hello", arg3=12)
```

Fonctions à nombre variable d'arguments

Il est possible de créer des fonctions qui acceptent un nombre variable d'arguments, à l'image de la fameuse `printf` du langage C. Pour cela, on crée une fonction dont le dernier paramètre de la liste est précédé par le signe `*`, comme suit :

```
>>> def test(arg1, *args):
...     print "arg1", arg1
...     print "args", args
...
>>> test(1, 2, 3, 4, 5, 6)
arg1 1
args (2, 3, 4, 5, 6)
```

La variable `args` contient, sous forme de tuple, les paramètres qui ne sont pas affectés à d'autres paramètres de la fonction. Si `test` est appelée avec un seul paramètre, celui-ci est affecté à `arg1`, et `args` contient un tuple vide.

Cette syntaxe peut être étendue pour prendre en charge des arguments nommés : on utilise alors deux étoiles pour spécifier le nom de la variable (de type dictionnaire) qui comprendra les arguments nommés de la fonction, comme suit :

```
>>> def test(arg1, *args, **kw):
...     print "arg1", arg1
...     print "args", args
...     print "kw", kw
...
>>> test(1, 2, 3, 4, 5, 6)
arg1 1
args (2, 3, 4, 5, 6)
kw {}
>>> test(1, 2, 3, argX = 4, argY = 5)
arg1 1
args(2, 3)
kw {'argX': 4, 'argY': 5}
```

Ces constructions ne sont utilisées qu'avec parcimonie dans la plupart des programmes Python, mais Zope y fait massivement appel en interne.

Gestion des exceptions

Les erreurs de Python

La gestion des erreurs dans un programme est cruciale : c'est bien souvent sur ce point que la plupart des programmes sont mis en défaut. Un problème aussi simple que la protection en écriture d'un fichier peut provoquer l'arrêt de certains programmes.

La gestion des erreurs est une tâche généralement fastidieuse pour les programmeurs. À chaque appel de fonction, ils doivent tester la valeur de retour et, le cas échéant, appeler les routines qui traiteront le problème, libéreront les ressources ouvertes, etc.

Python (tout comme C++ et Java) propose une manière plus simple de gérer les erreurs, au travers d'exceptions.

> **Remarque**
>
> *La gestion des exceptions en DTML par Zope au moyen des balises* `<dtml-try>` *et* `<dtml-except>` *est fortement inspirée de celle de Python : le lecteur reconnaîtra donc ici les concepts exposés au chapitre 6, « DTML avancé ».*

Une exception est un problème qui est pris immédiatement en compte par le programme, ce qui provoque son déroutement dans une routine de niveau supérieur. Considérons le code suivant, qui ouvre un fichier, y lit quelque chose, puis le ferme :

```
>>> def lit (nom_fichier):
...     f = open (nom_fichier, 'r')
...     ret = f.read ()
...     f.close ()
```

```
...     return ret
...
```

Si on passe à la fonction un mauvais nom de fichier, Python signale quelque chose d'anormal en déclenchant une exception :

```
>>> lit ('xyz')
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in lit
IOError: [Errno 2] No such file or directory: 'xyz'
```

Si le message `Traceback` apparaît sur l'écran suivi d'une série d'informations, c'est qu'une exception a été déclenchée. Par défaut, en cas d'exception, Python affiche un message pour indiquer, d'une part, la ligne fautive et, d'autre part, le type d'erreur et sa valeur. Ici, comme nous travaillons dans l'interpréteur, Python ne connaît ni le nom du fichier (et affiche un point d'interrogation), ni le numéro de la ligne (la ligne 1 correspond à la définition de la fonction `lit`).

Il est possible de contrôler avec précision le comportement de Python lorsqu'une erreur se produit avec les instructions de gestion des exceptions. Par défaut, lorsqu'une erreur se produit, elle provoque immédiatement la sortie de Python avec un message d'erreur tel que celui qui figure dans l'exemple précédent. Ce comportement est bien entendu impensable dans une application digne de ce nom : lorsque l'utilisateur entre un mauvais nom de fichier, l'application devrait pouvoir intercepter l'erreur, la signaler (correctement) et demander à l'utilisateur un nouveau nom de fichier.

Traitement des exceptions

Pour qu'une exception soit traitée, il faut qu'elle soit capturée. En Python, comme dans la plupart des langages supportant les exceptions, une exception ne peut être capturée que dans un bloc protégé.

Un bloc protégé est introduit par l'instruction `try`, pour indiquer qu'il s'agit d'un bloc dans lequel une exception est susceptible de se produire.

Un bloc protégé doit être suivi immédiatement d'un bloc de traitement. Il y a deux types de blocs de traitement sous Python, introduits par deux instructions différentes, `except` et `finally` :

- Les blocs `except` ne sont exécutés que si une exception est déclenchée dans le bloc protégé.
- Les blocs `finally` sont exécutés soit immédiatement après qu'une exception a été déclenchée dans le bloc protégé, soit après l'exécution de toutes les instructions du bloc protégé.

Un exemple sera plus parlant :

```
>>> def AfficheFichier (nom):
...     try:
...             f=open (nom, 'r')
```

```
...             print f.read ()
...     except:
...             print "Erreur sur le fichier '%s'." % nom
...
>>> test ('xyz')
Erreur sur le fichier 'xyz'.
```

Dans la fonction `AfficheFichier`, si l'une des deux instructions du bloc `try` provoque une exception, celle-ci est interceptée par l'instruction `except`, et le nom du fichier fautif est imprimé à l'écran.

Si nous avions écrit `finally` au lieu d'`except`, l'instruction `print` aurait été exécutée dans tous les cas – ce qui n'est pas le comportement souhaité ici.

Remarque

Les blocs `try` *peuvent être imbriqués : dans ce cas, c'est le bloc* `except` *ou* `finally` *du bloc* `try` *le plus profond dans lequel s'est produite l'erreur qui est exécuté.*

Lorsqu'une exception est traitée par un bloc `finally`*, cette exception est à nouveau déclenchée à la fin du bloc* `finally` *: cela permet de placer du code de « nettoyage » dans le bloc ; on peut ainsi s'assurer qu'il sera exécuté et que l'erreur sera traitée à un plus haut niveau.*

Filtrage des exceptions

Si un programmeur ne souhaite traiter que certaines exceptions, il peut préciser, dans un bloc `except`, quels sont les types d'exceptions qu'il souhaite intercepter.

Ainsi, les instructions suivantes provoquent une exception de type `IndexError` :

```
>>> liste = []
>>> liste[0]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```

Un programmeur peut intercepter cette erreur en faisant suivre l'instruction `except` d'un tuple de types d'erreurs. Par exemple :

```
>>> try:
...     x = liste[0]
... except (IndexError, ):
...     print "La liste est vide !"
...
La liste est vide !
```

Il est également possible d'écrire une suite de plusieurs blocs `except`, chacun traitant des types d'exceptions différents. Le dernier bloc `except` peut ne préciser aucun type : dans ce cas, il sera exécuté si une exception d'un type non géré jusqu'à présent survient.

Enfin, la liste de blocs `except` peut être suivie de l'instruction `else:`. Dans ce cas, le bloc `else` sera exécuté si aucune exception n'intervient dans le bloc protégé.

Déclencher une exception

Le programmeur peut déclencher lui-même une exception, de quelque type que ce soit. Il faut pour cela utiliser l'instruction `raise` (le corps de la fonction n'est volontairement pas traité) :

```
>>> def test (liste_fichiers):
...     if type (liste_fichiers) != type ([]):
...             raise ValueError, "Vous devez passer en paramètre \
une liste de fichiers."
...     # ...traitements normaux...
...
>>> test (['abc', 'def', 'ghi'])
>>> test ('chaine')
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in test
ValueError: Vous devez passer en paramètre une liste de fichiers.
```

Lors du premier appel à `test`, aucune exception n'est déclenchée car la fonction reconnaît bien une liste. En revanche, au deuxième appel, une exception de type `ValueError` est déclenchée car une chaîne de caractères a été passée en paramètre au lieu d'une liste.

L'instruction `raise` est suivie d'un type, d'une virgule et d'une valeur. Le *type* et la *valeur* peuvent être n'importe quelle expression Python (classe, objet, chaîne, entier…). Idéalement, le type est une classe, et la valeur une instance de cette classe – cela n'est cependant pas toujours respecté.

Le type sert à distinguer l'exception (dans l'instruction `except`), tandis que la valeur permet, dans le bloc de gestion des exceptions, d'obtenir des informations supplémentaires sur l'exception. Souvent, pour simplifier, le type est un type d'erreur standard de Python (comme `ValueError`, `AttributeError`, `IndexError`, `KeyError`…), et la valeur est une chaîne de caractères décrivant l'erreur. Pour plus d'informations sur la récupération de la valeur d'erreur dans un bloc de gestion des exceptions, consulter la documentation de la fonction `exc_info` dans le manuel de référence de Python.

Il est possible de déclencher une exception dans un bloc `except` et `finally`. Dans le cas d'un bloc `except`, pour déclencher à nouveau la même exception (même type, même valeur), utilisez l'instruction `raise` sans préciser ni type ni valeur.

Remarque

N'oubliez pas que, à moins que vous ne déclenchiez explicitement une exception dans un bloc `finally`*, l'exception qui a le cas échéant provoqué un déroutement du programme dans ce bloc est à nouveau déclenchée à la fin de l'exécution du bloc.*

Protection de ressources

La fonction `AfficheFichier` que nous avons définie est correcte d'un strict point de vue syntaxique, mais pose un problème sur le plan sémantique. Que se passe-t-il si la fonction `open` s'exécute, mais que la fonction `read` provoque une exception ? Le fichier `f` restera ouvert. Certes, Python a la faculté de fermer automatiquement le fichier lorsque la variable `f` est détruite, mais toutes les ressources ne fonctionnent pas de la sorte : pour un accès à un moteur de données, par exemple, la base de données risquerait fort de rester ouverte – et de poser de graves problèmes de blocage de ressource.

La technique proposée pour éviter ce type d'erreur, très difficile à déceler lorsqu'un problème survient, est appelée la technique de protection de ressource. Elle s'exprime de la manière suivante : une demande de ressource (l'ouverture d'un fichier ou d'une base de données, par exemple) doit immédiatement être suivie d'un bloc `try`, puis d'un bloc `finally` dans lequel la première chose à faire est de libérer la ressource. Ainsi, la fonction `AfficherFichier` devrait s'écrire de la sorte (remarquez les deux blocs `try` imbriqués) :

```
>>> def AffichierFichier (nom):
...     try:
...         f = open (nom)
...         try:
...             print f.read ()
...         finally:
...             f.close ()
...     except:
...         print "Erreur sur le fichier '%s'." % nom
...
```

De la sorte, on aura l'assurance que, si une erreur survient sur le fichier, il sera de toute façon fermé.

Classes

Cette section propose un parcours du support de l'orientation objet par Python. Il ne s'agit pas d'un cours de programmation objet – simplement d'une description du comportement de Python en tant que langage orienté objet.

Qu'est-ce qu'un objet ?

En Python, un objet est une variable unique (son *identité* est garantie, mais nous détaillerons ce point plus avant) sur laquelle il est possible d'effectuer des opérations (au sens informatique du terme). Par exemple, une liste est un objet : elle est unique (l'emplacement mémoire qui lui fait référence ne contient que la liste) et il est possible d'y appliquer des opérations (ajout d'élément, référencement d'un élément, etc.).

Un objet peut être référencé par plusieurs noms. Ainsi :

```
>>> a = []
>>> b = a
>>> b.append(1)
>>> print a
[1]
```

On voit bien sur cet exemple que `a` et `b` font référence à un seul et même objet.

D'une manière générale, un objet est défini par trois choses :

- son identité ;
- son état ;
- son comportement.

L'identité d'un objet est obtenue grâce à la fonction `id`. Il s'agit, dans le cas de Python, de l'adresse de l'objet en mémoire. À un moment donné, on a la garantie absolue que deux objets distincts n'auront jamais la même identité. En revanche, la mémoire pouvant être recyclée, un emplacement mémoire peut être réutilisé, plus tard, pour un autre objet.

Les objets de Python (tels que les listes, les fichiers, les modules, etc.) sont d'une certaine manière figés : on ne peut pas agir sur leur structure – c'est-à-dire sur la définition de leur comportement. On peut certes agir sur l'objet lui-même : ajouter un élément à la liste, ouvrir ou fermer un fichier, etc. Mais on ne peut pas décider que cette liste supportera une nouvelle opération, par exemple.

En revanche, on peut créer nos propres structures d'objet, en les définissant – tout comme on définit une fonction pour pouvoir l'utiliser par la suite. Cette définition de la structure d'un objet s'appelle une *classe*.

Remarque

Pour compliquer les choses, une classe est un objet en Python… En fait, cette propriété garantit l'homogénéité du langage et permet d'effectuer des opérations sur les classes elles-mêmes.

On définit une classe avec le mot-clé `class`.

Voici la plus minimale qui soit :

```
>>> class MaClasse:
...     pass
...
```

Et on obtient un objet de cette classe comme suit (on appelle cela *instancier* un objet) :

```
>>> obj = MaClasse()
>>> obj
```

Cette classe ne définit aucune structure et ne fait rien, strictement rien. Il est possible de définir une classe un peu moins inutile, en lui affectant des variables et/ou des fonctions :

```
>>> class Hello:
...     chaine = "Hello, World !"
...     def Afficher(self):
...         print self.chaine
...
>>> obj = Hello()
>>> obj.chaine
'Hello, World !'
>>> obj2 = Hello()
>>> obj.chaine = "Bonjour !"
>>> obj2.chaine
'Hello, World !'
>>> obj.chaine
'Bonjour !'
```

La classe `Hello` contient un *attribut*, ou *propriété*, `chaîne`. Elle contient également une méthode : `Afficher` (nous verrons ultérieurement comment on écrit une méthode).

Les objets obtiennent la valeur de l'attribut de la classe lorsqu'ils sont instanciés. Ils peuvent alors la modifier sans impacter les autres instances.

Il est également possible de référencer des *attributs d'instance*, voire de les créer s'ils n'existent pas :

```
>>> obj.autre_chaine = "Bonjour !"
>>> obj.autre_chaine
'Bonjour !'
>>> obj2.autre_chaine
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: autre_chaine
```

Ici, l'attribut `autre_chaine` a été créé pour l'objet `obj` et n'existe pas pour les autres objets de classe `MaClasse`. On peut éventuellement détruire cet attribut avec l'instruction `del` :

```
>>> del obj.autre_chaine
>>> obj.autre_chaine
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: autre_chaine
```

Il est enfin possible d'appeler la méthode `Afficher`, en utilisant la notation à point :

```
>>> obj.Afficher()
Hello, World !
```

Méthodes

Observez la classe suivante :

```
class Hello:
    chaine = "Hello, World !"
    def Afficher():
        print chaine
```

Logiquement, lorsqu'on instancie un objet de classe `Hello` et qu'on appelle sur lui la méthode `Afficher`, on s'attend à ce que la méthode « Hello, World ! » s'affiche à l'écran. Et pourtant, ce n'est pas le cas : d'une part, Python refuse d'exécuter la méthode et, d'autre part, quand bien même il y arriverait, il indiquerait que la variable `chaine` n'existe pas.

En fait, la portée des variables de Python est soit locale, soit globale, mais la notion de « portée de classe » n'existe pas en Python. Si la variable `chaine` était définie en dehors de la classe `Hello`, que se passerait-il ? Comment le programmeur ferait-il pour décider qu'il veut afficher la `chaine` extérieure à la classe plutôt que la `chaine` définie par la classe ?

Pour éliminer ce problème, lors de l'appel d'une méthode, Python passe systématiquement l'objet comme premier argument. La référence à toute variable de classe ou d'objet se fait alors explicitement, avec la notation à point. Par convention, l'argument représentant l'objet est appelé `self`. Voici notre exemple, corrigé :

```
class Hello:
    chaine = "Hello, World !"
    def Afficher(self)
        print self.chaine
```

On peut cette fois instancier un objet de classe `Hello` et appeler la méthode `Afficher` : la variable est correctement référencée.

Du point de vue de Python, les deux lignes suivantes sont donc parfaitement équivalentes :

```
obj.Afficher()
Hello.Afficher(obj)
```

Bien entendu, les méthodes peuvent spécifier d'autres arguments ; voici un autre exemple de classe :

```
class Hello:
    chaine = "Hello, World !"
    def Afficher(self)
        self.AfficherChaine(self.chaine)

    def AfficherChaine(self, chaine):
        print chaine
```

Méthodes particulières

Instanciation et destruction

Lors de l'instanciation d'un objet, il peut s'avérer pratique d'effectuer certaines actions (initialisation, par exemple). Pour cela, une méthode `__init__` est à disposition du programmeur. De même, la méthode `__del__` est appelée lorsque l'objet est détruit :

```
class Hello:
    def __init__(self):
        print "Hello, World !"

    def __del__(self):
        print "Good bye, cruel world."

obj = Hello()
del obj
```

La ligne `obj = Hello()` aura pour effet d'exécuter `__init__`. La ligne `del obj` provoquera l'appel de `__del__`.

Il est également possible de spécifier d'autres arguments pour `__init__` :

```
class Hello:
    def __init__(self, chaine):
        self.chaine = chaine
```

Dans ce cas, lors de l'instanciation, il faut passer obligatoirement l'argument `chaine`, comme suit :

```
>>> x = Hello("Hello, World !")
Hello, World !
```

> **Remarque**
> *La méthode `__del__`, en revanche, accepte un seul et unique attribut, `self`.*

Méthode d'appel : `__call__`

Il est également possible de définir une méthode particulière, `__call__`, qui est utilisée intensivement dans Zope. Cette méthode permet de rendre possible l'appel de l'instance d'une classe, comme s'il s'agissait d'une fonction. Considérons l'exemple suivant :

```
class Hello:
    def __init__(self, chaine):
        self.chaine = chaine

    def __call__(self):
        print self.chaine
```

Il est possible d'utiliser la classe `Hello` comme suit :

```
>>> x = Hello("Hello, World !")
>>> x()
Hello, World !
```

Dans cet exemple, l'objet `x` a été utilisé comme une fonction. Il est possible de spécifier des arguments à la méthode `__call__`, comme pour `__init__`. Dans ce cas, les paramètres correspondants doivent être mentionnés lors de « l'appel » de l'objet :

```
class Hello:
    def __call__(self, chaine):
        print chaine
```

Cet exemple s'utilise comme suit :

```
>>> x = Hello()
>>> x('Hello, World !')
Hello, World !
```

Nous verrons, lors de l'étude des produits dans le chapitre 12, que la méthode `__call__` est invoquée par Zope pour rendre un objet.

Autres méthodes

Il est possible de surcharger d'autres méthodes pour modifier ou étendre le comportement des instances d'une classe. L'annexe B en dresse une liste à titre d'information.

Héritage

Hériter d'une classe consiste à créer une nouvelle classe qui reprend l'intégralité (ou presque) des informations de structure de la première classe (appelée classe de base). La classe ainsi résultante est appelée « classe dérivée ».

Considérons par exemple une classe que nous appelons `Folder`, et qui représente la structure d'un dossier Zope :

```
class Folder
    # ...l'implémentation réelle n'a pas d'importance
    pass
```

Nous pouvons définir une nouvelle classe, `DossierRecherche`, supportant une méthode `RechercherDossier` qui est en mesure de trouver une information dans les objets de `Folder` :

```
class DossierRecherche(Folder):
    def RechercherDossier(self, argument):
        # ...l'implémentation réelle n'a pas d'importance
        pass
```

La méthode `RechercherDossier` peut alors utiliser toutes les méthodes et tous les attributs définis par la classe `Folder` : elle a hérité la structure et le comportement de la classe `Folder`.

Remarque

Nous verrons dans les chapitres consacrés aux ZClasses et aux produits que l'implémentation d'une telle méthode peut tout à fait être réalisée avec Zope…

Lorsqu'on parle de recherche sous Zope, il est naturel de penser au ZCatalog (voir chapitre 7) : nous pourrions très bien faire dériver notre classe `DossierRecherche` des classes `Folder` et ZCatalog ; il suffit de rajouter le nom des autres classes de base à la suite, séparés par des virgules (on appelle ce procédé l'*héritage multiple*) :

```
class DossierRecherche(Folder, ZCatalog):
```

Attention

Lors de l'héritage, une méthode n'est jamais héritée implicitement : il s'agit de `__init__`*. Pour que la classe dérivée appelle tout de même les méthodes* `__init__` *des classes de base, il suffit d'utiliser le procédé présenté plus haut :* `classe_de_base.__init__(self)` *dans le corps de la méthode* `__init__` *de la classe dérivée.*

On verra lors de l'étude des produits qu'il n'est pas possible d'utiliser ce procédé lorsqu'on hérite de classes Zope, et ce, en raison même de la façon dont Zope utilise le langage C.

En résumé…

Dans ce chapitre, nous avons passé en revue les grands principes du langage Python. Nous avons notamment étudié la syntaxe du langage, les principales constructions (expressions, fonctions, boucles, fonctions…).

Nous avons également évoqué la gestion des modules par Python, ainsi que quelques fonctions importantes proposées par la bibliothèque standard de Python.

Nous en avons détaillé certains aspects : la manipulation des séquences et des dictionnaires, des paramètres fonctions et la gestion des exceptions par Python. Nous avons enfin étudié la gestion des concepts objets par Python, avec l'étude des classes et des instances de classes.

Tout au long de ce chapitre, nous avons souligné les points du langage qui sont en rapport avec Zope : nous allons donc mettre toute cette théorie en pratique, et apprendre, dans le chapitre suivant, à intégrer des fonctions ou modules écrits en Python à Zope.