Bob Cordeau & Laurent Pointal

Une introduction à Python 3

version 1.6180

Remerciements

Grand merci à Hélène Cordeau pour ses illustrations ; les aventures de *Pythoon* enchantent les têtes de paragraphe.

Merci à Tarek Ziadé pour les emprunts à ses publications, en particulier nous remercions les éditions **Eyrolles** pour leur aimable autorisation de publier le dialogue de la page 110 et les éditions **Dunod** pour leur aimable autorisation de publier les exemples des pages 102, 104 et 107. Merci à Xavier Olive pour les emprunts à son papier « Introduction à l'écriture chinoise ».

Merci à Cécile Trevian pour son aide à la traduction du « Zen de Python ».

Merci à Stéphane Gimenez et à Claude Leloup pour leur relecture *en profondeur*.

Une pensée spéciale pour Stéphane Barthod : son enseignement didactique auprès des étudiants et son engagement envers ses collègues ne seront pas oubliés.

Enfin il faudrait saluer tous les auteurs butinés sur Internet…

En se partageant le savoir ne se divise pas, il se multiplie.

Sommaire

Avant-propos

À qui s'adresse ce cours ?

Utilisé à l'origine par les étudiants de Mesures Physiques de l'IUT d'Orsay, ce cours s'adresse plus généralement à toute personne désireuse d'apprendre Python en tant que premier langage de programmation.

Ce cours se veut essentiellement pratique. D'une part en fournissant de très nombreux exemples et une vingtaine d'exercices corrigés, et d'autre part en proposant plusieurs moyens de naviguer dans le document : un sommaire en début et une table des matières détaillée en fin, un index et un glossaire, tous deux munis d'hyperliens.

Nos choix

Cette introduction reposent sur quelques partis pris :

- la version 3 choix du langage Python, version qui abolit la compatibilité descendante [1] avec la série des versions 2.x dans le but d'éliminer les faiblesses originelles du langage ;
- le choix de logiciels libres :
  - la distribution Python scientifique Pyzo comprenant notamment l'éditeur IEP et l'outil IPython notebook,
  - des outils *open source* de production de documents : le moteur XeTeX [2], le script latexmk [3], l'éditeur TeXworks et le logiciel de création de diagrammes Dia ;
- enfin sur l'abondance des ressources et de la documentation sur le Web !

Numéro de version ?

Suivant l'exemple de Donald Knuth, l'inventeur du logiciel de composition TeX, le numéro de la i[e] version de ce document, au lieu d'être « 1.1 » ou « 2.0.3 », est la i[e] décimale d'un nombre célèbre [4].

Conventions

Conventions typographiques :

- le texte courant utilise la police : Linux Libertine ;
- la couverture et les en-têtes de chapitre utilisent la police : **Linux Biolinum** ;
- le code utilise la police : `DejaVu Sans Mono`.

Un script Python complet sera présenté sous la forme :

```
# -*- coding: utf8 -*-
"""Un exemple de script."""

# Import ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
import math

# Programme principal ========================================================
print("pi / 2 =", math.pi / 2)
```

1. C'est une grave décision, mûrement réfléchie : « Un langage qui bouge peu permet une industrie qui bouge beaucoup » (Bertrand Meyer).
2. Le « moteur » est le programme qui permet de composer ce texte. Il est associé au format XeLaTeX, ensemble de macros simplifiant l'écriture.
3. Permet d'automatiser la composition.
4. $\varphi = \frac{1+\sqrt{5}}{2}$, le nombre d'or.

Le code Python interprété sera présenté sous la forme :

```
>>> import math

>>> print("pi / 2 =", math.pi/2)
pi / 2 = 1.5707963267948966
```

Des commandes textuelles seront présentées sous la forme :

```
Une commande ou un fichier "texte".
```

Enfin, on mettra en lumières des parties de texte de la façon suivante :

Définition

✍ Une boîte de définition.

Remarque

✓ Une boîte de remarque.

Syntaxe

✎ Une boîte de syntaxe.

Attention

☢ Une boîte d'alerte.

Les exercices

Des exercices corrigés sous forme de *notebook* accompagnent ce cours. Pour plus de détails, cf. annexe E p. 123.

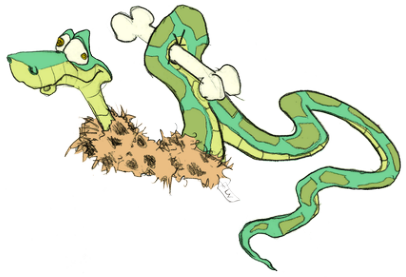
Pour joindre les auteurs

✉ pycours@kordeo.eu
✉ laurent.pointal@limsi.fr

FIGURE 1 – Creative Commons By-Nc-Sa

Chapitre 1

Introduction



Ce premier chapitre introduit les grandes caractéristiques du langage Python, le replace dans l'histoire des langages informatiques, donne les particularités de production des programmes, définit la notion si importante d'algorithme et conclut sur les divers implémentations disponibles.

1.1 Principales caractéristiques du langage Python

Historique :

- 1991 : Guido van Rossum travaille aux Pays-Bas [1] sur le projet AMOEBA, un système d'exploitation distribué. Il conçoit Python à partir du langage ABC et publie la version 0.9.0 sur un forum Usenet,
- 1996 : sortie de *Numerical Python*, ancêtre de *numpy*,
- 2001 : naissance de la PSF (Python Software Fundation)
- les versions se succèdent… Un grand choix de modules est disponible, des colloques annuels sont organisés, Python est enseigné dans plusieurs universités et est utilisé en entreprise…
- 2006 : première sortie de IPython,
- fin 2008 : sorties simultanées de Python 2.6 et de Python 3.0,
- 2013 : versions en cours des branches 2 et 3 : v2.7.3 et v3.3.0 ;

Langage open source :

- licence open source CNRI, compatible GPL, mais sans la restriction *copyleft*. Donc Python est libre et gratuit même pour les usages commerciaux,
- GvR (Guido van Rossum) est le « BDFL » (dictateur bénévole à vie !),
- importante communauté de développeurs,
- nombreux outils standard disponibles : *Batteries included* ;

Travail interactif :

- nombreux interpréteurs interactifs disponibles (notamment IPython),
- importantes documentations en ligne,
- développement rapide et incrémentiel,
- tests et débogage outillés,
- analyse interactive de données ;

Langage interprété rapide :

- interprétation du *bytecode* compilé,
- de nombreux modules sont disponibles à partir de bibliothèques optimisées (souvent écrites en C ou C++) ;

Simplicité du langage : (cf. annexe A p. 109) :

- syntaxe claire et cohérente,

1. au CWI : Centrum voor Wiskunde en Informatica.

- indentation significative,
- gestion automatique de la mémoire (*garbage collector*),
- Typage dynamique fort : pas de déclaration ;

Orientation objet :

- modèle objet puissant mais pas obligatoire,
- structuration multifichier aisée des applications : facilite les modifications et les extensions,
- les classes, les fonctions et les méthodes sont des objets dits *de première classe*. Ces objets sont traités comme tous les autres (on peut les affecter, les passer en paramètre) ;

Ouverture au monde :

- interfaçable avec C/C++/FORTRAN,
- langage de script de plusieurs applications importantes,
- excellente portabilité ;

Disponibilité de bibliothèques :

- plusieurs milliers de packages sont disponibles dans tous les domaines.

On définit parfois Python comme un langage algorithmique exécutable.

1.2 Environnements matériel et logiciel

1.2.1 L'ordinateur

Définition

✍ On peut schématiser la définition de l'ordinateur de la façon suivante : **automate déterministe à composants électroniques**.

L'ordinateur comprend entre autres :

- un microprocesseur avec une UC (Unité de Contrôle), une UAL (Unité Arithmétique et Logique), une horloge, une mémoire cache rapide ;
- de la mémoire volatile (dite *vive* ou RAM), contenant les instructions et les données nécessaires à l'exécution des programmes. La RAM est formée de cellules binaires (*bits*) organisées en mots de 8 bits (*octets*) ;
- des périphériques : entrées/sorties, mémoires permanentes (dites *mortes* : disque dur, clé USB, CD-ROM…), réseau…

1.2.2 Deux sortes de programmes

On distingue, pour faire rapide :

- le système d'exploitation : ensemble des programmes qui gèrent les ressources matérielles et logicielles. Il propose une aide au dialogue entre l'utilisateur et l'ordinateur : l'interface textuelle (interpréteur de commande) ou graphique (gestionnaire de fenêtres). Il est souvent multitâche et parfois multiutilisateur ;
- les programmes applicatifs sont dédiés à des tâches particulières. Ils sont formés d'une série de commandes contenues dans un programme *source* qui est transformé pour être exécuté par l'ordinateur.

1.3 Langages

1.3.1 Des langages de différents niveaux

- Chaque processeur possède un langage propre, directement exécutable : le langage machine. Il est formé de 0 et de 1 et n'est pas portable, c'est le *seul* que l'ordinateur puisse utiliser ;
- le langage d'assemblage est un codage alphanumérique du langage machine. Il est plus lisible que le langage machine, mais n'est toujours pas portable. On le traduit en langage machine par un *assembleur* ;
- les langages de haut niveau. Souvent normalisés, ils permettent le portage d'une machine à l'autre. Ils sont traduits en langage machine par un *compilateur* ou un *interpréteur*.

> Retour *Table des matières*

1.3.2 Très bref historique des langages

— Années 50 (approches expérimentales) : FORTRAN, LISP, COBOL, ALGOL…
— années 60 (langages universels) : PL/1, Simula, Smalltalk, Basic…
— années 70 (génie logiciel) : C, PASCAL, ADA, MODULA-2…
— années 80 (programmation objet) : C++, LabView, Eiffel, Perl, VisualBasic…
— années 90 (langages interprétés objet) : Java, tcl/Tk, Ruby, Python…
— années 2000 (langages commerciaux propriétaires) : C#, VB.NET…

Des centaines de langages ont été créés, mais l'industrie n'en utilise qu'une minorité.

1.4 Production des programmes

1.4.1 Deux techniques de production des programmes

La compilation est la traduction du *source* en langage *objet*. Elle comprend au moins quatre phases (trois phases d'analyse — lexicale, syntaxique et sémantique — et une phase de production de code objet). Pour générer le langage machine il faut encore une phase particulière : l'édition de liens. La compilation est contraignante, mais offre au final une grande vitesse d'exécution.

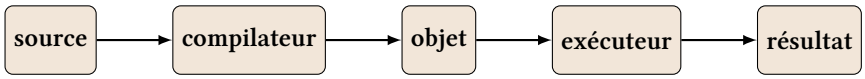


Figure 1.1 – Chaîne de compilation

Dans la technique de l'interprétation chaque ligne du *source* analysé est traduite au fur et à mesure en instructions directement exécutées. Aucun programme objet n'est généré. Cette technique est très souple, mais les codes générés sont peu performants : l'interpréteur doit être utilisé à chaque nouvelle exécution…

Figure 1.2 – Chaîne d'interprétation

1.4.2 Technique de production de Python

— Technique mixte : l'interprétation du bytecode compilé. Bon compromis entre la facilité de développement et la rapidité d'exécution ;
— le *bytecode* (forme intermédiaire) est portable sur tout ordinateur muni de la machine virtuelle Python.

Figure 1.3 – Interprétation du bytecode compilé

Pour **exécuter un programme**, Python charge le fichier source `.py` en mémoire vive, en fait l'analyse (lexicale, syntaxique et sémantique), produit le bytecode et enfin l'exécute.

Afin de ne pas refaire inutilement toute la phase d'analyse et de production, Python sauvegarde le bytecode produit (dans un fichier `.pyo` ou `.pyc`) et recharge simplement le fichier bytecode s'il est plus récent que le fichier source dont il est issu.

En pratique, il n'est pas nécessaire de compiler explicitement un module, Python gère ce mécanisme de façon transparente.

1.4.3 Construction des programmes

Le génie logiciel étudie les méthodes de construction des programmes. Plusieurs modèles sont envisageables, entre autres :

- la méthodologie procédurale. On emploie l'analyse descendante (division des problèmes) et remontante (réutilisation d'un maximum de sous-algorithmes). On s'efforce ainsi de décomposer un problème complexe en sous-programmes plus simples. Ce modèle structure d'abord les actions ;
- la méthodologie objet. Centrée sur les données, elle est considérée plus stable dans le temps et meilleure dans sa conception. On conçoit des fabriques (*classes*) qui servent à produire des composants (*objets*) qui contiennent des données (*attributs*) et des actions (*méthodes*). Les classes dérivent (*héritage* et *polymorphisme*) de classes de base dans une construction hiérarchique.

Python offre les *deux* techniques.

1.5 Algorithme et programme

1.5.1 Définitions

Définition

✍ Algorithme : ensemble des étapes permettant d'atteindre un but en répétant un nombre fini de fois un nombre fini d'instructions.

Donc un algorithme se termine en un temps fini.

Définition

✍ Programme : un programme est la traduction d'un algorithme en un langage compilable ou interprétable par un ordinateur.

Il est souvent écrit en plusieurs parties dont une qui *pilote* les autres : le programme principal.

1.5.2 Présentation des programmes

Un programme *source* est destiné à l'être humain. Pour en faciliter la lecture, il doit être judicieusement *présenté* et *commenté*.

La signification de parties non triviales (et uniquement celles-ci) doit être expliquée par un commentaire. En Python, un commentaire commence par le caractère # et s'étend jusqu'à la fin de la ligne :

```
#~~~~~~~~~~~~~~~~~~~~
# Voici un commentaire
#~~~~~~~~~~~~~~~~~~~~

9 + 2 # en voici un autre
```

1.6 Implémentations de Python

Remarque

✓ Une « implémentation » signifie une « mise en œuvre ».

- **CPython** : *Classic Python*, codé en C, implémentation portable sur différents systèmes ;
- **MicroPython** : version optimisée et allégée de Python 3 pour système embarqué. Cf. par exemple le site `http://wiki.mchobby.be/index.php?title=MicroPython-Accueil` ;
- **Jython** : ciblé pour la JVM (utilise le bytecode de JAVA) ;
- **IronPython** : *Python.NET*, écrit en C#, utilise le MSIL (*MicroSoft Intermediate Language*) ;
- **Stackless Python** : élimine l'utilisation de la pile du langage C (permet de récurser tant que l'on veut) ;
- **Pypy** : projet de recherche européen d'un interpréteur Python écrit en Python.

1.7 La distribution Pyzo

1.7.1 Présentation

Parmi la multitude de distributions Python, le choix de `Pyzo` est particulièrement pertinent pour l'enseignement. `Pyzo` est une distribution scientifique simple : la version en cours (`2015a`) comprend Python version 3.4.2, l'interpréteur `IPython` version `2.4.1`, notamment `IPython notebook` qui encourage des pratiques modernes de la programmation (travail collaboratif, *literate computing*, ...), `IEP` [1] un éditeur qui offre les principales fonctionnalités d'un environnement de développement intégré, les principales bibliothèques scientifiques (`Numpy`, `Scipy`, `Matplotlib`...) et graphiques (`tkinter`, `pyside`).

1.7.2 Installation

ATTENTION ! texte à reprendre pour la distribution à jour.

`Pyzo` existe pour les trois principaux systèmes d'exploitation : Windows, Mac OS X et GNU/Linux. Son installation est grandement facilité par le fait qu'il n'est pas nécessaire d'être administrateur et que toute la distribution est contenue dans un seul fichier [2]. En outre, la distribution est portable : on peut décompresser `Pyzo` sur une clé USB.

1.7.3 Utilisation

Dans le cadre de ce cours, la distribution `Pyzo` permet :
A terminer...

1. `IEP` : *Interactive Editor for Python.*
2. téléchargeable à l'adresse `http://www.pyzo.org/downloads.html`

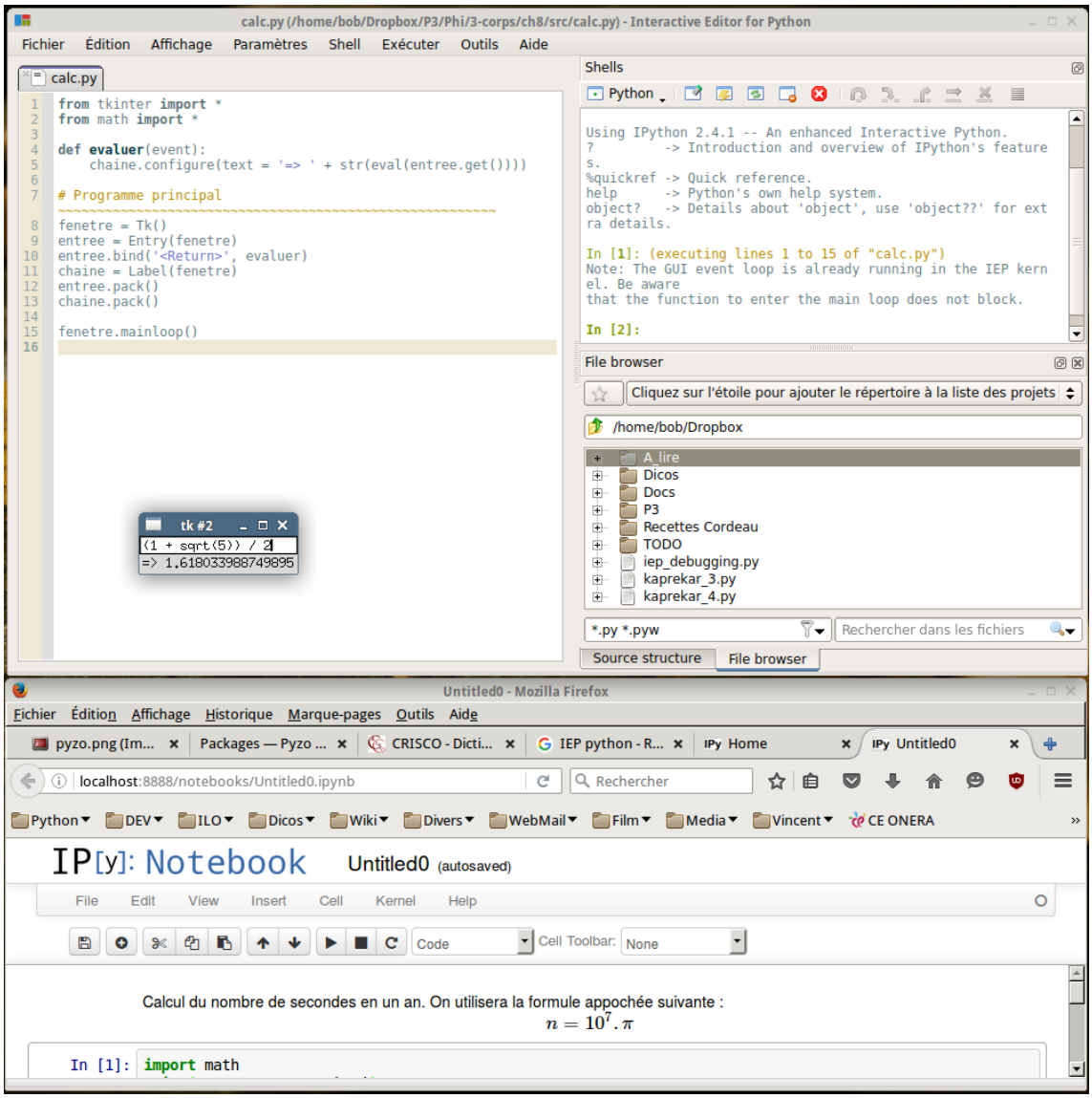


FIGURE 1.4 – L'environnement de programmation Pyzo.

> Retour *Table des matières*

Chapitre 2

La calculatrice Python

Comme tout langage, Python permet de manipuler des données grâce à un *vocabulaire* de mots réservés et grâce à des *types de données* – approximation des ensembles de définition utilisés en mathématique.

Ce chapitre présente les règles de construction des identificateurs, les types de données simples (les conteneurs seront examinés au chapitre 4) ainsi que les types chaîne de caractères.

Enfin, *last but not least*, ce chapitre s'étend sur les notions non triviales de variable, de référence d'objet et d'affectation.

2.1 Les modes d'exécution

2.1.1 Les deux modes d'exécution d'un code Python

— Soit on enregistre un ensemble d'instructions Python dans un fichier grâce à un éditeur (on parle alors d'un *script Python*) que l'on exécute par une commande ou par une touche du menu de l'éditeur ;
— soit on utilise l'interpréteur Python embarqué qui exécute la *boucle d'évaluation* [1] (☞ Fig. 2.1).

```
>>> 5 + 3      Python affiche l'invite. L'utilisateur tape une expression.
8              Python évalue et affiche le résultat...
>>>            ... puis ré-affiche l'invite.
```

Figure 2.1 – La boucle d'évaluation de l'interpréteur Python (*REPL*)

2.2 Identificateurs et mots clés

2.2.1 Identificateurs

Comme tout langage, Python utilise des *identificateurs* pour nommer ses objets.

Définition

✍ Un identificateur Python est une suite non vide de caractères, de longueur quelconque, formée d'**un** *caractère de début* (n'importe quelle lettre Unicode [2], y compris le caractère souligné) et de **zéro ou plusieurs** *caractères de continuation* (un caractère de début ou un chiffre).

1. En anglais *REPL* (*Read-Eval-Print Loop*).
2. cf. annexe C p. 116.

Attention

☢ Les identificateurs sont sensibles à la casse et ne doivent pas être un mot réservé de Python 3 (voir le § « Les mots réservés de Python 3 ci-dessous »).

2.2.2 Style de nommage

Il est important d'utiliser une politique cohérente de nommage des identificateurs. Voici le style utilisé dans ce document [1] :

- `NOM_DE_MA_CONSTANTE` pour les constantes ;
- `maFonction`, `maMethode` pour les fonctions et les méthodes ;
- `MaClasse` pour les classes ;
- `UneExceptionError` pour les exceptions ;
- `nom_de_ma_variable` pour les variables et pour tous les autres identificateurs.

Exemples :

```
NB_ITEMS = 12 # UPPER_CASE_WITH_UNDERSCORES
class MaClasse: pass # camelCase
def maFonction(): pass # mixedCase
mon_id = 5 # lower_case_with_underscores
```

Pour ne pas prêter à confusion, éviter d'utiliser les caractères `l` (minuscule), `O` et `I` (majuscules) seuls.
Enfin, on évitera d'utiliser les notations suivantes :

```
_xxx # usage interne
__xxx # attribut de classe
__xxx__ # nom spécial réservé
```

2.2.3 Les mots réservés de Python 3

La version 3.4 de Python compte 33 mots clés :

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

2.3 Notion d'expression

Définition

✍ Une expression est une portion de code que l'interpréteur Python peut évaluer pour obtenir une valeur.

Les expressions peuvent être simples ou complexes. Elles sont formées d'une combinaison de littéraux représentant directement des valeurs, d'identificateurs et d'opérateurs.

```
>>> id1 = 15.3

>>> id2 = id1 - 13

>>> if id2 > 0:
... id3 = 7
... else:
... id3 = -4
```

1. Voir les détails dans la PEP 8 : « Style Guide for Python », Guido van Rossum et Barry Warsaw

2.4 Les types de données entiers

Définition

✍ Le *type* d'un objet Python détermine de quelle sorte d'objet il s'agit.
L'instruction `type()` fournit le type d'une valeur.

Python offre deux types entiers standard : `int` et `bool`.

2.4.1 Le type `int`

Le type `int` n'est limité en taille que par la mémoire de la machine [1].
Les entiers littéraux sont décimaux par défaut, mais on peut aussi utiliser les bases suivantes :

```
>>> 2013 # décimal (base 10)
2013

>>> 0b11111011101 # binaire (base 2)
2013

>>> 0o3735 # octal (base 8)
2013

>>> 0x7dd # hexadecimal (base 16)
2013
```

Opérations arithmétiques

Les principales opérations :

```
>>> 20 + 3
23

>>> 20 - 3
17

>>> 20 * 3
60

>>> 20 ** 3
8000

>>> 20 / 3
6.666666666666667

>>> 20 // 3 # division entière
6

>>> 20 % 3 # modulo (reste de la division entière)
2

>>> abs(3 - 20) # valeur absolue
17
```

Bien remarquer le rôle des deux opérateurs de division :

/ : produit une division flottante, même entre deux entiers ;

// : produit une division entière.

Bases usuelles

Un entier écrit en base 10 (par exemple 179) peut être représenté en binaire, octal et hexadécimal en utilisant les syntaxes suivantes :

1. Dans la plupart des autres langages les entiers sont codés sur un nombre fixe de bits et ont un domaine de définition limité auquel il convient de faire attention.

```
>>> 0b10110011 # binaire
179

>>> bin(179)
'0b10110011'

>>> 0o263 # octal
179

>>> oct(179)
'0o263'

>>> 0xB3 # hexadécimal
179

>>> hex(179)
'0xb3'
```

Les bases arithmétiques

Définition

En arithmétique, une base n désigne la valeur dont les puissances successives interviennent dans l'écriture des nombres, ces puissances définissant l'ordre de grandeur de chacune des positions occupées par les chiffres composant tout nombre.
Par exemple : $57_n = (5 \times n^1) + (7 \times n^0)$

Certaines bases sont couramment employées :

- la base 2 (système binaire), en électronique numérique et informatique ;
- la base 8 (système octal), en informatique ;
- la base 16 (système hexadécimal), fréquente en informatique ;
- la base 60 (système sexagésimal), dans la mesure du temps et des angles.

Les changements de base

Un nombre dans une base n donnée s'écrit sous la forme d'addition des puissances successives de cette base.

Exemples

$$57_{16} = (5 \times 16^1) + (7 \times 16^0) = 87_{10}$$
$$57_8 = (5 \times 8^1) + (7 \times 8^0) = 47_{10}$$

2.4.2 Le type `bool`

Principales caractéristiques du type `bool` [1] :
— deux valeurs possibles : `False`, `True` ;
— opérateurs de comparaison entre deux valeurs comparables, produisant un résultat de type `bool` : ==, !=, >, >=, < et <=

```
>>> 2 > 8
False

>>> 2 <= 8 < 15
True
```

— opérateurs logiques : `not`, `or` et `and`.
En observant les tables de vérité des opérateurs `and` et `or`, on remarque que :
— dès qu'un premier membre a la valeur `False`, l'expression `False and expression2` vaudra `False`. On n'a donc pas besoin d'évaluer `expression2`,
— de même dès qu'un premier membre a la valeur `True`, l'expression `True or expression2` vaudra `True` ;
Cette optimisation est appelée « principe du *shortcut* » ou évaluation « au plus court » :

1. Nommé d'après George Boole, logicien et mathématicien britannique du XIX^e siècle.

```
>>> (3 == 3) or (9 > 24)
True

>>> (9 > 24) and (3 == 3)
False
```

Les opérateurs booléens de base

En Python les valeurs des variables booléennes sont notées `False` et `True`. Les opérateurs sont notés respectivement `not`, `and` et `or`.

Opérateur unaire `not`

a	not(a)
False	True
True	False

Opérateurs binaires `or` et `and`

a	b	a or b	a and b
False	False	False	False
False	True	True	False
True	False	True	False
True	True	True	True

Attention

☢ Pour être sûr d'avoir un résultat booléen avec une expression reposant sur des valeurs transtypées, appliquez `bool()` sur l'expression.

2.5 Les types de données flottants

Remarque

✓ La notion mathématique de *réel* est une notion idéale. Ce graal est impossible à atteindre en informatique. On utilise une représentation interne (normalisée) permettant de déplacer la virgule grâce à une valeur d'exposant variable. On nommera ces nombres des *flottants*.

2.5.1 Le type `float`

— Un `float` est noté avec un point décimal (jamais avec une virgule) ou en notation exponentielle avec un « e » symbolisant le « 10 puissance » suivi des chiffres de l'exposant. Par exemple :

```
2.718
.02
-1.6e-19
6.023e23
```

— les flottants supportent les mêmes opérations que les entiers ;
— ils ont une précision finie limitée ;
— l'import du module `math` autorise toutes les opérations mathématiques usuelles. Par exemple :

```
>>> import math

>>> print(math.sin(math.pi/4))
0.7071067811865475

>>>> print(math.degrees(math.pi))
180.0

>>>> print(math.factorial(9))
362880

>>> print(math.log(1024, 2))
10.0
```

2.5.2 Le type `complex`

— Les complexes sont écrits en notation cartésienne formée de deux flottants ;
— la partie imaginaire est suffixée par `j` :

```
>>> print(1j)
1j

>>> print((2+3j) + (4-7j))
(6-4j)

>>> print((9+5j).real)
9.0

>>> print((9+5j).imag)
5.0

>>> print((abs(3+4j))) # module
5.0
```

— Un module mathématique spécifique (`cmath`) leur est réservé :

```
>>> import cmath

>>> print(cmath.phase(-1 + 0j))
3.141592653589793

>>> print(cmath.polar(3 + 4j))
(5.0, 0.9272952180016122)

>>> print(cmath.rect(1., cmath.pi/4))
(0.7071067811865476+0.7071067811865475j)
```

2.6 Variables et affectation

2.6.1 Les variables

Pour stocker des données, on a besoin de *variables*. En réalité, Python n'offre pas directement la notion de variable, mais plutôt celle de *référence d'objet*.

Définition

✍ Une variable est un identificateur associé à une valeur. En Python, c'est une référence d'objet.

Tant que l'objet n'est pas modifiable (entier, flottant, chaîne, etc.), il n'y a pas de différence notable entre variable et référence. On verra que la situation change dans le cas des objets modifiables…

Une variable spéciale : `_`, utilisable uniquement en mode interactif, contient le résultat de la dernière opération :

```
>>> 6000 + 4523.50 + 135.12
10658.62

>>> _ - 8192.14
2466.4800000000014
```

2.6.2 L'affectation (ou assignation)

Syntaxe

✎ On affecte une variable par une valeur en utilisant le signe égale (=).
Attention : l'affectation *n'a rien à voir* avec l'égalité en math !

```
a = 2
b = 'John Deuf'
```

Remarque

✓ Puisqu'une variable est une référence, pour ne pas confondre *affectation* et *égalité mathématique*, `b = 'John Deuf'` pourra se dire « `b` *pointe sur* 'John Deuf' ».
La figure 2.2 illustre ce mécanisme : le cercle contient une référence et le rectangle une valeur.

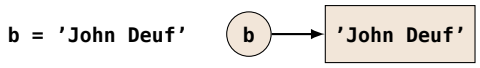


FIGURE 2.2 – Une affectation : « b pointe sur 'John Deuf' ».

La valeur d'une variable, comme son nom l'indique, peut évoluer au cours du temps. Dans une réaffectation, la valeur antérieure est perdue :

```
>>> a = 3 * 7

>>> a
21

>>> b = 2 * 2

>>> b
4

>>> a = b + 5

>>> a
9
```

Le membre de droite d'une affectation étant évalué avant de réaliser l'affectation elle-même, la variable affectée peut se trouver en partie droite et c'est sa valeur avant l'affectation qui est utilisée dans le calcul :

```
>>> a = 2

>>> a = a + 1 # incrémentation

>>> a
3

>>> a = a - 1 # décrémentation

>>> a
2
```

2.6.3 Affecter n'est pas comparer !

L'affectation a un **effet** (elle modifie l'état interne du programme en cours d'exécution), mais n'a pas de valeur (on ne peut pas l'utiliser dans une expression) :

```
>>> c = True

>>> s = (c = True) and True
  File "<stdin>", line 1
    s = (c = True) and True
           ^
SyntaxError: invalid syntax
```

La comparaison a une **valeur** (de type `bool`) utilisable dans une expression, mais n'a pas d'effet :

```
>>> c = "a"

>>> s = (c == "a") and True

>>> s
True
```

2.6.4 Les variantes de l'affectation

Outre l'affectation simple, on peut aussi utiliser les formes suivantes :

```
>>> v = 4 # affectation simple

>>> v += 2 # affectation augmentée. Idem à v = v + 2 si v est déjà référencé

>>> v
6

>>> c = d = 8 # d reçoit 8 puis c reçoit d (c et d sont des alias)

>>> c, d
(8, 8)

>>> e, f = 2.7, 5.1 # affectation parallèle d'un tuple

>>> (e, f)
(2.7, 5.1)

>>> g, h = ['G', 'H'] # affectation parallèle d'une liste

>>> [g, h]
['G', 'H']

>>> a = 3

>>> a, b = a + 2, a * 2 # toutes les expressions sont évaluées avant la première affectation

>>> a, b
(5, 6)
```

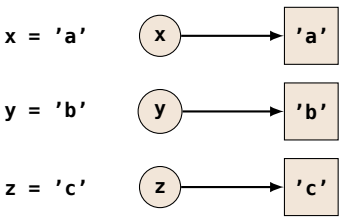
Remarque

✓ Dans une affectation, le membre de gauche pointe sur le membre de droite ce qui nécessite d'évaluer la valeur du membre de droite *avant* de l'affecter au membre de gauche. On voit dans l'affectation parallèle l'importance de cet ordre temporel.

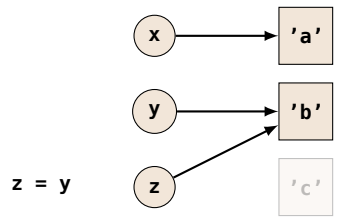
2.6.5 Représentation graphiques des affectations

Comme auparavant, dans les schémas de la figure 2.3, les cercles représentent les références alors que les rectangles représentent les valeurs.

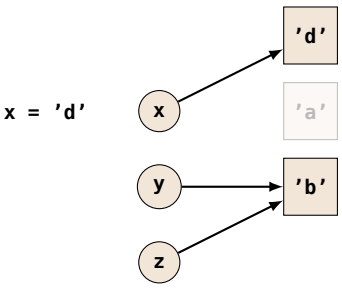
Les affectations **relient** les identificateurs aux données : si une donnée en mémoire n'est plus reliée, le ramasse-miettes (*garbage collector*) de Python la supprime automatiquement (car son nombre de références tombe à zéro) :



(a) Trois affectations



(b) La donnée `'c'` est supprimée lorsque z référence y



(c) La donnée `'a'` est supprimée lorsque x référence la nouvelle donnée `'d'`

FIGURE 2.3 – L'affectation illustrée.

> Retour *Table des matières*

2.7 Les chaînes de caractères

2.7.1 Présentation

Définition

✍ Les chaînes de caractères : le type de données non modifiable `str` représente une séquence de caractères Unicode.

Non modifiable signifie qu'une donnée, une fois créée en mémoire, ne pourra plus être changée ; toute transformation résultera en la création d'une nouvelle valeur distincte.

Trois syntaxes de chaîne sont disponibles. Remarquez que l'on peut aussi utiliser le ' à la place de ", ce qui permet d'inclure une notation dans l'autre :

```
>>> syntaxe1 = "Première forme avec un retour à la ligne \n"

>>> print(syntaxe1)
Première forme avec un retour à la ligne

>>> syntaxe2 = r"Deuxième forme. Chaîne brute (raw) sans retour\n à la ligne"

>>> print(syntaxe2)
Deuxième forme. Chaîne brute (raw) sans retour\n à la ligne

>>> syntaxe3 = """
Troisième forme
multi-lignes
  très utile pour
    la documentation
"""

>>> print(syntaxe3)

Troisième forme
multi-lignes
  très utile pour
    la documentation

>>> guillemets = "L'eau vive"

>>> guillemets
"L'eau vive"

>>> apostrophes = 'Il a dit "gère !"'

>>> apostrophes
'Il a dit "gère !"'
```

2.7.2 Opérations

— Longueur :

```
>>> s = "abcde"

>>> len(s)
5
```

— concaténation :

```
>>> s1 = "abc"

>>> s2 = "defg"

>>> s3 = s1 + s2

>>> s3
'abcdefg'
```

— répétition :

```
>>> s4 = "Fi! "

>>> s5 = s4 * 3

>>> s5
'Fi! Fi! Fi! '
```

2.7.3 Fonctions *vs* méthodes

On peut agir sur une chaîne en utilisant des *fonction* (notion procédurale) communes à tous les types séquences ou conteneurs, ou bien des *méthodes* (notion objet) spécifiques aux chaînes.

— Exemple d'utilisation de la fonction `len()` :

```
>>> lng = len("abc")

>>> lng
3
```

— Exemple d'utilisation de la méthode `upper()`. Remarquez la différence de syntaxe : les méthodes utilisent la notation pointée :

```
>>> "abracadabra".upper()
"ABRACADABRA"
```

2.7.4 Méthodes de test de l'état d'une chaîne

Les méthodes suivantes sont à valeur booléenne, c'est-à-dire qu'elles retournent la valeur `True` ou `False`.

Remarque

✓ La notation entre crochets *`[xxx]`* indique un élément optionnel que l'on peut donc omettre lors de l'utilisation de la méthode.

La chaîne `s = "cHAise basSe"` nous servira de test pour toutes les méthodes de cette section.

— `isupper()` et `islower()` : retournent `True` si la chaîne ne contient respectivement que des majuscules/minuscules :

```
>>> s.isupper()
False
```

— `istitle()` : retourne `True` si seule la première lettre de chaque mot de la chaîne est en majuscule :

```
>>> s.istitle()
False
```

— `isalnum()`, `isalpha()`, `isdigit()` et `isspace()` : retournent `True` si la chaîne ne contient respectivement que des caractères alphanumériques, alphabétiques, numériques ou des espaces :

```
>>> s.isalpha()
True

>>> s.isdigit()
False
```

— `startswith(prefix[, start[, stop]])` et `endswith(suffix[, start[, stop]])` : testent si la sous-chaîne définie par `start` et `stop` commence respectivement par `prefix` ou finit par `suffix` :

```
>>> s.startswith('cH')
True

>>> s.endswith('aSSe')
False
```

2.7.5 Méthodes retournant une nouvelle chaîne

— `lower()`, `upper()`, `capitalize()` et `swapcase()` : retournent respectivement une chaîne en minuscules, en majuscules, en minuscules commençant par une majuscule, ou en casse inversée :

```
>>> s.lower()
chaise basse

>>> s.upper()
CHAISE BASSE

>>> s.capitalize()
Chaise basse

>>> s.swapcase()
ChaISE BASsE
```

— `expandtabs([tabsize])` : remplace les tabulations par `tabsize` espaces (8 par défaut).

— `center(width[, fillchar])`, `ljust(width[, fillchar])` et `rjust(width[, fillchar])` : retournent respectivement une chaîne centrée, justifiée à gauche ou à droite, complétée par le caractère `fillchar` (ou par l'espace par défaut) :

```
>>> s.center(20, '-')
----cHAise basSe----

>>> s.rjust(20, '@')
@@@@@@@@cHAise basSe
```

— `zfill(width)` : complète `ch` à gauche avec des `0` jusqu'à une longueur maximale de `width` :

```
>>> s.zfill(20)
00000000cHAise basSe
```

— `strip([chars])`, `lstrip([chars])` et `rstrip([chars])` : suppriment toutes les combinaisons de `chars` (ou l'espace par défaut) respectivement au début et en fin, au début, ou en fin d'une chaîne :

```
>>> s.strip('ce')
HAise basS
```

— `find(sub[, start[, stop]])` : renvoie l'index de la chaîne `sub` dans la sous-chaîne `start` à `stop`, sinon renvoie `-1`. `rfind()` effectue le même travail en commençant par la fin. `index()` et `rindex()` font de même, mais produisent une erreur (*exception*) si la chaîne n'est pas trouvée :

```
>>> s.find('se b')
4
```

— `replace(old, new[, count])` : remplace `count` instances (toutes par défaut) de `old` par `new` :

```
>>> s.replace('HA', 'ha')
chaise basSe
```

— `split(seps[, maxsplit])` : découpe la chaîne en `maxsplit` morceaux (tous par défaut). `rsplit()` effectue la même chose en commençant par la fin et `striplines()` effectue ce travail avec les caractères de fin de ligne :

```
>>> s.split()
['cHAise', 'basSe']
```

— `join(seq)` : concatène les chaînes du conteneur `seq` en intercalant entre chaque élément la chaîne sur laquelle la méthode est appliquée :

```
>>> "**".join(['cHAise', 'basSe'])
cHAise**basSe
```

2.7.6 Indexation simple

Pour indexer une chaîne, on utilise l'opérateur `[]` dans lequel l'**index**, un entier signé qui commence à `0` indique la position d'un caractère :

```
>>> s = "Rayons X" # len(s) ==> 8

>>> s[0]
'R'

>>> s[2]
'y'

>>> s[-1]
'X'

>>> s[-3]
'n'
```

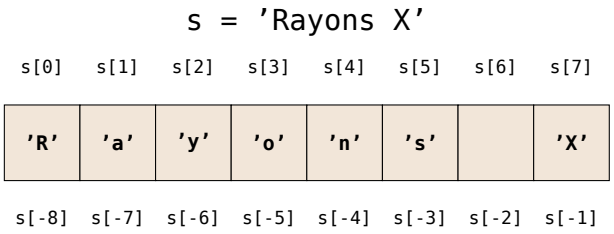


FIGURE 2.4 – L'indexation d'une chaîne

2.7.7 Extraction de sous-chaînes

Définition

✍ Extraction de sous-chaînes. L'opérateur `[]` avec 2 ou 3 index séparés par le caractère `:` permet d'extraire des sous-chaînes (ou tranches) d'une chaîne.

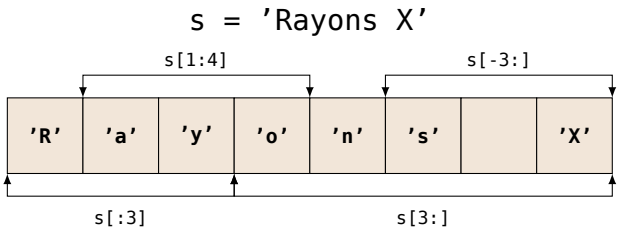


FIGURE 2.5 – Extraction de sous-chaînes

Par exemple :

```
>>> s = "Rayons X" # len(s) ==> 8

>>> s[1:4] # de l'index 1 compris à 4 non compris
'ayo'

>>> s[-2:] # de l'index -2 compris à la fin
' X'

>>> s[:3] # du début à l'index 3 non compris
'Ray'

>>> s[3:] # de l'index 3 compris à la fin
'on X'

>>> s[::2] # du début à la fin, de 2 en 2
```

```
'RynX'

>>> s[::-1] # de la fin au début (retournement)
'X noyaR'
```

2.8 Les données binaires

Les types binaires

Python 3 propose deux types de données binaires : `bytes` (non modifiable) et `bytearray` (modifiable).

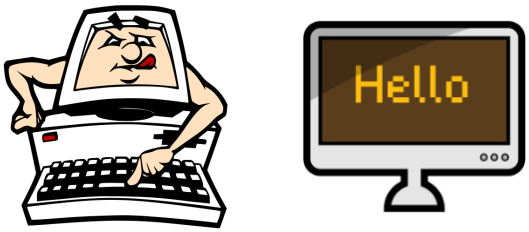
```
>>> mot = "Animal"

>>> type(mot)
<class 'str'>

>>> b_mot = b"Animal"

>>> type(b_mot)
<class 'bytes'>

>>> bmot = bytearray(b_mot)

>>> type(bmot)
<class 'bytearray'>
```

Une donnée binaire contient une suite de zéro ou plusieurs octets, c'est-à-dire d'entiers non signés sur 8 bits (compris dans l'intervalle [0…255]). Ces types « *à la C* » sont bien adaptés pour stocker de grandes quantités de données. De plus Python fournit des moyens de manipulation efficaces de ces types.

Les deux types sont assez semblables au type `str` et possèdent la plupart de ses méthodes. Le type modifiable `bytearray` possède des méthodes communes au type `list`.

2.9 Les entrées-sorties

L'utilisateur a besoin d'interagir avec le programme (☞ Fig. 2.6). En mode « console » (on abordera les interfaces graphiques ultérieurement), on doit pouvoir *saisir* ou *entrer* des informations, ce qui est généralement fait depuis une **lecture** au clavier. Inversement, on doit pouvoir *afficher* ou *sortir* des informations, ce qui correspond généralement à une **écriture** sur l'écran.



(a) Entrée (b) Sortie

FIGURE 2.6 – Les entrées-sorties.

2.9.1 Les entrées

Il s'agit de réaliser une *saisie* au clavier : la fonction standard `input()` interrompt le programme, affiche une éventuelle invite à l'écran et attend que l'utilisateur entre une donnée au clavier (affichée à l'écran) et la valide par *Entrée*.

La fonction `input()` effectue toujours une saisie en *mode texte* (la valeur retournée est une chaîne) dont on peut ensuite changer le type (on dit aussi « transtyper » ou *cast* en anglais) :

```
>>> f1 = input("Entrez un flottant : ")
Entrez un flottant : 12.345

>>> type(f1)
<class 'str'>

>>> f2 = float(input("Entrez un autre flottant : "))
Entrez un autre flottant : 12.345

>>> type(f2)
<class 'float'>
```

On rappelle que Python est un langage *dynamique* (les variables peuvent changer le type au gré des affectations), mais également *fortement typé* (contrôle de la cohérence des types) :

```
>>> i = input("Entrez un entier : ")
Entrez un entier : 3

>>> i
'3'

>>> iplus = int(input("Entrez un entier : ")) + 1
Entrez un entier : 3

>>> iplus
4

>>> ibug = input("Entrez un entier : ") + 1
Entrez un entier : 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

2.9.2 Les sorties

En mode « calculatrice », Python *lit-évalue-affiche* (☞ Fig. 2.1, p. 7), mais la fonction `print()` reste indispensable aux affichages dans les scripts. Elle se charge d'afficher la représentation textuelle des informations qui lui sont données en paramètre, en plaçant un blanc séparateur entre deux informations, et en faisant un retour à la ligne à la fin de l'affichage (le séparateur et la fin de ligne peuvent être modifiés) :

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

>>> a, b = 2, 5

>>> print(a, b)
2 5

>>> print("Somme :", a + b)
Somme : 7

>>> print(a - b, "est la différence")
-3 est la différence

>>> print("Le produit de", a, "par", b, "vaut :", a * b)
Le produit de 2 par 5 vaut : 10

>>> print("On a <", 2**32, a*b, "> cas !", sep="~~~")
On a <~~~4294967296~~~10~~~> cas !

>>> # pour afficher autre chose qu'un espace en fin de ligne :

>>> print(a, end="@")
2@
>>> print()

>>>
```


> Retour *Table des matières*

2.9.3 Les séquences d'échappement

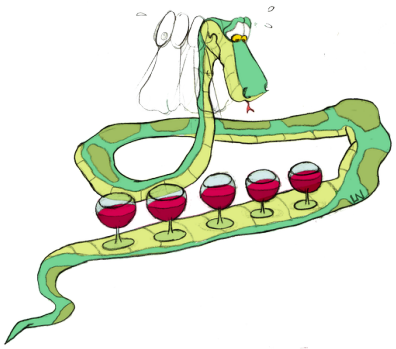
À l'intérieur d'une chaîne, le caractère antislash (\) permet de donner une signification spéciale à certaines séquences de caractères :

Séquence	Signification
`\saut_ligne`	saut de ligne ignoré (en fin de ligne)
`\\`	affiche un antislash
`\'`	apostrophe
`\"`	guillemet
`\a`	sonnerie (*bip*)
`\b`	retour arrière
`\f`	saut de page
`\n`	saut de ligne
`\r`	retour en début de ligne
`\t`	tabulation horizontale
`\v`	tabulation verticale
`\N{nom}`	caractère sous forme de code Unicode nommé
`\uhhhh`	caractère sous forme de code Unicode 16 bits
`\Uhhhhhhhh`	caractère sous forme de code Unicode 32 bits
`\ooo`	caractère sous forme de code octal
`\xhh`	caractère sous forme de code hexadécimal

```
>>> print("\N{pound sign} \u00A3 \U000000A3")
£ £ £

>>> print("d \144 \x64")
d d d

>>> print(r"d \144 \x64") # la notation r"..." désactive la signification spéciale du caractère "\"
d \144 \x64
```

Chapitre 3

Contrôle du flux d'instructions



Un script Python est formé d'une suite d'instructions exécutées en séquence de haut en bas : le flux normal d'instructions.

Chaque ligne d'instructions est formée d'une ou plusieurs lignes physiques qui peuvent être continuées par un antislash `\` ou un caractère ouvrant `[({` pas encore fermé.

Ce flux peut être modifiée pour *choisir* ou *répéter* des portions de code en utilisant des « instructions composées ».

3.1 Instructions composées

Pour identifier les instructions composées, Python utilise la notion d'*indentation significative*. Cette syntaxe, légère et visuelle, met en lumière un bloc d'instructions et permet d'améliorer grandement la présentation et donc la lisibilité des programmes sources.

Syntaxe

✎ Une instruction composée se compose :
- d'une ligne d'en-tête terminée par **deux-points** ;
- d'un bloc d'instructions indenté **par rapport à la ligne d'en-tête**. On utilise habituellement quatre espaces par indentation et on ne mélange pas les tabulations et les espaces.

Attention

☢ Toutes les instructions au même niveau d'indentation appartiennent au même bloc.

Exemple d'instruction composée simple :

```
# Exemples
# (les espaces sont indiqués par un caractère spécifique) :

ph = 6.0

if ph < 7.0:
    print("C'est un acide.")

if ph > 7.0:
    print("C'est une base.")

if ph == 7.0:
    print("La solution est neutre.")
```

Exemple d'instruction composée imbriquée :

```
n␣=␣3

if␣n␣<=␣0:
␣␣␣␣print('"n"␣est␣négatif␣ou␣nul')
else:
␣␣␣␣print('"n"␣est␣positif')
␣␣␣␣if␣n␣>␣2:
␣␣␣␣␣␣␣␣print('"n"␣est␣supérieur␣à␣2')
```

3.2 Choisir

3.2.1 Choisir : `if - [elif] - [else]`

Contrôler une alternative :

```
>>> x = 5

>>> if x < 0:
... print("x est négatif")
... elif x % 2 != 0:
... print("x est positif et impair")
... print ("ce qui est bien aussi !")
... else:
... print("x n'est pas négatif et est pair")
...

x est positif et impair
ce qui est bien aussi !
```

Test d'une valeur booléenne :

```
>>> x = 8

>>> estPair = (x % 2 == 0)

>>> estPair
True

>>> if estPair: # mieux que "if estPair == True:"
... print('La condition est vraie')
...
La condition est vraie
```

3.2.2 Syntaxe compacte d'une alternative

Pour trouver, par exemple, le minimum de deux nombres, on peut utiliser l'opérateur *ternaire* :

```
>>> x = 4

>>> y = 3

>>> if x < y: # écriture classique
... plus_petit = x
... else:
... plus_petit = y
...

>>> print("Plus petit : ", plus_petit)
Plus petit : 3

>>> plus_petit = x if x < y else y # utilisation de l'opérateur ternaire

>>> print("Plus petit : ", plus_petit)
Plus petit : 3
```

Remarque

✓ L'opérateur ternaire est une *expression* qui fournit une valeur que l'on peut utiliser dans une affectation ou un calcul.

3.3 Boucles

Notions de conteneur et d'itérable

De façon générale, nous parlerons de *conteneur* pour désigner un type de données permettant de stocker un ensemble d'autres données, en ayant ou non, suivant les types, une notion d'ordre entre ces données.

Nous parlerons aussi d'*itérable* pour désigner un conteneur que l'on peut parcourir élément par élément.

Pour parcourir ces conteneurs, nous nous servirons parfois de l'instruction `range()` qui fournit un moyen commode pour générer une liste de valeurs.

Par exemple :

```
>>> uneListe = list(range(6))

>>> uneListe
[0, 1, 2, 3, 4, 5]
```

Ces notions seront étudiées plus en détail au chapitre 4, p. 29.

Python propose deux sortes de boucles.

3.3.1 Répéter : `while`

Répéter une portion de code tant qu'une expression booléenne est vraie :

```
>>> x, cpt = 257, 0

>>> sauve = x

>>> while x > 1:
... x = x // 2 # division avec troncature
... cpt = cpt + 1 # incrémentation
...

>>> print("Approximation de log2 de", sauve, ":", cpt)
Approximation de log2 de 257 : 8
```

Utilisation classique : la *saisie filtrée* d'une valeur numérique (on doit *préciser le type*, car on se rappelle que `input()` saisit une chaîne) :

```
n = int(input('Entrez un entier [1 .. 10] : '))
while not(1 <= n <= 10):
   n = int(input('Entrez un entier [1 .. 10], S.V.P. : '))
```

3.3.2 Parcourir : `for`

Parcourir un itérable :

```
>>> for lettre in "ciao":
... print(lettre)
...
c
i
a
o

>>> for x in [2, 'a', 3.14]:
... print(x)
...
2
a
3.14
```

```
>>> for i in range(5):
... print(i)
...
0
1
2
3
4

>>> nb_voyelles = 0

>>> for lettre in "Python est un langage tres sympa":
... if lettre in "aeiouy":
... nb_voyelles = nb_voyelles + 1
...

>>> nb_voyelles
10
```

3.4 Ruptures de séquences

3.4.1 interrompre une boucle : `break`

Sort immédiatement de la boucle `for` ou `while` en cours contenant l'instruction :

```
>>> for x in range(1, 11):
... if x == 5:
... break
... print(x, end=" ")
...
1 2 3 4

>>> print("Boucle interrompue pour x =", x)
Boucle interrompue pour x = 5
```

3.4.2 Court-circuiter une boucle : `continue`

Passe immédiatement à l'itération suivante de la boucle `for` ou `while` en cours contenant l'instruction ; reprend à la ligne de l'en-tête de la boucle :

```
>>> for x in range(1, 11):
... if x == 5:
... continue
... print(x, end=" ")
...
1 2 3 4 6 7 8 9 10

>>> # la boucle a sauté la valeur 5
```

3.4.3 Utilisation avancée des boucles

La syntaxe complète des boucles autorise des utilisations plus rares.

`while - else`

Les boucles `while` et `for` peuvent posséder une clause `else` qui ne s'exécute que **si la boucle se termine normalement**, c'est-à-dire sans interruption :

```
y = int(input("Entrez un entier positif : "))
while not(y > 0):
   y = int(input('Entrez un entier positif, S.V.P. : '))

x = y // 2
while x > 1:
   if y % x == 0:
```

```
      print(x, "a pour facteur", y)
      break # voici l'interruption !
    x -= 1
else:
    print(y, "est premier.")
```

for - else

Un exemple avec le parcours d'une liste :

```
>>> entiers = [2, 11, 8, 7, 5]

>>> cible = int(input("Entrez un entier : "))
Entrez un entier : 7

>>> for entier in entiers:
... if cible == entier:
... print(cible, 'est dans la séquence', entiers)
... break # voici l'interruption !
... else:
... print(cible, "n'est pas dans la séquence", entiers)
...
7 est dans la séquence [2, 11, 8, 7, 5]

>>> cible = int(input("Entrez un entier : "))
Entrez un entier : 6

>>> for entier in entiers:
... if cible == entier:
... print(cible, 'est dans la séquence', entiers)
... break # voici l'interruption !
... else:
... print(cible, "est absent de la séquence", entiers)
...
6 est absent de la séquence [2, 11, 8, 7, 5]
```

3.4.4 Traitement des erreurs : les exceptions

Afin de rendre les applications plus robustes, il est nécessaire de gérer les erreurs d'exécution des parties sensibles du code.

Lorsqu'une erreur se produit, elle traverse toutes les couches de code comme une bulle d'air remonte à la surface de l'eau. Si elle atteint la surface sans être interceptée par le mécanisme des exceptions, le programme s'arrête et l'erreur est affichée. Le *traceback* (message complet d'erreur affiché) précise l'ensemble des couches traversées.

Définition

✍ Gérer une exception permet d'intercepter une erreur pour éviter un arrêt du programme.

Une exceptions sépare d'un côté la séquence d'instructions à exécuter lorsque tout se passe bien et, d'un autre côté, une ou plusieurs séquences d'instructions à exécuter en cas d'erreur.

Lorsqu'une erreur survient, un *objet exception* est passé au mécanisme de propagation des exceptions, et l'exécution est transférée à la séquence de traitement *ad hoc*.

Le mécanisme s'effectue donc en deux phases :
— la *levée* d'exception lors de la détection d'erreur ;
— le *traitement* approprié.

Syntaxe

✎ La séquence normale d'instructions est placée dans un bloc **`try`**.
Si une erreur est détectée (levée d'exception), elle est traitée dans le bloc **`except`** approprié (le gestionnaire d'exception).

```
from math import sin

for x in range(-4, 5): # -4, -3, -2, -1, 0, 1, 2, 3, 4
   try:
      print('{:.3f}'.format(sin(x)/x), end=" ")
   except ZeroDivisionError: # toujours fournir un type d'exception
      print(1.0, end=" ") # gère l'exception en 0
# -0.189 0.047 0.455 0.841 1.0 0.841 0.455 0.047 -0.189
```

Toutes les exceptions levées par Python appartiennent à un ensemble d'exceptions nommé `Exception`. Cette famille offre une vingtaine d'exceptions standard [1]. Normalement, toutes les exceptions doivent donc être *typées* (munies d'un nom de type d'exception) pour éviter des erreurs silencieuses.

Syntaxe complète d'une exception :

```
try:
   ... # séquence normale d'exécution
except <exception_1> as e1:
   ... # traitement de l'exception 1
except <exception_2> as e2:
   ... # traitement de l'exception 2
...
else:
   ... # clause exécutée en l'absence d'erreur
finally:
   ... # clause toujours exécutée
```

L'instruction **`raise`** permet de lever *volontairement* une exception. On peut trouver l'instruction à tout endroit du code, pas seulement dans un bloc `try` :

```
x = 2
if not(0 <= x <= 1):
   raise ValueError("x n'est pas dans [0 .. 1]")
```

Remarque

✓ `raise`, sans valeur, dans un bloc `except`, permet de ne pas bloquer une exception et de la propager. On peut l'utiliser pour redéclancher une erreur d'un bloc `except` non typé.

1. Citons quelques exemplaires : `AritmeticError`, `ZeroDivisionError`, `IndexError`, `KeyError`, `AttributeError`, `IOError`, `ImportError`, `NameError`, `SyntaxError`, `TypeError`...

> Retour *Table des matières*

Chapitre 4

Conteneurs standard

Le chapitre 2 a présenté les types de données simples, mais Python offre beaucoup plus : les conteneurs.

De façon générale, un conteneur est un objet composite destiné à contenir d'autres objets. Ce chapitre détaille les séquences, les tableaux associatifs, les ensembles et les fichiers textuels.

4.1 Séquences

Définition

✍ Une séquence est un conteneur ordonné d'éléments indexés par des entiers indiquant leur position dans le conteneur.

Python dispose de trois types prédéfinis de séquences :
- les chaînes (vues précédemment) ;
- les listes ;
- les tuples [1].

4.2 Listes

4.2.1 Définition, syntaxe et exemples

Définition

✍ Une liste est une collection ordonnée et modifiable d'éléments éventuellement hétérogènes.

Syntaxe

✎ Éléments séparés par des virgules, et entourés de crochets.

1. « tuple » n'est pas vraiment un anglicisme, mais plutôt un néologisme informatique.

Exemples simples de listes :

```
couleurs = ['trèfle', 'carreau', 'coeur', 'pique']
print(couleurs) # ['trèfle', 'carreau', 'coeur', 'pique']
couleurs[1] = 14
print(couleurs) # ['trèfle', 14, 'coeur', 'pique']
list1 = ['a', 'b']
list2 = [4, 2.718]
list3 = [list1, list2] # liste de listes
print(list3) # [['a', 'b'], [4, 2.718]]
```

4.2.2 Initialisations et tests d'appartenance

Utilisation de la répétition, de l'itérateur d'entiers `range()` et de l'opérateur d'appartenance (`in`) :

```
>>> truc = []

>>> machin = [0.0] * 3

>>> truc
[]

>>> machin
[0.0, 0.0, 0.0]

>>> liste_1 = list(range(4))

>>> liste_1
[0, 1, 2, 3]

>>> liste_2 = list(range(4, 8))

>>> liste_2
[4, 5, 6, 7]

>>> liste_3 = list(range(2, 9, 2))

>>> liste_3
[2, 4, 6, 8]

>>> 2 in liste_1, 8 in liste_2, 6 in liste_3
(True, False, True)
```

4.2.3 Méthodes

Quelques méthodes de modification des listes :

```
>>> nombres = [17, 38, 10, 25, 72]

>>> nombres.sort()

>>> nombres
[10, 17, 25, 38, 72]

>>> nombres.append(12)

>>> nombres.reverse()

>>> nombres
[12, 72, 38, 25, 17, 10]

>>> nombres.remove(38)

>>> nombres
[12, 72, 25, 17, 10]

>>> nombres.index(17)
3

>>> nombres[0] = 11
```

```
>>> nombres[1:3] = [14, 17, 2]

>>> nombres
[11, 14, 17, 2, 17, 10]

>>> nombres.pop()
10

>>> nombres
[11, 14, 17, 2, 17]

>>> nombres.count(17)
2

>>> nombres.extend([1, 2, 3])

>>> nombres
[11, 14, 17, 2, 17, 1, 2, 3]
```

4.2.4 Manipulation des « tranches » (ou sous-chaînes)

Syntaxe

✎ Si on veut supprimer, remplacer ou insérer *plusieurs* éléments d'une liste, il faut indiquer une tranche (cf. 2.7.7, p. 18) dans le membre de gauche d'une affectation et fournir une liste dans le membre de droite.

```
>>> mots = ['jambon', 'sel', 'miel', 'confiture', 'beurre']

>>> mots[2:4] = [] # effacement par affectation d'une liste vide

>>> mots
['jambon', 'sel', 'beurre']

>>> mots[1:3] = ['salade']

>>> mots
['jambon', 'salade']

>>> mots[1:] = ['mayonnaise', 'poulet', 'tomate']

>>> mots
['jambon', 'mayonnaise', 'poulet', 'tomate']

>>> mots[2:2] = ['miel'] # insertion en 3è position

>>> mots
['jambon', 'mayonnaise', 'miel', 'poulet', 'tomate']
```

4.2.5 Séquences de séquences

Les séquences, comme du reste les autres conteneurs, peuvent être imbriquées.
Par exemple :

```
>>> liste_1 = [1, 2, 3]

>>> listes = [liste_1, [4, 5], "abcd"]

>>> for liste in listes:
... for elem in liste:
... print(elem)
... print()
...
1
2
3

4
5
```

```
a
b
c
d
```

4.3 Tuples

Définition

✍ Un tuple est une collection ordonnée et non modifiable d'éléments éventuellement hétérogènes.

Syntaxe

✎ Éléments séparés par des virgules, et entourés de parenthèses.

```
>>> mon_tuple = ('a', 2, [1, 3])
```

- L'indexage des tuples s'utilise comme celui des listes ;
- le parcours des tuples est plus rapide que celui des listes ;
- ils sont utiles pour définir des constantes.

Attention

☢ Comme les chaînes de caractères, les tuples ne sont pas modifiables !

```
>>> mon_tuple.append(4) # attention ne pas modifier un tuple !
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

Les opérations des objets de type séquentiel

Les types prédéfinis de séquences Python (chaîne, liste et tuple) ont en commun les opérations résumées dans le tableau suivant où `s` et `t` désignent deux séquences du même type et `i`, `j` et `k` des entiers :

l'opération	son effet
`x in s`	`True` si `s` contient `x`, `False` sinon
`x not in s`	`True` si `s` ne contient pas `x`, `False` sinon
`s + t`	concaténation de `s` et `t`
`s * n`, `n * s`	`n` copies (superficielles) concaténées de `s`
`s[i]`	i^e élément de `s` (à partir de 0)
`s[i:j]`	tranche de `s` de `i` (inclus) à `j` (exclu)
`s[i:j:k]`	tranche de `s` de `i` à `j` avec un pas de `k`
`len(s)`	longueur de `s`
`max(s)`, `min(s)`	plus grand, plus petit élément de `s`
`s.index(i)`	indice de la 1^{re} occurrence de `i` dans `s`
`s.count(i)`	nombre d'occurrences de `i` dans `s`

4.4 Retour sur les références

Nous avons déjà vu que l'opération d'affectation, apparemment innocente, est une réelle difficulté de Python.

```
i = 1
msg = "Quoi de neuf ?"
e = 2.718
```

Dans l'exemple ci-dessus, les affectations réalisent plusieurs opérations :
- création en mémoire d'un objet du type approprié (membre de droite) ;
- stockage de la donnée dans l'objet créé ;
- création d'un nom de variable (membre de gauche) ;
- association de ce nom de variable avec l'objet contenant la valeur.

Une conséquence de ce mécanisme est que, si un objet modifiable est affecté à plusieurs variables, tout changement de l'objet via une variable sera visible sur tous les autres :

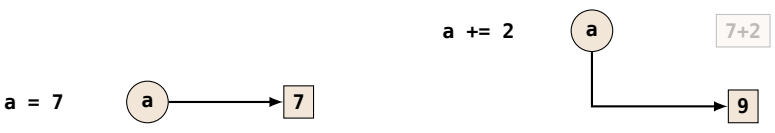
```
fable = ["Je", "plie", "mais", "ne", "romps", "point"]
phrase = fable

phrase[4] = "casse"

print(fable) # ['Je', 'plie', 'mais', 'ne', 'casse', 'point']
```

Si on veut pouvoir effectuer des modifications séparées, il faut affecter l'autre variable par une copie distincte de l'objet, soit en créant une tranche complète des séquences dans les cas simples, soit en utilisant le module `copy` dans les cas les plus généraux (autres conteneurs). Dans les rares occasions où l'on veut aussi que chaque élément et attribut de l'objet soit copié séparément et de façon récursive, on emploie la fonction `copy.deepcopy()` :

```
>>> import copy

>>> a = [1, 2, 3]

>>> b = a # une référence

>>> b.append(4)

>>> a
[1, 2, 3, 4]

>>> c = a[:] # une copie simple

>>> c.append(5)

>>> c
[1, 2, 3, 4, 5]

>>> d = copy.copy(a) # une copie de "surface"

>>> d.append(6)

>>> d
[1, 2, 3, 4, 6]

>>> a
[1, 2, 3, 4]

>>> d1 = {'a' : [1, 2], 'b' : [3, 4]}

>>> d2 = {'c' : (1, 2, 3), 'c' : (4, 5, 6)}

>>> liste_de_dicos = [d1, d2]

>>> nouvelle_liste_de_dicos = copy.deepcopy(liste_de_dicos) # copie "profonde" (ou "récursive")

>>> nouvelle_liste_de_dicos
[{'a': [1, 2], 'b': [3, 4]}, {'c': (4, 5, 6)}]
```

4.4.1 Complément graphique sur l'assignation

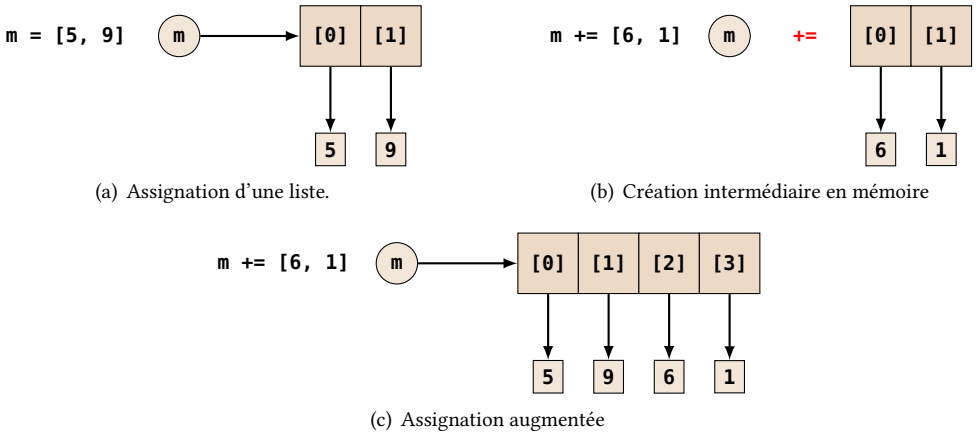
— Assignation augmentée d'un objet **non modifiable** (cas d'un entier : ☞ Fig. 4.1). On a représenté l'étape de l'addition intermédiaire.



(a) Assignation d'un entier (b) Assignation augmentée

FIGURE 4.1 – Assignation augmentée d'un objet non modifiable.

— assignation augmentée d'un objet **modifiable** (cas d'une liste : ☞ Fig. 4.2). On a représenté l'étape de la création de la liste intermédiaire.



(a) Assignation d'une liste.

(b) Création intermédiaire en mémoire

(c) Assignation augmentée

FIGURE 4.2 – Assignation augmentée d'un objet modifiable.

4.5 Tableaux associatifs

Définition

✍ Un tableau associatif est un type de données permettant de stocker des couples `(cle : valeur)`, avec un accès très rapide à la valeur à partir de la clé, la clé ne pouvant être présente qu'une seule fois dans le tableau.

Il possède les caractéristiques suivantes :

— l'opérateur d'appartenance d'une clé (`in`) ;
— la fonction taille (`len()`) donnant le nombre de couples stockés ;
— il est *itérable* (on peut le parcourir), mais *n'est pas ordonné.*

Python propose le type standard `dict`.

4.5.1 Dictionnaires (`dict`)

Syntaxe

✎ Collection de couples `cle : valeur` entourée d'accolades.

Les dictionnaires constituent un type composite, mais ils n'appartiennent pas aux séquences.

Les dictionnaires sont *modifiables* mais *non ordonnés* : les couples enregistrés n'occupent pas un ordre immuable, leur emplacement est géré par un algorithme spécifique (algorithme de *hash*). Le caractère non ordonné des dictionnaires est le prix à payer pour leur rapidité !

Une *clé* pourra être alphabétique, numérique… en fait tout type *hachable* (donc liste et dictionnaire exclus). Les *valeurs* pourront être de tout type sans exclusion.

Exemples de création

```
>>> d1 = {} # dictionnaire vide. Autre notation : d1 = dict()

>>> d1["nom"] = 3

>>> d1["taille"] = 176

>>> d1
{'nom': 3, 'taille': 176}

>>> d2 = {"nom": 3, "taille": 176} # définition en extension

>>> d2
{'nom': 3, 'taille': 176}

>>> d3 = {x: x**2 for x in (2, 4, 6)} # définition en compréhension

>>> d3
{2: 4, 4: 16, 6: 36}

>>> d4 = dict(nom=3, taille=176) # utilisation de paramètres nommés

>>> d4
{'taille': 176, 'nom': 3}

>>> d5 = dict([("nom", 3), ("taille", 176)]) # utilisation d'une liste de couples clés/valeurs

>>> d5
{'nom': 3, 'taille': 176}
```

Méthodes

Quelques méthodes applicables aux dictionnaires :

```
>>> tel = {'jack': 4098, 'sape': 4139}

>>> tel['guido'] = 4127

>>> tel
{'sape': 4139, 'jack': 4098, 'guido': 4127}

>>> tel['jack']
4098

>>> del tel['sape']

>>> tel['irv'] = 4127

>>> tel
{'jack': 4098, 'irv': 4127, 'guido': 4127}

>>> tel.keys()
['jack', 'irv', 'guido']

>>> sorted(tel.keys())
['guido', 'irv', 'jack']

>>> sorted(tel.values())
[4098, 4127, 4127]

>>> 'guido' in tel, 'jack' not in tel
(True, False)
```

4.6 Ensembles (set)

Définition

✍ Un ensemble est une collection itérable non ordonnée d'éléments *hachables* uniques.

Donc un `set` est la transposition informatique de la notion d'ensemble mathématique.

En Python, il existe deux types d'ensembles, les ensembles modifiables : `set` et les ensembles non modifiables : `frozenset`. On retrouve ici les mêmes différences qu'entre les listes et les tuples.

```
>>> X = set('spam')

>>> Y = set('pass')

>>> X
{'s', 'p', 'm', 'a'}

>>> Y # pas de duplication : qu'un seul 's'
{'s', 'p', 'a'}

>>> 'p' in X
True

>>> 'm' in Y
False

>>> X - Y # ensemble des éléments de X qui ne sont pas dans Y
{'m'}

>>> Y - X # ensemble des éléments de Y qui ne sont pas dans X
set()

>>> X ^ Y # ensemble des éléments qui sont soit dans X soit dans Y
{'m'}

>>> X | Y # union
{'s', 'p', 'm', 'a'}

>>> X & Y # intersection
{'s', 'p', 'a'}
```

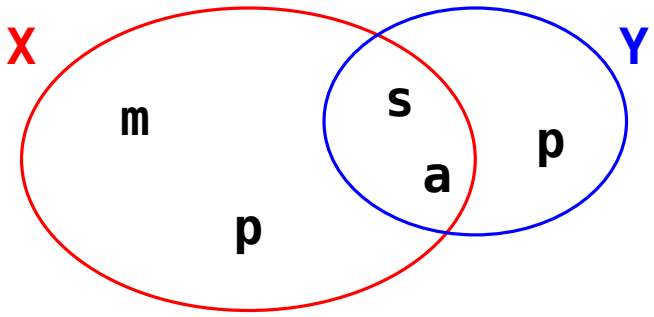


FIGURE 4.3 – Opérations sur les ensembles.

4.7 Fichiers textuels

4.7.1 Introduction

On rappelle que l'ordinateur n'exécute que les programmes présents dans sa mémoire volatile (la RAM). Pour conserver durablement des informations, il faut utiliser une mémoire permanente (un disque dur, une clé USB, un DVD) sur laquelle le système d'exploitation organise les données sous la forme de fichiers.

Comme la plupart des langages, Python utilise classiquement la notion de fichier.

Nous limiterons nos exemples aux fichiers *textuels* (lisibles par un éditeur), mais signalons que les fichiers stockés en codage *binaire* sont plus compacts et plus rapides à gérer (utiles pour les grands volumes de données).

4.7.2 Gestion des fichiers

Ouverture et fermeture des fichiers

Principaux *modes* d'ouverture des fichiers textuels [1] :

1. Remarquez que l'on précise l'*encoding*. Ceci sera justifié ultérieurement (cf. annexe C p. 116)

```
f1 = open("monFichier_1", "r", encoding='utf8') # "r" mode lecture
f2 = open("monFichier_2", "w", encoding='utf8') # "w" mode écriture
f3 = open("monFichier_3", "a", encoding='utf8') # "a" mode ajout
```

Python utilise les fichiers en mode *texte* par défaut (mode `t`). Pour les fichiers *binaires*, il faut préciser le mode `b`.

Le paramètre optionnel `encoding` assure les conversions entre les types `byte` (c'est-à-dire des tableaux d'octets), format de stockage des fichiers sur le disque, et le type `str` (qui, en Python 3, signifie toujours Unicode), manipulé lors des lectures et écritures. Il est prudent de toujours l'utiliser.

Les encodages [1] les plus fréquents sont `'utf8'` (c'est l'encodage à privilégier en Python 3), `'latin1'`, `'ascii'`...

Tant que le fichier n'est pas fermé [2], son contenu n'est pas garanti sur le disque.

Une seule méthode de fermeture :

```
f1.close()
```

Écriture séquentielle

Le fichier sur disque est considéré comme une séquence de caractères qui sont ajoutés à la suite, au fur et à mesure que l'on écrit dans le fichier.

Méthodes d'écriture :

```
f = open("truc.txt", "w", encoding='utf8')
s = 'toto\n'
f.write(s) # écrit la chaîne s dans f
l = ['a', 'b', 'c']
f.writelines(l) # écrit les chaînes de la liste l dans f
f.close()

f2 = open("truc2.txt", "w", encoding='utf8')
print("abcd", file=f2) # utilisation de l'option file
f2.close()
```

Lecture séquentielle

En lecture, la séquence de caractères qui constitue le fichier est parcourue en commençant au début du fichier et en avançant au fur et à mesure des lectures.

Méthodes de lecture d'un fichier en entier :

```
>>> f = open("truc.txt", "r", encoding='utf8')

>>> s = f.read() # lit tout le fichier --> chaîne

>>> f.close()

>>> f = open("truc.txt", "r", encoding='utf8')

>>> s = f.readlines() # lit tout le fichier --> liste de chaînes

>>> f.close()
```

Méthodes de lecture d'un fichier partiel :

```
>>> f = open("truc.txt", "r", encoding='utf8')

>>> s = f.read(3) # lit au plus n octets --> chaîne

>>> s = f.readline() # lit la ligne suivante --> chaîne

>>> f.close()

>>> # Affichage des lignes d'un fichier une à une
```

1. cf. annexe C p. 115
2. ou bien *flushé* par un appel à la méthode `flush()`.

```
>>> f = open("truc.txt", encoding='utf8') # mode "r" par défaut

>>> for ligne in f:
... print(ligne)
...

>>> f.close()
```

4.8 Travailler avec des fichiers et des répertoires

Dès que l'on manipule les répertoires on a besoin de se déplacer dans l'arborescence des fichiers, de connaître les noms de base ou l'extension de leur nom, etc.

4.8.1 Se positionner dans l'arborescence

```
>>> import os

>>> os.chdir('/home/bob/Esperanto') # changer de répertoire

>>> os.getcwd() # afficher le répertoire courant
'/home/bob/Esperanto'
```

4.8.2 Construction de noms de chemins

```
>>> import os

>>> os.path.join('home', 'bob/Esperanto', 'Baza_kurso') # concaténer les noms d'un chemin
'home/bob/Esperanto/Baza_kurso'

>>> os.path.join('/home', 'bob', 'Esperanto', 'Baza_kurso')
'/home/bob/Esperanto/Baza_kurso'

>>> os.path.expanduser('~/Esperanto') # remplace ~ par le 'home' de l'utilisateur
'/home/bob/Esperanto'

>>> os.path.join(os.path.expanduser('~'), 'Esperanto')
'/home/bob/Esperanto'
```

4.8.3 Division de noms de chemins

```
>>> import os

>>> os.path.exists('/home/bob/Esperanto/Le_Pen')
False

>>> os.path.exists('/home/bob/Esperanto/Brassens')
True

>>> os.path.isfile('/home/bob/Esperanto/Brassens') # c'est un répertoire
False

>>> os.path.isfile('/home/bob/Esperanto/Brassens/Brassens-eo.odt')
True

>>> os.path.dirname('/home/bob/Esperanto')
'/home/bob'

>>> os.path.basename('/home/bob/Esperanto')
'Esperanto'

>>> os.path.split('/home/bob/Esperanto') # retourne le tuple (dirname, basename)
('/home/bob', 'Esperanto')

>>> (shortname, extension) = os.path.splitext('gerda.pdf')

>>> shortname
'gerda'

>>> extension
'.pdf'
```

4.8.4 Gestion des fichiers d'un répertoire

```
>>> import os

>>> os.listdir('/home/bob/Esperanto/Brassens')
['Brassens-fr.pdf', 'Brassens-eo.odt', 'Brassens-fr.odt']

>>> dirname = '/home/bob/Esperanto/Einstein'

>>> os.listdir(dirname)
['Einstein.ods', 'farendajxoj.txt', 'Src', 'Einstein.odt']
```

Signalons également le module standard `shutil` qui autorise des opérations de haut niveau sur des fichiers ou des répertoires comme la copie, la suppression ou le renommage.

4.9 Itérer sur les conteneurs

Les techniques suivantes sont classiques et très utiles.

Obtenir clés et valeurs en bouclant sur un dictionnaire

```
knights = {"Gallahad": "the pure", "Robin": "the brave"}
for k, v in knights.items():
   print(k, v)
# Gallahad the pure
# Robin the brave
```

Obtenir indice et élément en bouclant sur une liste

```
>>> for i, v in enumerate(["tic", "tac", "toe"]):
... print(i, '->', v)
...
0 -> tic
1 -> tac
2 -> toe
```

Boucler sur deux séquences (ou plus) appariées

La fonction `zip()` fait allusion à la fermeture-éclair qui joint et entrelace deux rangées de dents.

```
>>> questions = ['name', 'quest', 'favorite color']

>>> answers = ['Lancelot', 'the Holy Grail', 'blue']

>>> for question, answer in zip(questions, answers):
... print('What is your', question, '? It is', answer)
...
What is your name ? It is Lancelot
What is your quest ? It is the Holy Grail
What is your favorite color ? It is blue
```

Obtenir une séquence inversée (la séquence initiale est inchangée)

```
for i in reversed(range(1, 10, 2)):
   print(i, end=" ") # 9 7 5 3 1
```

Obtenir une séquence triée à éléments uniques (la séquence initiale est inchangée)

```
basket = ["apple", "orange", "apple", "pear", "orange", "banana"]
for f in sorted(set(basket)):
   print(f, end=" ") # apple banana orange pear
```

4.10 Affichage formaté

La méthode `format()` permet de contrôler finement la création de chaînes formatées. On l'utilisera pour un affichage via `print()`, pour un enregistrement via `f.write()`, ou dans d'autres cas.

Remplacements simples :

```
print("{} {} {}".format("zéro", "un", "deux")) # zéro un deux

# formatage d'une chaîne pour usages ultérieurs
chain = "{2} {0} {1}".format("zéro", "un", "deux")

print(chain) # affichage : deux zéro un
with open("test.txt", "w", encoding="utf8") as f:
   f.write(chain) # enregistrement dans un fichier

print("Je m'appelle {}".format("Bob")) # Je m'appelle Bob
print("Je m'appelle {{{}}}".format("Bob")) # Je m'appelle {Bob}
print("{}".format("-"*10)) # ----------
```

Remplacements avec champs nommés :

```
a, b = 5, 3
print("The story of {c} and {d}".format(c=a+b, d=a-b)) # The story of 8 and 2
```

Formatages à l'aide de liste :

```
stock = ['papier', 'enveloppe', 'chemise', 'encre', 'buvard']
print("Nous avons de l'{0[3]} et du {0[0]} en stock\n".format(stock)) # Nous avons de l'encre et du papier en stock
```

Formatages à l'aide de dictionnaire :

```
print("My name is {0[name]}".format(dict(name='Fred'))) # My name is Fred

d = dict(poids = 12000, animal = 'éléphant')
print("L'{0[animal]} pèse {0[poids]} kg\n".format(d)) # L'éléphant pèse 12000 kg
```

Remplacement avec attributs nommés :

```
import math
import sys

print("math.pi = {.pi}, epsilon = {.float_info.epsilon}".format(math, sys))
# math.pi = 3.14159265359, epsilon = 2.22044604925e-16
```

Conversions textuelles, `str()` et `repr()` [1] :

```
>>> print("{0!s} {0!r}".format("texte\n"))
texte
  'texte\n'
```

Formatages numériques :

```
s = "int :{0:d}; hex: {0:x}; oct: {0:o}; bin: {0:b}".format(42)
print(s) # int :42; hex: 2a; oct: 52; bin: 101010
s = "int :{0:d}; hex: {0:#x}; oct: {0:#o}; bin: {0:#b}".format(42)
print(s) # int :42; hex: 0x2a; oct: 0o52; bin: 0b101010

n = 100
pi = 3.1415926535897931
k = -54

print("{:.4e}".format(pi)) # 3.1416e+00
print("{:g}".format(pi)) # 3.14159
print("{:.2%}".format(n/(47*pi))) # 67.73%

msg = "Résultat sur {:d} échantillons : {:.2f}".format(n, pi)
print(msg) # Résultat sur 100 échantillons : 3.14

msg = "{0.real} et {0.imag} sont les composantes du complexe {0}".format(3-5j)
print(msg) # 3.0 et -5.0 sont les composantes du complexe (3-5j)

print("{:+d} {:+d}".format(n, k)) # +100 -54 (on force l'affichage du signe)

print("{:,}".format(1234567890.123)) # 1,234,567,890.12
```

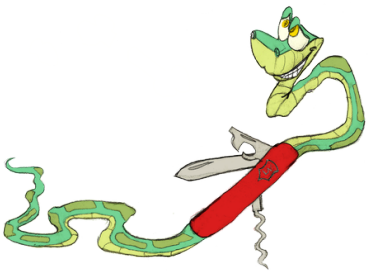
1. `str()` est un affichage orienté utilisateur alors que `repr()` est une représentation littérale.

Formatages divers :

```
>>> s = "The sword of truth"

>>> print("[{}]".format(s))
[The sword of truth]

>>> print("[{:25}]".format(s))
[The sword of truth
     ]
>>> print("[{:>25}]".format(s))
[ The sword of truth]

>>> print("[{:^25}]".format(s))
[ The sword of truth ]

>>> print("[{:-^25}]".format(s))
[---The sword of truth----]

>>> print("[{:.<25}]".format(s))
[The sword of truth.......]

>>> lng = 12

>>> print("[{}]".format(s[:lng]))
[The sword of]

>>> m = 123456789

>>> print("{:0=12}".format(m))
000123456789
```

Chapitre 5

Fonctions et espaces de noms



Les fonctions sont les éléments structurants de base de tout langage procédural.

Elles offrent différents avantages :

Évitent la répétition : on peut « factoriser » une portion de code qui se répète lors de l'exécution en séquence d'un script ;

Mettent en relief les données et les résultats : entrées et sorties de la fonction ;

Permettent la réutilisation : mécanisme de l'import ;

Décomposent une tâche complexe en tâches plus simples : conception de l'application.

Ces avantages sont illustrés sur la figure 5.1 qui utilise entre autres la notion d'*import*, mécanisme très simple qui permet de réutiliser des fichiers de fonctions, souvent appelés *modules* ou *bibliothèques*.

5.1 Définition et syntaxe

Nous avons déjà rencontré des fonctions internes à Python [1], par exemple `len()`. Intéressons-nous maintenant aux fonctions définies par l'utilisateur.

Définition

✍ Une fonction est un ensemble d'instructions regroupées sous un *nom* et s'exécutant à la demande.

On doit définir une fonction à chaque fois qu'un bloc d'instructions se trouve à plusieurs reprises dans le code ; il s'agit d'une « factorisation de code ».

Syntaxe

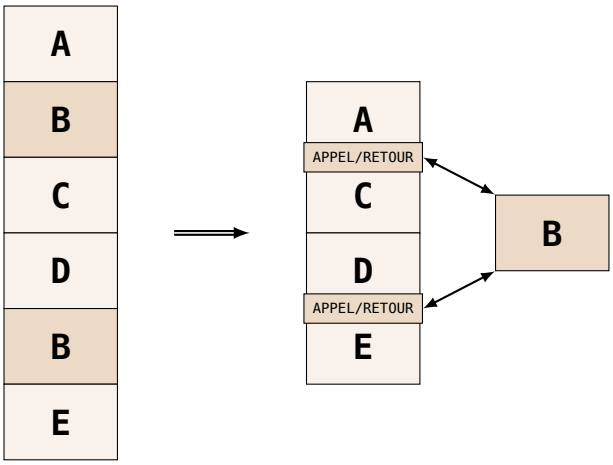
✎ La définition d'une fonction se compose :

- du mot clé `def` suivi de l'identificateur de la fonction, de parenthèses entourant les paramètres de la fonction séparés par des virgules, et du caractère « deux-points » qui termine toujours une instruction composée ;
- d'une chaîne de documentation (ou *docstring*) indentée comme le corps de la fonction ;
- du bloc d'instructions indenté par rapport à la ligne de définition, et qui constitue le corps de la fonction.

Le bloc d'instructions est obligatoire. S'il est vide, on emploie l'instruction `pass`.

La documentation, bien que facultative, est *fortement* conseillée.

1. appelées *builtin*.



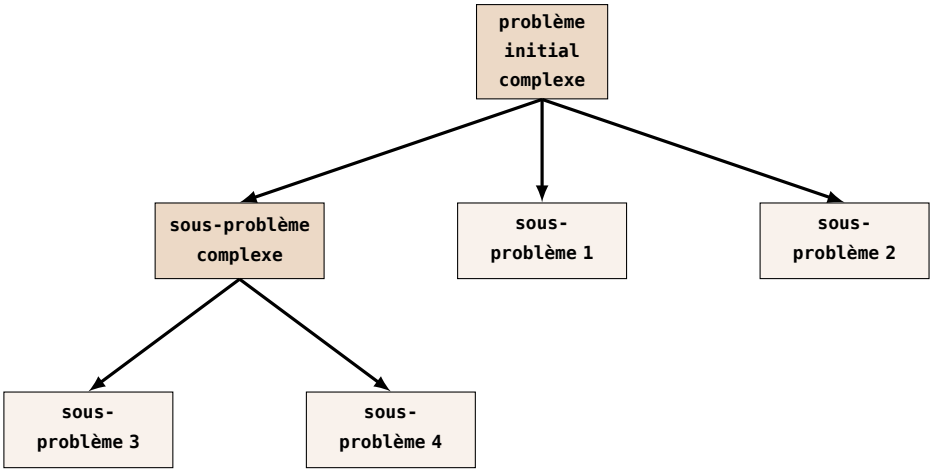
(a) Évite la duplication de code

```
# util.py
def proportion(chaine, motif):
   """Fréquence de <motif> dans <chaine>."""
   n = len(chaine)
   k = chaine.count(motif)
   return k/n
```

(b) Met en relief entrées et sorties

```
import util

...
p1 = util.proportion(une_chaine, 'le')
...
p2 = util.proportion(une_autre_chaine, 'des')
...
```

(c) L'instruction import permet la réutilisation



(d) Améliore la conception

FIGURE 5.1 – Les avantages de l'utilisation des fonctions.

```
def afficheAddMul(a, b):
    """Calcule et affiche :
      - la somme de a et b,
      - le produit de a et b.
    """
    somme = a + b
    produit = a * b
    print("La somme de", a, " et", b, " est", somme, "et le produit", produit)
```

5.2 Composition des fonctions

Comme on vient de le voir, on peut améliorer la conception d'un problème en le décomposant en fonctions plus simples. Dans d'autres circonstances, l'opération inverse sera pertinente, la composition de fonctions.

Rappel

Soit **E**, **F** et **G** trois ensembles, f une application de **E** vers **F** et g une application de **F** vers **G**.
À chaque élément x de **E** correspond par f un élément y de **F** : $y = f(x)$; à cet élément y de **F**, on peut faire correspondre par g un élément z de **G** : $z = g(y)$.
Ainsi, par ce procédé, à chaque élément x de **E** on a pu faire correspondre un élément z de **G**.
On a donc défini une application de **E** dans **G** que l'on appelle la *composée* de f et g et que l'on note $g \circ f$.
Cette opération n'est pas commutative : $g \circ f \neq f \circ g$.

Par exemple :

```
>>> def f(x):
... return 2*x + 3
...

>>> def g(x):
... return x**2
...

>>> def compose(f1, f2):
... def fonc(x):
... return f2(f1(x))
... return fonc
...

>>> print(compose(f, g)(2.0)) # g[f(x)] = (2*x + 3)**2
49.0

>>> print(compose(g, f)(2.0)) # f[g(x)] = 2*x**2 + 3
11.0
```

5.3 Passage des arguments

5.3.1 Mécanisme général

Remarque

✓ Passage par affectation : chaque paramètre de la définition de la fonction correspond, *dans l'ordre*, à un argument de l'appel. La correspondance se fait par *affectation* des arguments aux paramètres.

5.3.2 Un ou plusieurs paramètres, pas de retour

Exemple sans l'instruction `return`, ce qu'on appelle souvent une `procédure` [1]. Dans ce cas la fonction renvoie implicitement la valeur `None` :

1. Une fonction *vaut* quelque chose, une procédure *fait* quelque chose.

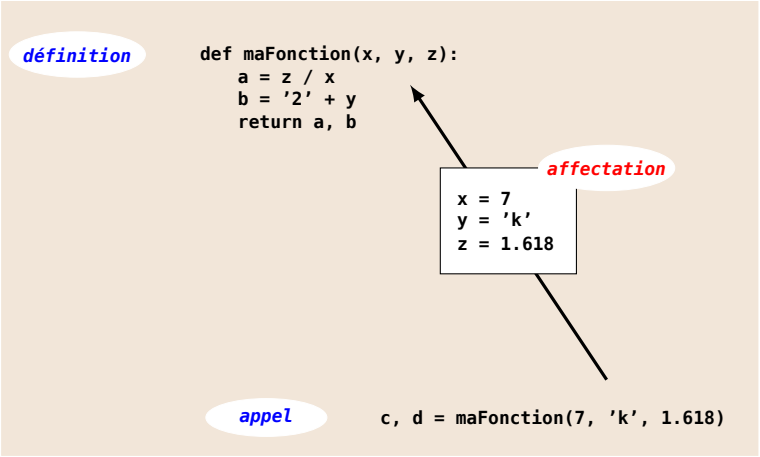


FIGURE 5.2 – Passage par affectation des arguments d'appel aux paramètres de définition.

```
def table(base, debut, fin):
   """Affiche la table de multiplication des <base> de <debut> à <fin>."""
   n = debut
   while n <= fin:
      print(n, 'x', base, '=', n * base)
      n += 1

# exemple d'appel :
table(7, 2, 8)
# 2 x 7 = 14 3 x 7 = 21 4 x 7 = 28 5 x 7 = 35 6 x 7 = 42 7 x 7 = 49 8 x 7 = 56

# autre exemple du même appel, mais en nommant les paramètres ;
table(base=7, debut=2, fin=8)
```

5.3.3 Un ou plusieurs paramètres, un ou plusieurs retours

Exemple avec utilisation d'un `return` unique :

```
from math import pi

def cube(x):
   """Retourne le cube de l'argument."""
   return x**3

def volumeSphere(r):
   """Retourne le volume d'une sphère de rayon <r>;"""
   return 4.0 * pi * cube(r) / 3.0

# Saisie du rayon et affichage du volume
rayon = float(input('Rayon : '))
print("Volume de la sphère =", volumeSphere(rayon))
```

Exemple avec utilisation d'un `return` multiple :

```
import math

def surfaceVolumeSphere(r):
   surf = 4.0 * math.pi * r**2
   vol = surf * r/3
   return surf, vol

# programme principal
rayon = float(input('Rayon : '))
s, v = surfaceVolumeSphere(rayon)
print("Sphère de surface {:g} et de volume {:g}".format(s, v))
```

5.3.4 Passage d'une fonction en paramètre

Puisqu'en Python une variable peut référencer une fonction, on peut transmettre une fonction comme paramètre :

```
>>> def f(x):
... return 2*x+1
...

>>> def g(x):
... return x//2
...

>>> def h(fonc, x):
... return fonc(x)
...

>>> h(f, 3)
7

>>> h(g, 4)
2
```

5.3.5 Paramètres avec valeur par défaut

Il est possible de spécifier, lors de la déclaration, des valeurs par défaut à utiliser pour les arguments. Cela permet, lors de l'appel, de ne pas avoir à spécifier les paramètres correspondants.

Il est également possible, en combinant les valeurs par défaut et le nommage des paramètres, de n'indiquer à l'appel que les paramètres dont on désire modifier la valeur de l'argument. Il est par contre nécessaire de regrouper tous les paramètres optionnels à la fin de la liste des paramètres.

```
>>> def accueil(nom, prenom, depart="MP", semestre="S2"):
... print(prenom, nom, "Département", depart, "semestre", semestre)
...

>>> accueil("Student", "Joe")
Joe Student Département MP semestre S2

>>> accueil("Student", "Eve", "Info")
Eve Student Département Info semestre S2

>>> accueil("Student", "Steph", semestre="S3")
Steph Student Département MP semestre S3
```

Attention

☢ On utilise de préférence des valeurs par défaut non modifiables (`int`, `float`, `str`, `bool`, `tuple`) car la modification d'un paramètre par un premier appel est visible les fois suivantes.

Si on a besoin d'une valeur par défaut qui soit modifiable (`list`, `dict`), on utilise la valeur prédéfinie `None` et on fait un test dans la fonction avant modification :

```
def maFonction(liste=None):
   if liste is None:
      liste = [1, 3]
```

5.3.6 Nombre d'arguments arbitraire : passage d'un tuple de valeurs

Le passage d'un nombre arbitraire d'arguments est permis en utilisant la notation d'un argument final `*nom`. Les paramètres surnuméraires sont alors transmis sous la forme d'un tuple affecté à cet argument (que l'on appelle généralement `args`).

```
def somme(*args):
   """Renvoie la somme du tuple <args>."""
   resultat = 0
   for nombre in args:
      resultat += nombre
   return resultat
```

```
# Exemples d'appel :
print(somme(23)) # 23
print(somme(23, 42, 13)) # 78
```

Attention

☢ Si la fonction possède plusieurs arguments, le tuple est en *dernière* position.

Réciproquement, il est aussi possible de passer un tuple (en fait une séquence) à l'appel qui sera *décompressé* en une liste de paramètres d'une fonction « classique ».

```
def somme(a, b, c):
   return a+b+c

# Exemple d'appel :
elements = (2, 4, 6)
print(somme(*elements)) # 12
```

5.3.7 Nombre d'arguments arbitraire : passage d'un dictionnaire

De la même façon, il est possible d'autoriser le passage d'un nombre arbitraire d'arguments nommés en plus de ceux prévus lors de la définition en utilisant la notation d'un argument final `**nom`. Les paramètres surnuméraires nommés sont alors transmis sous la forme d'un dictionnaire affecté à cet argument (que l'on appelle généralement **kwargs** pour *keyword args*).

Réciproquement il est aussi possible de passer un dictionnaire à l'appel d'une fonction, qui sera décompressé et associé aux paramètres nommés de la fonction.

```
def unDict(**kwargs):
   return kwargs

# Exemples d'appels
## par des paramètres nommés :
print(unDict(a=23, b=42)) # {'a': 23, 'b': 42}

## en fournissant un dictionnaire :
mots = {'d': 85, 'e': 14, 'f':9}
print(unDict(**mots)) # {'e': 14, 'd': 85, 'f': 9}
```

Attention

☢ Si la fonction possède plusieurs arguments, le dictionnaire est en *toute dernière* position (après un éventuel tuple).

5.4 Espaces de noms

Un espace de noms est une notion permettant de lever une ambiguïté sur des termes qui pourraient être *homonymes* sans cela. Il est matérialisé par un préfixe identifiant de manière unique la signification d'un terme. Au sein d'un même espace de noms, il n'y a pas d'homonymes :

```
>>> import math

>>> pi = 2.718

>>> print(pi) # une valeur de l'espace de nom local
2.718

>>> print(math.pi) # une valeur de l'espace de noms math
3.141592653589793
```

5.4.1 Portée des objets

On distingue :

La portée globale : celle du module ou du fichier script en cours. Un dictionnaire gère les objets globaux : l'instruction `globals()` fournit un dictionnaire contenant les couples `nom:valeur` ;

La portée locale : les objets internes aux fonctions sont locaux. Les objets globaux ne sont *pas modifiables* dans les portées locales. L'instruction `locals()` fournit un dictionnaire contenant les couples `nom:valeur`.

5.4.2 Résolution des noms : règle « LGI »

La recherche des noms est d'abord locale (L), puis globale (G), enfin interne (I) (☞ Fig. 5.3) :

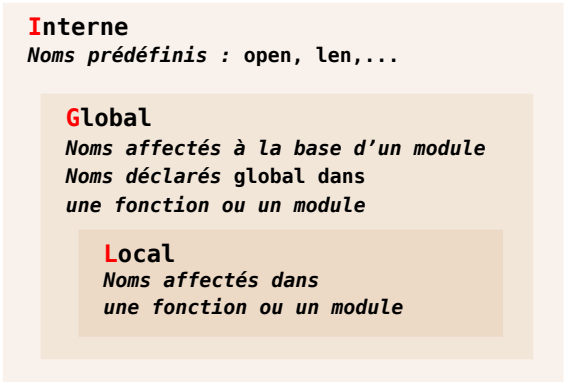


FIGURE 5.3 – Règle LGI

Exemples de portée

Par défaut, tout identificateur utilisé dans le corps d'une fonction est local à celle-ci. Si une fonction a besoin de modifier certains identificateurs globaux, la première instruction de cette fonction doit être :

```
global <identificateurs>
```

Par exemple :

```
# x et fonc sont affectés dans le module => globaux

def fonc(y): # y et z sont affectés dans fonc => locaux
   global x # permet de modifier x ligne suivante
   x = x + 2
   z = x + y
   return z

x = 99
print(fonc(1)) # 102
print(x) # 101
```

```
# x et fonc sont affectés dans le module => globaux

def fonc(y): # y et z sont affectés dans fonc => locaux
   # dans fonc : portée locale
   z = x + y
   return z

x = 99
print(fonc(1)) # 100
print(x) # 99
```

```
# x et fonc sont affectés dans le module => globaux

def fonc(y): # y, x et z sont affectés dans fonc => locaux
   x = 3 # ce nouvel x est local et masque le x global
   z = x + y
   return z

x = 99
print(fonc(1)) # 4
print(x) # 99
```

Chapitre 6

Modules et packages

Un programme Python est généralement composé de plusieurs fichiers sources, appelés *modules*.

S'ils sont correctement codés les modules doivent être indépendants les uns des autres pour être réutilisés à la demande dans d'autres programmes.

Ce chapitre explique comment coder des modules et comment les importer dans un autre.

Nous verrons également la notion de *package* qui permet de grouper plusieurs modules.

6.1 Modules

Définition

✍ Module : fichier script Python permettant de définir des élements de programme réutilisables. Ce mécanisme permet d'élaborer efficacement des bibliothèques de fonctions ou de classes.

Avantages des modules :

- réutilisation du code ;
- la documentation et les tests peuvent être intégrés au module ;
- réalisation de services ou de données partagés ;
- Le mécanisme crée un nouvel espace de noms et exécute toutes les instructions du fichier `.py` associé dans cet espace de noms.

Remarque

✓ Lorsqu'on parle du module, on omet l'extension : le module `machin` est dans le fichier `machin.py`.

6.1.1 Import

L'instruction `import` charge et exécute le module indiqué s'il n'est pas déjà chargé. L'ensemble des définitions contenues dans ce module deviennent alors disponibles : variables globales, fonctions, classes.

Suivant la syntaxe utilisée, on accède aux définitions du module de différentes façons :

- l'instruction `import <nom_module>` donne accès à l'ensemble des définitions du module importé en utilisant le nom du module comme espace de nom.

```
>>> import tkinter

>>> print("Version de tkinter :", tkinter.TkVersion)
Version de tkinter : 8.5
```

— l'instruction `from <nom_module> import nom1, nom2...` donne accès directement à une sélection choisie de noms définis dans le module.

```
>>> from math import pi, sin

>>> print("Valeur de Pi :", pi, "sinus(pi/4) :", sin(pi/4))
Valeur de Pi : 3.14159265359 sinus(pi/4) : 0.707106781187
```

Dans les deux cas, le module et ses définitions existent dans leur espace mémoire propre, et on duplique simplement dans le module courant les noms que l'on a choisis, comme si on avait fait les affectations :

```
>>> sin = math.sin

>>> pi = math.pi
```

Remarque

✓ Il est conseillé d'importer dans l'*ordre* :
— les modules de la bibliothèque standard ;
— les modules des bibliothèques tierces ;
— Les modules personnels.

Attention

☢ Pour tout ce qui est fonction et classe, ainsi que pour les « constantes » (variables globales définies et affectées une fois pour toutes à une valeur), l'import direct du nom ne pose pas de problème.

Par contre, pour les variables globales que l'on désire pouvoir modifier, il est préconisé de passer systématiquement par l'espace de nom du module afin de s'assurer de l'existence de cette variable en un unique exemplaire ayant la même valeur dans tout le programme.

6.1.2 Exemples

Notion d'« auto-test »

Le *module principal* est celui qui est donné en argument sur la ligne de commande ou qui est lancé en premier lors de l'exécution d'un script. Son nom est contenu dans la variable globale `__name__`. Sa valeur dépend du contexte de l'exécution.

Soit le module :

```
# je_me_nomme.py
print("Je me nomme :", __name__)
```

Premier contexte exécution sur la ligne de commande (ou dans un EDI), on obtient la valeur de la variable prédéfinie `__main__` :

```
$ python3 je_me_nomme.py
Je me nomme : __main__
```

Second contexte import de ce module (ce n'est donc plus le module principal), on obtient l'identificateur du module :

```
>>> import je_me_nomme
Je me nomme : je_me_nomme
```

Grâce à un test, on peut donc facilement savoir si le code est exécuté en tant que script principal :

— on écrit un script de fonctions ou de classes (souvent appelé *bibliothèque*) et on termine le fichier par un test, l'« auto-test », pour vérifier que l'on est dans le module principal. On en profite pour vérifier tous les éléments de la bibliothèque ;
— quand on importe le script, le test inclus est faux et on se borne à utiliser la bibliothèque.

```
# cube.py

def cube(x):
   """retourne le cube de <x>."""
   return x**3

# auto-test ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
if __name__ == "__main__": # vrai car module principal
   if cube(9) == 729:
      print("OK !")
   else:
      print("KO !")
```

Utilisation de ce module dans un autre (par exemple celui qui contient le programme principal) :

```
import cube

# programme principal ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
for i in range(1, 4):
   print("cube de", i, "=", cube_m.cube(i))

"""
cube de 1 = 1
cube de 2 = 8
cube de 3 = 27
"""
```

Autre exemple :

```
def ok(message) :
   """
   Retourne True si on saisit <Entrée>, <O>, <o>, <Y> ou <y>,
   False dans tous les autres cas.
   """
   s = input(message + " (O/n) ? ")
   return True if s == "" or s[0] in "OoYy" else False


# auto-test ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
if __name__ == '__main__' :
   while True:
      if ok("Encore"):
         print("Je continue")
      else:
         print("Je m'arrête")
         break

"""
Encore (O/n) ?
Je continue
Encore (O/n) ? o
Je continue
Encore (O/n) ? n
Je m'arrête
"""
```

6.2 *Batteries included*

On dit souvent que Python est livré « piles comprises » (*batteries included*) tant sa bibliothèque standard, riche de plus de 200 packages et modules, répond aux problèmes courants les plus variés.

Ce survol présente quelques fonctionnalités utiles.

La gestion des chaînes

Le module `string` fournit des constantes comme `ascii_lowercase`, `digits`…ainsi que la classe `Formatter` qui peut être spécialisée en sous-classes de *formateurs* de chaînes.

Le module `textwrap` est utilisé pour formater un texte : longueur de chaque ligne, contrôle de l'indentation.

Le module `struct` permet de convertir des nombres, booléens et des chaînes en leur représentation binaire afin de communiquer avec des bibliothèques de bas-niveau (souvent en C).

Le module `difflib` permet la comparaison de séquences et fournit des sorties au format standard « diff » ou en HTML.

Enfin, on ne saurait oublier le module `re` qui offre à Python la puissance des expressions régulières [1].

La gestion de la ligne de commande

Pour gérer la ligne de commande, Python propose l'instruction `sys.argv`. Elle fournit simplement une liste contenant les arguments de la ligne de commande : `argv[1], argv[2]`...sachant que `argv[0]` est le nom du script lui-même.

Par ailleurs, Python propose un module de *parsing* [2], le module `argparse`.
C'est un module objet qui s'utilise en trois étapes :

1. Création d'un objet `parser` ;
2. Ajout des arguments prévus en utilisant la méthode `add_argument()`. Chaque argument peut déclencher une action particulière spécifiée dans la méthode ;
3. Analyse de la ligne de commande par la méthode `parse_args()`.

Enfin, selon les paramètres détectés par l'analyse, on effectue les actions adaptées.

Dans l'exemple suivant, extrait de la documentation officielle du module, on se propose de donner en argument à la ligne de commande une liste d'entiers. Par défaut le programme retourne le plus grand entier de la liste, mais s'il détecte l'argument –`sum`, il retourne la somme des entiers de la liste. De plus, lancé avec l'option `-h` ou –`help`, le module `argparse` fournit automatiquement une documentation du programme :

```
# -*- coding: utf8 -*-
import argparse

# 1. création du parser
parser = argparse.ArgumentParser(description='Process some integers')

# 2. ajout des arguments
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')

parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max')

# 3. parsing de la ligne de commande
args = parser.parse_args()

# processing
print(args.accumulate(args.integers))
```

Voici les sorties correspondant aux différents cas de la ligne de commande :

```
$ python argparse.py -h
usage: argparse.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
  N           an integer for the accumulator

optional arguments:
  -h, --help  show this help message and exit
  --sum       sum the integers (default: find the max)

$ python argparse.py --help
usage: argparse.py [-h] [--sum] N [N ...]

Process some integers.
```

1. C'est tout un monde...
2. analyse grammaticale de la ligne de commande.

> Retour *Table des matières*

```
positional arguments:
  N           an integer for the accumulator

optional arguments:
  -h, --help  show this help message and exit
  --sum       sum the integers (default: find the max)

$ python argparse.py 1 2 3 4 5 6 7 8 9
9

$ python argparse.py 1 2 3 4 5 6 7 8 9 --sum
45
```

La gestion du temps et des dates

Les modules `calendar`, `time` et `datetime` fournissent les fonctions courantes de gestion du temps et des durées :

```
import calendar, datetime, time

moon_apollo11 = datetime.datetime(1969, 7, 20, 20, 17, 40)
print(moon_apollo11)
print(time.asctime(time.gmtime(0)))
# Thu Jan 01 00:00:00 1970 ("epoch" UNIX)

vendredi_precedent = datetime.date.today()
un_jour = datetime.timedelta(days=1)
while vendredi_precedent.weekday() != calendar.FRIDAY:
   vendredi_precedent -= un_jour
print(vendredi_precedent.strftime("%A, %d-%b-%Y"))
# Friday, 09-Oct-2009
```

Algorithmes et types de données `collection`

Le module `bisect` fournit des fonctions de recherche de séquences triées. Le module `array` propose un type semblable à la liste, mais plus rapide, car de contenu homogène.

Le module `heapq` gère des *tas* dans lesquels l'élément d'index o est toujours le plus petit :

```
import heapq
import random

heap = []
for i in range(10):
   heapq.heappush(heap, random.randint(2, 9))

print(heap) # [2, 3, 5, 4, 6, 6, 7, 8, 7, 8]
```

À l'instar des structures C, Python propose désormais, via le module `collections`, la notion de type tuple nommé (il est bien sûr possible d'avoir des tuples nommés emboîtés) :

```
import collections

# description du type :
Point = collections.namedtuple("Point", "x y z")
# on instancie un point :
point = Point(1.2, 2.3, 3.4)
# on l'affiche :
print("point : [{}, {}, {}]"
   .format(point.x, point.y, point.z)) # point : [1.2, 2.3, 3.4]
```

Le type `defaultdict` permet des utilisations avancées :

```
from collections import defaultdict

s = [('y', 1), ('b', 2), ('y', 3), ('b', 4), ('r', 1)]
d = defaultdict(list)
for k, v in s:
```

```
    d[k].append(v)
print(d.items())
# dict_items([('y', [1, 3]), ('r', [1]), ('b', [2, 4])])

s = 'mississippi'
d = defaultdict(int)
for k in s:
    d[k] += 1
print(d.items())
# dict_items([('i', 4), ('p', 2), ('s', 4), ('m', 1)])
```

Et tant d'autres domaines…

Beaucoup d'autres sujets pourraient être explorés :

— accès au système ;
— utilitaires fichiers ;
— programmation réseau ;
— persistance ;
— les fichiers XML ;
— la compression ;
— …

6.3 Python scientifique

Dans les années 1990, Travis Oliphant et d'autres commencèrent à élaborer des outils efficaces de traitement des données numériques : *Numeric*, *Numarray*, et enfin *NumPy*. *SciPy*, bibliothèque d'algorithmes scientifiques, a également été créée à partir de *NumPy*. Au début des années 2000, John Hunter crée *matplotlib* un module de tracer de graphiques 2D. À la même époque Fernando Perez crée *IPython* en vue d'améliorer l'interactivité et la productivité en Python.

En moins de 10 ans après sa création, les outils essentiels pour faire de Python un langage scientifique performant étaient en place.

6.3.1 Bibliothèques mathématiques et types numériques

On rappelle que Python possède la bibliothèque `math` :

```
>>> import math

>>> math.pi / math.e
1.1557273497909217

>>> math.exp(1e-5) - 1
1.0000050000069649e-05

>>> math.log(10)
2.302585092994046

>>> math.log(1024, 2)
10.0

>>> math.cos(math.pi/4)
0.7071067811865476

>>> math.atan(4.1/9.02)
0.4266274931268761

>>> math.hypot(3, 4)
5.0

>>> math.degrees(1)
57.29577951308232
```

Par ailleurs, Python propose en standard les modules `fraction` et `decimal` :

```python
from fractions import Fraction
import decimal as d

print(Fraction(16, -10)) # -8/5
print(Fraction(123)) # 123
print(Fraction(' -3/7 ')) # -3/7
print(Fraction('-.125')) # -1/8
print(Fraction('7e-6')) # 7/1000000

d.getcontext().prec = 6
print(d.Decimal(1) / d.Decimal(7)) # 0.142857
d.getcontext().prec = 18
print(d.Decimal(1) / d.Decimal(7)) # 0.142857142857142857
```

En plus des bibliothèques `math` et `cmath` déjà vues, la bibliothèque `random` propose plusieurs fonctions de nombres aléatoires.

6.3.2 L'interpréteur IPython

Remarque

✓ On peut dire que `IPython` est devenu *de facto* l'interpréteur standard du Python scientifique.

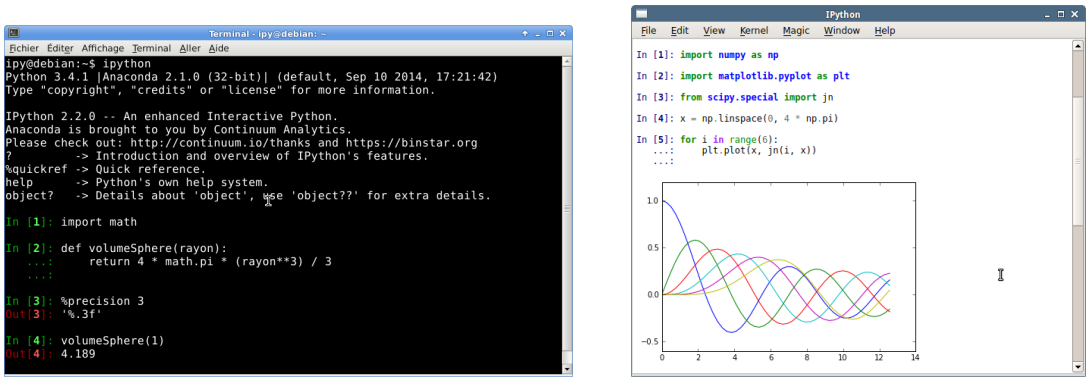
En mars 2013, ce projet a valu le Prix du développement logiciel libre par la *Free Software Foundation* à son créateur Fernando Perez. Depuis début 2013 et pour deux ans, la fondation Alfred P. Sloan subventionne le développement de `IPython`.

`IPython` (actuellement en version 2.4.1 dans `Pyzo` version `2015a`) est disponible en trois déclinaisons (☞ Fig. 6.1 et 6.2) :

`ipython` l'interpréteur de base ;

`ipython qtconsole` sa version améliorée dans une fenêtre graphique de la bibliothèque graphique Qt, agrémentée d'un menu ;

`ipython notebook` sa dernière variante qui offre une interface simple, mais très puissante dans le navigateur par défaut.



(a) `ipython` (b) `ipython qtconsole`

FIGURE 6.1 – `IPython` en mode console texte ou graphique.

La version *notebook* mérite une mention spéciale : chaque cellule du notebook peut être du code, des figures, du texte enrichi (y compris des formules mathématiques), des vidéos, etc.

La figure 6.2 présente un exemple de tracé interactif.

Objectifs

D'après ses concepteurs, les objectifs d'`IPython` sont les suivants :

— fournir un interpréteur Python plus puissant que celui par défaut. `IPython` propose de nombreuses caractéristiques comme l'*introspection d'objet*, l'*accès au shell système* ainsi que ses propres commandes permettant une grande interaction avec l'utilisateur ;

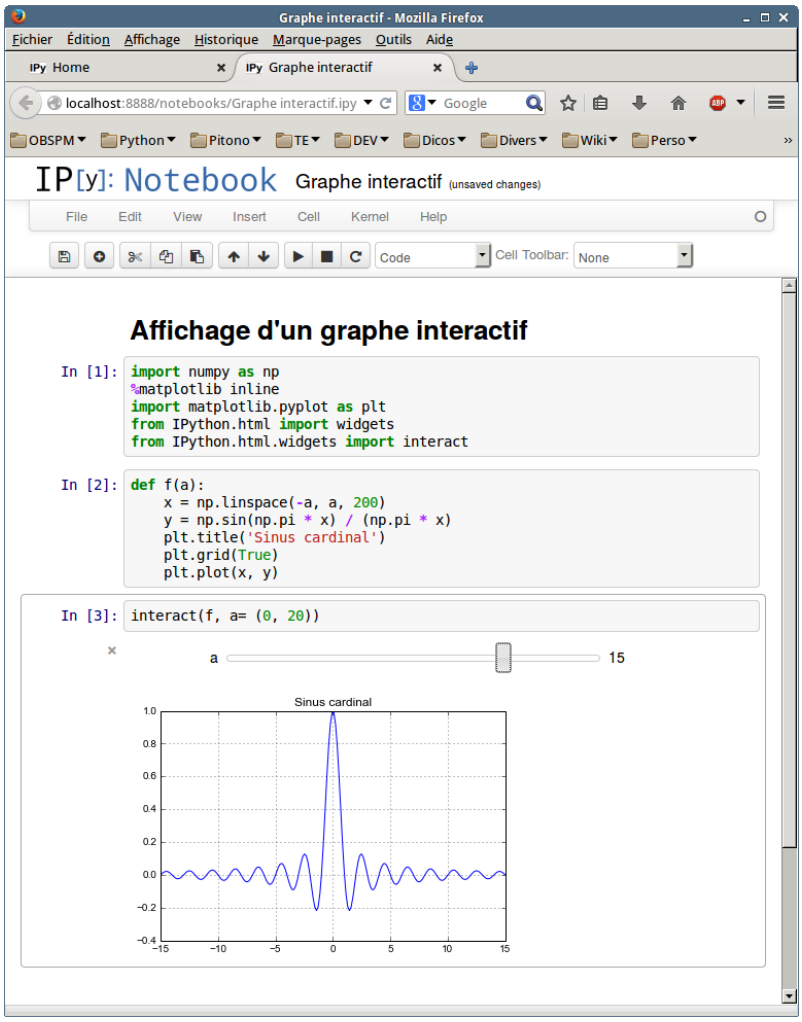


FIGURE 6.2 – `ipython notebook`

— proposer un interpréteur *embarquable* et prêt à l'emploi pour vos programmes Python. `IPython` s'efforce d'être un environnement efficace à la fois pour le développement de code Python et pour la résolution des problèmes liés à l'utilisation d'objets Python ;
— offrir un ensemble de bibliothèques pouvant être utilisé comme environnement pour d'autres systèmes utilisant Python comme langage sous-jacent (particulièrement pour les environnements scientifiques comme `matlab`, `mapple` ou `mathematica`) ;
— permettre le test interactif des bibliothèques graphiques gérées comme `Tkinter`, `wxPython`, `PyGTK` alors que `IDLE` ne le permet qu'avec des applications `Tkinter`.

Quelques caractéristiques

— `IPython` est auto-documenté ;
— coloration syntaxique ;
— les *docstrings* des objets Python sont disponibles en accolant un « ? » au nom de l'objet ou « ?? » pour une aide plus détaillée ;
— numérote les entrées et les sorties ;
— organise les sorties : messages d'erreur ou retour à la ligne entre chaque élément d'une liste si on l'affiche ;
— auto-complétion avec la touche `TAB` :
  — l'auto-complétion trouve les variables qui ont été déclarées,
  — elle trouve les mots clés et les fonctions locales,
  — la complétion des méthodes sur les variables tient compte du type actuel de cette dernière,
  — par contre la complétion ne tient pas compte du contexte ;

— historique persistant (même si on quitte l'interpréteur, on peut retrouver les dernières commandes par l'historique) :
  — recherche dans l'historique avec les flèches du clavier,
  — isole dans l'historique les entrées multilignes,
  — on peut appeler les entrées et sorties précédentes ;
— contient des raccourcis et des alias. On peut en afficher la liste en tapant la commande `lsmagic` ;
— permet d'exécuter des commandes système en les préfixant par un point d'exclamation. Par exemple `!ls` sous Linux ou OSX, ou `!dir` sous une fenêtre de commande Windows.

6.3.3 La bibliothèque `NumPy`

Introduction

Le module `numpy` est la boîte à outils indispensable pour faire du calcul scientifique avec Python [1].

Pour modéliser les vecteurs, matrices et, plus généralement, les tableaux à n dimensions, `numpy` fournit le type `ndarray`.

On note des différences majeures avec les listes (resp. les listes de listes) qui pourraient elles aussi nous servir à représenter des vecteurs (resp. des matrices) :

— les tableaux `numpy` sont *homogènes*, c'est-à-dire constitués d'éléments du même type.
  On trouvera donc des tableaux d'entiers, des tableaux de flottants, des tableaux de chaînes de caractères, etc.
— la taille des tableaux `numpy` est fixée à la création. On ne peut donc augmenter ou diminuer la taille d'un tableau comme on le ferrait pour une liste (à moins de créer un tout nouveau tableau, bien sûr).

Ces contraintes sont en fait des avantages :

— le format d'un tableau `numpy` et la taille des objets qui le composent étant fixés, l'empreinte du tableau en mémoire est invariable et l'accès à ses éléments se fait en temps constant ;
— les opérations sur les tableaux sont optimisées en fonction du type des éléments, et sont beaucoup plus rapides qu'elles ne le seraient sur des listes équivalentes.

Exemples

Dans ce premier exemple, on définit un tableau `a` d'entiers puis on le multiplie *globalement*, c'est-à-dire sans utiliser de boucle, par le scalaire `2.5`. On définit de même le tableau `d` qui est affecté en une seule instruction à `a + b`.

```
In [1]: import numpy as np

In [2]: a = np.array([1, 2, 3, 4])

In [3]: a # a est un tableau d'entiers
Out[3]: array([1, 2, 3, 4])

In [4]: b = a * 2.5

In [5]: b # b est bien devenu un tableau de flottants
Out[5]: array([ 2.5, 5. , 7.5, 10. ])

In [6]: c = np.array([5, 6, 7, 8])

In [7]: d = b + c

In [8]: d # transtypage en flottants
Out[8]: array([ 7.5, 11. , 14.5, 18. ])
```

L'exemple suivant définit un tableau `positions` de 10 000 000 lignes et 2 colonnes, formant des positions aléatoires. Les vecteurs colonnes `x` et `y` sont extraits du tableau `position`. On affiche le tableau et le vecteur `x` avec 3 chiffres après le point décimal. On calcule (bien sûr *globalement*) le vecteur des distances euclidiennes à un point particulier (x_0, y_0) et on affiche l'indice du tableau de la distance minimale à ce point.

1. Cette introduction est reprise de l'excellente présentation de Jean-Michel Ferrard [cf. p. 127].

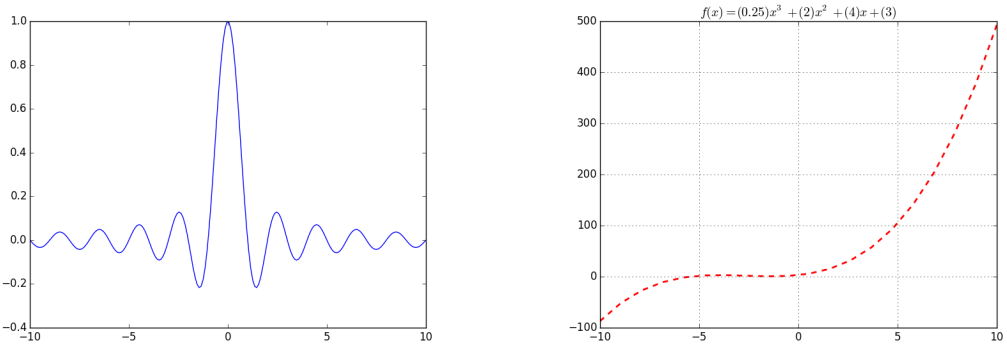
```
In [1]: import numpy as np

In [2]: positions = np.random.rand(10000000, 2)

In [3]: x, y = positions[:, 0], positions[:, 1]

In [4]: %precision 3
Out[4]: '%.3f'

In [5]: positions
Out[5]:
array([[ 0.861, 0.373],
       [ 0.627, 0.935],
       [ 0.224, 0.058],
       ...,
       [ 0.628, 0.66 ],
       [ 0.546, 0.416],
       [ 0.396, 0.625]])

In [6]: x
Out[6]: array([ 0.861, 0.627, 0.224, ..., 0.628, 0.546, 0.396])

In [7]: x0, y0 = 0.5, 0.5

In [8]: distances = (x - x0)**2 + (y - y0)**2

In [9]: distances.argmin()
Out[9]: 4006531
```

Ce type de traitement très efficace et élégant est typique des logiciels analogues comme MATLAB.

6.3.4 La bibliothèque `matplotlib`

Cette bibliothèque permet toutes sortes de représentations [1] de graphes 2D (et quelques unes en 3D) :

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-10, 10, 200)
y = np.sin(np.pi * x)/(np.pi * x)

plt.plot(x, y)
plt.show()
```



(a) Un tracé simple

(b) Un tracé décoré

FIGURE 6.3 – Tracé d'un sinus cardinal.

Ce second exemple améliore le tracé. Il utilise le *style objet* de `matplotlib` :

1. Notons que sa syntaxe de base a été pensée pour ne pas dépayser l'utilisateur de la bibliothèque graphique de MATLAB.

```python
import numpy as np
import matplotlib.pyplot as plt

def plt_arrays(x, y, title='', color='red', linestyle='dashed', linewidth=2):
   """Définition des caractéristiques et affichage d'un tracé y(x)."""
   fig = plt.figure()
   axes = fig.add_subplot(111)
   axes.plot(x, y, color=color, linestyle=linestyle, linewidth=linewidth)
   axes.set_title(title)
   axes.grid()
   plt.show()


def f(a, b, c, d):
   x = np.linspace(-10, 10, 20)
   y = a*(x**3) + b*(x**2) + c*x + d
   title = '$f(x) = (%s)x^3 + (%s)x^2 + (%s)x + (%s)$' % (a, b, c, d)
   plt_arrays(x, y, title=title)


f(0.25, 2, 4, 3)
```

6.3.5 La bibliothèque `SymPy`

Introduction

`SymPy` est une bibliothèque en pur Python spécialisée dans le calcul formel à l'instar de Mapple ou Mathematica [1]. Elle permet de faire du calcul arithmétique formel basique, de l'algèbre, des mathématiques différentielles, de la physique, de la mécanique…

Il est facile de se familiariser avec `SymPy` en l'expérimentant directement sur Internet [2].

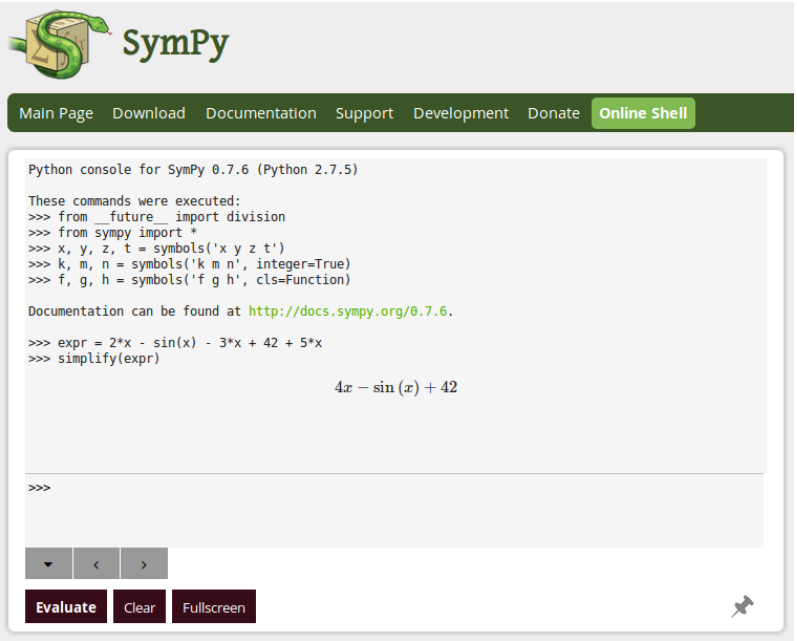


FIGURE 6.4 – Expérimenter `SymPy` avec « live sympy »

Exemples

Si l'interpréteur `IPython` est déjà installé, une alternative intéressante est fournie par le script `isympy`. Voici quelques exemples des très nombreuses possibilités offertes par cette bibliothèque :

1. On parle aussi de CAS (*Computer Algebra System*).
2. `http://live.sympy.org`

```
In [1]: P = (x-1) * (x-2) * (x-3)

In [2]: P.expand()
Out[2]:
 3 2
x - ·6x + ·11x - 6

In [3]: P.factor()
Out[3]: (x - 3)·(x - 2)·(x - 1)

In [4]: roots = solve(P, x)

In [5]: roots
Out[5]: [1, 2, 3]

In [6]: sin(pi/6)
Out[6]: 1/2

In [7]: cos(pi/6)
Out[7]:
  ___
╲╱ 3
─────
  2

In [8]: tan(pi/6)
Out[8]:
  ___
╲╱ 3
─────
  3

In [9]: e = 2*sin(x)**2 + 2*cos(x)**2

In [10]: trigsimp(e)
Out[10]: 2

In [11]: z = 4 + 3*I

In [12]: Abs(z)
Out[12]: 5

In [13]: arg(z)
Out[13]: atan(3/4)

In [14]: f = x**2 * sin(x)

In [15]: diff(f, x)
Out[15]:
 2
x ·cos(x) + ·2·xsin(x)
```

6.4 Bibliothèques tierces

6.4.1 Une grande diversité

Outre les nombreux modules intégrés à la distribution standard de Python, on trouve des bibliothèques dans tous les domaines :

- scientifique ;
- bases de données ;
- tests fonctionnels et contrôle de qualité ;
- 3D ;
- ...

Le site pypi.python.org/pypi [1] recense des milliers de modules et de packages !

1. The **py**thon **p**ackage **i**ndex

6.4.2 Un exemple : la bibliothèque `Unum`

Cette bibliothèque permet de calculer en tenant compte des unités du système SI (Système International d'unités).

Voici un exemple de session interactive :

```
>>> from unum.units import *

>>> distance = 100*m

>>> temps = 9.683*s

>>> vitesse = distance / temps

>>> vitesse
10.327377878756584 [m/s]

>>> vitesse.asUnit(mile/h)
23.1017437978 [mile/h]

>>> acceleration = vitesse/temps

>>> acceleration
1.0665473385063085 [m/s2]
```

6.5 Packages

Outre le module, un deuxième niveau d'organisation permet de structurer le code : les fichiers Python peuvent être organisés en une arborescence de répertoires appelée paquet, en anglais *package*.

Définition

✍ Un *package* est un module contenant d'autres modules. Les modules d'un package peuvent être des *sous-packages*, ce qui donne une structure arborescente.

Pour être reconnu comme un package valide, chaque répertoire du paquet doit posséder un fichier `__init__` qui peut soit être vide soit contenir du code d'initialisation.

Chapitre 7

La Programmation Orientée Objet

La *Programmation Orientée Objet* :

- la *POO* permet de mieux modéliser la réalité en concevant des modèles d'objets, les *classes* ;
- ces classes permettent de construire des *objets* interactifs entre eux et avec le monde extérieur ;
- les objets sont créés indépendamment les uns des autres, grâce à l'*encapsulation*, mécanisme qui permet d'embarquer leurs propriétés ;
- les classes permettent d'éviter au maximum l'emploi des variables globales ;
- enfin les classes offrent un moyen économique et puissant de construire de nouveaux objets à partir d'objets préexistants.

7.1 Terminologie

Le vocabulaire de base de la POO

Une classe est équivalente à un nouveau type de données. On connaît déjà par exemple les classes `list` ou `str` et les nombreuses méthodes permettant de les manipuler, par exemple :

- `[3, 5, 1].sort()`
- `"casse".upper()`

Un objet ou une instance est un exemplaire particulier d'une classe. Par exemple `[3, 5, 1]` est une instance de la classe `list` et `"casse"` est une instance de la classe `str`.

Les objets ont généralement deux sortes d'attributs : les données nommées simplement attributs et les fonctions applicables appelées méthodes.

Par exemple un objet de la classe `complex` possède :

- deux attributs : `imag` et `real` ;
- plusieurs méthodes, comme `conjugate()`, `abs()`...

La plupart des classes encapsulent à la fois les données et les méthodes applicables aux objets. Par exemple un objet `str` contient une chaîne de caractères Unicode (les données) et de nombreuses méthodes.

On peut définir un objet comme une *capsule* contenant des attributs et des méthodes :

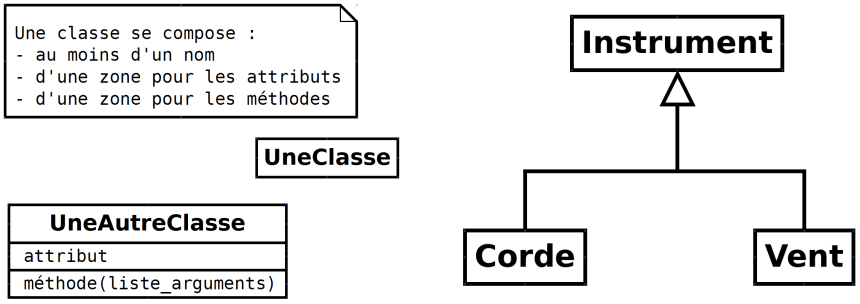
```
objet = attributs + méthodes
```

7.1.1 Notations UML de base

Remarque

✓ L'UML (*Unified Modeling Language*) est un langage graphique très répandu de conception des systèmes d'information.

UML propose une grande variété de diagrammes (classes, objets, états, activités, etc.). En première approche, le diagramme de classes est le plus utile pour concevoir les classes et leurs relations.



(a) Une classe et une note (b) Un héritage

FIGURE 7.1 – Diagrammes de classe.

7.2 Classes et instanciation d'objets

7.2.1 L'instruction `class`

Cette instruction permet d'introduire la définition d'une nouvelle classe (c'est-à-dire d'un nouveau type de données).

Syntaxe

✎ `class` est une instruction composée. Elle comprend un en-tête (avec *docstring*) + corps indenté :

```
>>> class C:
... """Documentation de la classe C."""
... x = 23
```

Dans cet exemple, `C` est le nom de la *classe* (qui commence conventionnellement par une majuscule), et `x` est un *attribut de classe*, local à `C`.

7.2.2 L'instanciation et ses attributs

- Les classes sont des *fabriques d'objets* : on construit d'abord l'usine avant de produire des objets !
- On instancie un objet (c'est-à-dire qu'on le produit à partir de l'*usine*) en appelant le nom de sa classe comme s'il s'agissait d'une fonction :

```
class C:
    """Documentation de la classe C."""
    x = 23 # attribut de classe
    y = 'un string'
    z = [1, x, 3, y]


# a est un objet de la classe C (ou une instance)
a = C()
# Affichage des attributs de l'instance a
print(a.x)
print(a.y)
print(a.z)
```

Remarque

✓ En Python (car c'est un langage *dynamique* comme Ruby, contrairement à C++ ou Java) il est possible d'ajouter de nouveaux attributs d'instance (ici le `a.y = 44`) ou même de nouveaux attributs de classe (ici `C.z = 6`).

Définition

✍ Une variable définie au niveau d'une classe (comme `x` dans la classe `C`) est appelé attribut de classe et est partagée par tous les objets instances de cette classe.

Une variable définie au niveau d'un objet (comme `y` dans l'objet `a`) est appelée attribut d'instance et est liée uniquement à l'objet pour lequel elle est définie.

7.2.3 Retour sur les espaces de noms

On a déjà vu les espaces de noms [1] locaux (lors de l'appel d'une fonction), globaux (liés aux modules) et internes (fonctions standard), ainsi que la règle « Local Global Interne » (cf. section 5.3, p. 49) qui définit dans quel ordre ces espaces sont parcourus pour résoudre un nom.

Les classes ainsi que les objets instances définissent de nouveaux espaces de noms, et il y a là aussi des règles pour résoudre les noms :

- les classes peuvent utiliser les variables définies au niveau principal, mais elles ne peuvent pas les modifier ;
- les instances peuvent utiliser les variables définies au niveau de la classe, mais elles ne peuvent pas les modifier (pour cela elles sont obligées de passer par l'espace de noms de la classe, par exemple `C.x = 3`).

Exemple de masquage d'attribut :

```
class C:
   """Documentation de la classe C."""
   x = 23 # attibut de classe

a = C() # a est un objet de la classe C (ou une instance)
a.x # 23 : affiche la valeur de l'attribut de l'instance a
a.x = 12 # modifie son attribut d'instance (attention...)
a.x # 12
C.x # 23 mais l'attribut de classe est inchangé
C.z = 6 # z : nouvel attribut de classe
a.y = 44 # y : nouvel attribut de l'instance a
b = C() # b est un autre objet de la classe C (une autre instance)
b.x # 23 : b connaît bien son attribut de classe, mais...
b.y # ... b n'a pas d'attribut y !
"""
Retraçage (dernier appel le plus récent) :
  Fichier "/home/ipy/Phi/3-corps/ch07/src/7_025.py", ligne 14, dans 0
builtins.AttributeError: 'C' object has no attribute 'y'
"""
```

Recherche des noms

- **Noms non qualifiés** (exemple `dimension`) l'affectation crée ou change le nom dans la *portée* locale courante. Ils sont cherchés suivant la règle LGI.
- **Noms qualifiés** (exemple `dimension.hauteur`) l'affectation crée ou modifie l'attribut dans l'espace de noms de l'objet. Un attribut est cherché dans l'objet, puis dans toutes les classes dont l'objet dépend (mais pas dans les modules).

```
>>> v = 5

>>> class C:
... x = v + 3 # utilisation d'une variable globale dans la définition de classe
... y = x + 1 # recherche dans l'espace de noms de la classe lors de la définition
...

>>> a = C()
```

1. Les espaces de noms sont implémentés par des dictionnaires.

```
>>> a.x # utilisation sans modification de la variable de classe en passant par l'objet
8

>>> a.x = 2 # création d'une variable d'instance pour l'objet a

>>> a.x
2

>>> C.x # la variable de classe n'est pas modifiée
8

>>> C.x = -1 # on peut modifier la variable de classe en passant par l'espace de noms de la classe

>>> C.x
-1
```

À chaque création d'une classe `C`, Python lui associe un dictionnaire (de nom `C.__dict__`) contenant un ensemble d'informations. L'exemple suivant affiche le dictionnaire lié à la classe `C` :

```
In [1]: class C:
  ...: """Une classe simple."""
  ...: x = 2
  ...: y = 5
  ...:

In [2]: C.__dict__
Out[2]: mappingproxy({'x': 2, '__module__': '__main__', '__weakref__': <attribute '__weakref__' of 'C' objects>, '
     __dict__': <attribute '__dict__' of 'C' objects>, 'y': 5, '__doc__': 'Une classe simple.'})

In [3]: for key in C.__dict__.keys():
  ...: print(key)
  ...:
x
__module__
__weakref__
__dict__
y
__doc__

In [4]: C.__dict__['__doc__']
Out[4]: 'Une classe simple.'

In [5]: C.__dict__['y']
Out[5]: 5

In [6]: a = C()

In [7]: a.__class__
Out[7]: __main__.C

In [8]: a.__doc__ # notation compacte
Out[8]: 'Une classe simple.'

In [9]: a.y # notation compacte
Out[9]: 5
```

Notons également l'instruction `dir()` qui fournit *tous* les noms définis dans l'espace de noms, méthodes et variables membres compris.

7.3 Méthodes

Syntaxe

Une méthode s'écrit comme une fonction *du corps de la classe* avec un premier paramètre `self` obligatoire, où `self` représente l'objet sur lequel la méthode sera appliquée.

Autrement dit `self` est la *référence d'instance.*

```
>>> class C: # x et y : attributs de classe
... x = 23
... y = x + 5
```

```
... def affiche(self): # méthode affiche()
... self.z = 42 # attribut d'instance
... print(C.y) # dans une méthode, on qualifie un attribut de classe,
... print(self.z) # mais pas un attribut d'instance
...

>>> obj = C()

>>> obj.affiche()
28
42
```

7.4 Méthodes spéciales

Python offre un mécanisme qui permet d'enrichir les classes de caractéristiques supplémentaires : les *méthodes spéciales*.

On pourra ainsi :

— à initialiser l'objet instancié ;
— à modifier son affichage ;
— à surcharger ses opérateurs ;
— ...

Par exemple la concaténation des chaînes utilise une syntaxe très intuitive :

```
>>> 'Bonjour' + ' Monde !'
Bonjour Monde !
```

grâce à la redéfinition de l'opérateur + dans la classe chaîne.

Syntaxe

Ces méthodes portent des noms pré-définis, précédés et suivis de deux caractères de soulignement.

7.4.1 L'initialisateur

Lors de l'instanciation d'un objet, la structure de base de l'objet est créée en mémoire, et la méthode `__init__` est automatiquement appelée pour initialiser l'objet. C'est typiquement dans cette méthode spéciale que sont créés les attributs d'instance avec leur valeur initiale.

```
>>> class C:
... def __init__(self, n):
... self.x = n # initialisation de l'attribut d'instance x
...

>>> une_instance = C(42) # paramètre obligatoire, affecté à n

>>> une_instance.x
42
```

C'est une *procédure* automatiquement invoquée lors de l'instanciation : elle ne retourne aucune valeur.

7.4.2 Surcharge des opérateurs

La *surcharge* permet à un opérateur de posséder un sens différent suivant le type de ses opérandes.
Par exemple, l'opérateur + permet :

```
x = 7 + 9 # addition entière
s = 'ab' + 'cd' # concaténation
```

Python possède des méthodes de surcharge pour :

— tous les types (`__call__`, `__str__`, ...) ;
— les nombres (`__add__`, `__div__`, ...) ;
— les séquences (`__len__`, `__iter__`, ...).

Soient deux instances, *obj1* et *obj2*, les méthodes spéciales suivantes permettent d'effectuer les opérations arithmétiques courantes [1] :

Nom	Méthode spéciale	Utilisation
opposé	`__neg__`	`-obj1`
addition	`__add__`	`obj1 + obj2`
soustraction	`__sub__`	`obj1 - obj2`
multiplication	`__mul__`	`obj1 * obj2`
division	`__div__`	`obj1 / obj2`
division entière	`__floordiv__`	`obj1 // obj2`

Voir l'*Abrégé dense, les méthodes spéciales* [cf. p. 133]

7.4.3 Exemple de surcharge

Dans l'exemple suivant nous surchargeons l'opérateur d'addition pour le type `Vecteur2D`.
Nous surchargeons également la méthode spéciale `__str__` utilisée pour l'affichage [2] par `print()`.

```
>>> class Vecteur2D:
... def __init__(self, x0, y0):
... self.x = x0
... self.y = y0
... def __add__(self, second): # addition vectorielle
... return Vecteur2D(self.x + second.x, self.y + second.y)
... def __str__(self): # affichage d'un Vecteur2D
... return "Vecteur({:g}, {:g})".format(self.x, self.y)
...

>>> v1 = Vecteur2D(1.2, 2.3)

>>> v2 = Vecteur2D(3.4, 4.5)

>>> print(v1 + v2)
Vecteur(4.6, 6.8)
```

7.5 Héritage et polymorphisme

Un avantage décisif de la POO est qu'une classe Python peut toujours être spécialisée en une classe fille qui hérite alors de tous les attributs (données et méthodes) de sa super classe. Comme tous les attributs peuvent être redéfinis, une méthode de la classe fille et de la classe mère peut posséder le même nom, mais effectuer des traitements différents (surcharge) et l'objet s'adaptera dynamiquement, dès l'instanciation. En proposant d'utiliser un même nom de méthode pour plusieurs types d'objets différents, le polymorphisme permet une programmation beaucoup plus générique. Le développeur n'a pas à savoir, lorsqu'il programme une méthode, le type précis de l'objet sur lequel la méthode va s'appliquer. Il lui suffit de savoir que cet objet implémentera la méthode.

7.5.1 Héritage et polymorphisme

Définition

✍ L'*héritage* est le mécanisme qui permet de se servir d'une classe préexistante pour en créer une nouvelle qui possédera des fonctionnalités supplémentaires ou différentes.

Le *polymorphisme par dérivation* est la faculté pour deux méthodes (ou plus) portant le même nom, mais appartenant à des classes héritées distinctes d'effectuer un travail différent. Cette propriété est acquise par la technique de la surcharge.

1. Pour plus de détails, consulter la documentation de référence du langage Python (*The Python language reference*) section 3, *Data model*, sous-section 3.3, *Special method names*.
2. Rappelons qu'il existe deux façons d'afficher un résultat : `repr()` et `str()`. La première est « pour la machine », la seconde « pour l'utilisateur ».

7.5.2 Exemple d'héritage et de polymorphisme

Dans l'exemple suivant, la classe `QuadrupedeDebout` hérite de la classe mère `Quadrupede`, et la méthode `piedsAuContactDuSol()` est polymorphe :

```
>>> class Quadrupede:
... def piedsAuContactDuSol(self):
... return 4
...

>>> class QuadrupedeDebout(Quadrupede):
... def piedsAuContactDuSol(self):
... return 2
...

>>> chat = Quadrupede()

>>> chat.piedsAuContactDuSol()
4

>>> homme = QuadrupedeDebout()

>>> homme.piedsAuContactDuSol()
2
```

7.6 Notion de conception orientée objet

Suivant les relations que l'on va établir entre les objets de notre application, on peut concevoir nos classes de deux façons possibles en utilisant l'association ou la dérivation.

Bien sûr, ces deux conceptions peuvent cohabiter, et c'est souvent le cas !

7.6.1 Association

Définition

✍ Une association représente un lien unissant les instances de classe. Elle repose sur la relation « a-un » ou « utilise-un ».

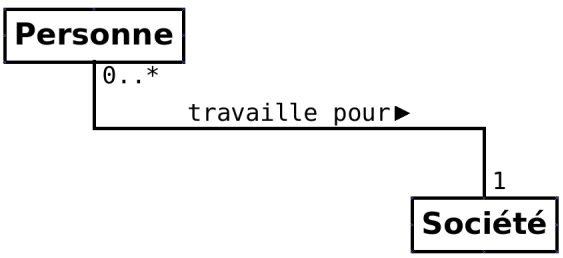


FIGURE 7.2 – Une association peut être étiquetée et avoir des multiplicités.

L'implémentation Python utilisée est généralement l'intégration d'autres objets dans le constructeur de la classe conteneur :

```
class Point:
   def __init__(self, x, y):
      self.px, self.py = x, y

class Segment:
   """Classe conteneur utilisant la classe Point."""
   def __init__(self, x1, y1, x2, y2):
      self.orig = Point(x1, y1)
      self.extrem = Point(x2, y2)

   def __str__(self):
      return ("Segment : [({:g}, {:g}), ({:g}, {:g})]"
         .format(self.orig.px, self.orig.py,
```

```
                self.extrem.px, self.extrem.py))

s = Segment(1.0, 2.0, 3.0, 4.0)
print(s) # Segment : [(1, 2), (3, 4)]
```

Agrégation

Définition

✍ Une agrégation est une association non symétrique entre deux classes (l'*agrégat* et le *composant*).

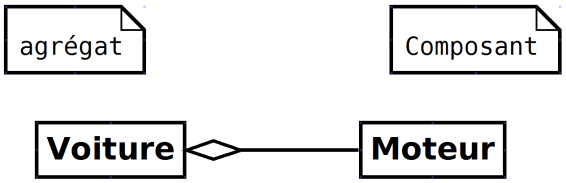


FIGURE 7.3 – Une voiture est un tout qui contient un moteur.

Composition

Définition

✍ Une composition est un type particulier d'agrégation dans laquelle la vie des composants est liée à celle de l'agrégat.

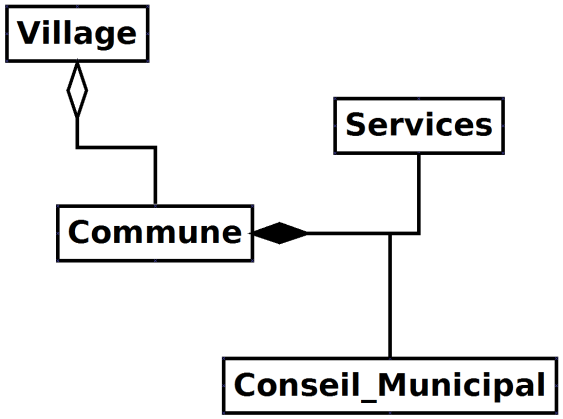


FIGURE 7.4 – On peut mêler les deux types d'association.

La disparition de l'agrégat `Commune` entraine le disparition des composants `Services` et `Conseil_Municipal` alors que `Village` n'en dépend pas.

7.6.2 Dérivation

Définition

✍ La dérivation décrit la création de sous-classes par spécialisation. Elle repose sur la relation « est-un ».

On utilise dans ce cas le mécanisme de l'*héritage*.

L'implémentation Python utilisée est généralement l'appel à l'initialisateur de la classe parente dans l'initialisateur de la classe dérivée (utilisation de la fonction `super()`).

Dans l'exemple suivant, un `Carre` « est-un » `Rectangle` particulier pour lequel on appelle l'initialisateur de la classe mère avec les paramètres `longueur=cote` et `largeur=cote` :

```
>>> class Rectangle:
... def __init__(self, longueur=30, largeur=15):
... self.L, self.l = longueur, largeur
```

```
... self.nom = "rectangle"
... def __str__(self):
... return "nom : {}".format(self.nom)
...

>>> class Carre(Rectangle): # héritage simple
... """Sous-classe spécialisée de la super-classe Rectangle."""
... def __init__(self, cote=20):
... # appel au constructeur de la super-classe de Carre :
... super().__init__(cote, cote)
... self.nom = "carré" # surcharge d'attribut
...

>>> r = Rectangle()

>>> c = Carre()

>>> print(r)
nom : rectangle

>>> print(c)
nom : carré
```

7.7 Un exemple complet

Le script suivant [1] propose un modèle simplifié d'atome et d'ion.
La variable de classe `table` liste les 10 premiers éléments du tableau de Mendeleïev.

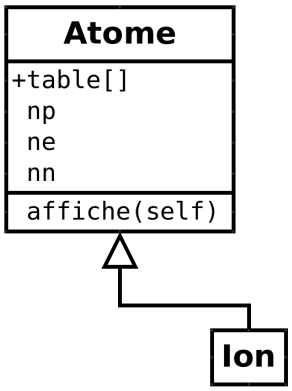


FIGURE 7.5 – Un `Ion` « est-un » `Atome`.

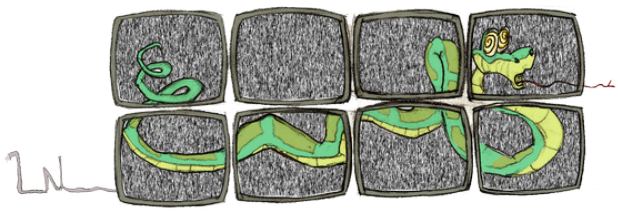
```
class Atome:
   """atomes simplifiés (les 10 premiers éléments)."""

   table = [None, ('hydrogene', 0), ('helium', 2), ('lithium', 4),
            ('beryllium', 5), ('bore', 6), ('carbone', 6), ('azote', 7),
            ('oxygene', 8), ('fluor', 10), ('neon', 10)]
```

1. adapté de [1], p. 127.

```
   def __init__(self, nat):
      "le numéro atomique détermine les nombres de protons, d'électrons et de neutrons"
      self.np, self.ne = nat, nat # nat = numéro atomique
      self.nn = Atome.table[nat][1]

   def affiche(self):
      print()
      print("Nom de l'élément :", Atome.table[self.np][0])
      print("%s protons, %s électrons, %s neutrons" % (self.np, self.ne, self.nn))

class Ion(Atome): # Ion hérite d'Atome
   """Les ions sont des atomes qui ont gagné ou perdu des électrons"""

   def __init__(self, nat, charge):
      "le numéro atomique et la charge électrique déterminent l'ion"
      super().__init__(nat)
      self.ne = self.ne - charge # surcharge
      self.charge = charge

   def affiche(self): # surcharge
      Atome.affiche(self)
      print("Particule électrisée. Charge =", self.charge)

# Programme principal =================================================
a1 = Atome(5)
a2 = Ion(3, 1)
a3 = Ion(8, -2)
a1.affiche()
a2.affiche()
a3.affiche()

"""

Nom de l'élément : bore
5 protons, 5 électrons, 6 neutrons

Nom de l'élément : lithium
3 protons, 2 électrons, 4 neutrons
Particule électrisée. Charge = 1

Nom de l'élément : oxygene
8 protons, 10 électrons, 8 neutrons
Particule électrisée. Charge = -2
"""
```

Chapitre 8

La POO graphique



Très utilisées dans les systèmes d'exploitation et dans les applications, les *interfaces graphiques* sont programmables en Python.

Parmi les différentes bibliothèques graphiques utilisables dans Python (`GTK+`, `wxPython`, `Qt`...), la bibliothèque `tkinter`, issue du langage `tcl/Tk` est installée de base dans toutes les distributions Python. `tkinter` facilite la construction d'interfaces graphiques simples.

Après avoir importé la bibliothèque, la démarche consiste à créer, configurer et positionner les éléments graphiques (widgets) utilisés, à coder les fonctions/méthodes associées aux widgets, puis d'entrer dans une boucle chargée de récupérer et traiter les différents événements pouvant se produire au niveau de l'interface graphique : interactions de l'utilisateur, besoins de mises à jour graphiques, etc.

8.1 Programmes pilotés par des événements

En programmation graphique objet, on remplace le déroulement *séquentiel* du script par une boucle d'événements (☞ Fig. 8.1)

8.2 La bibliothèque `tkinter`

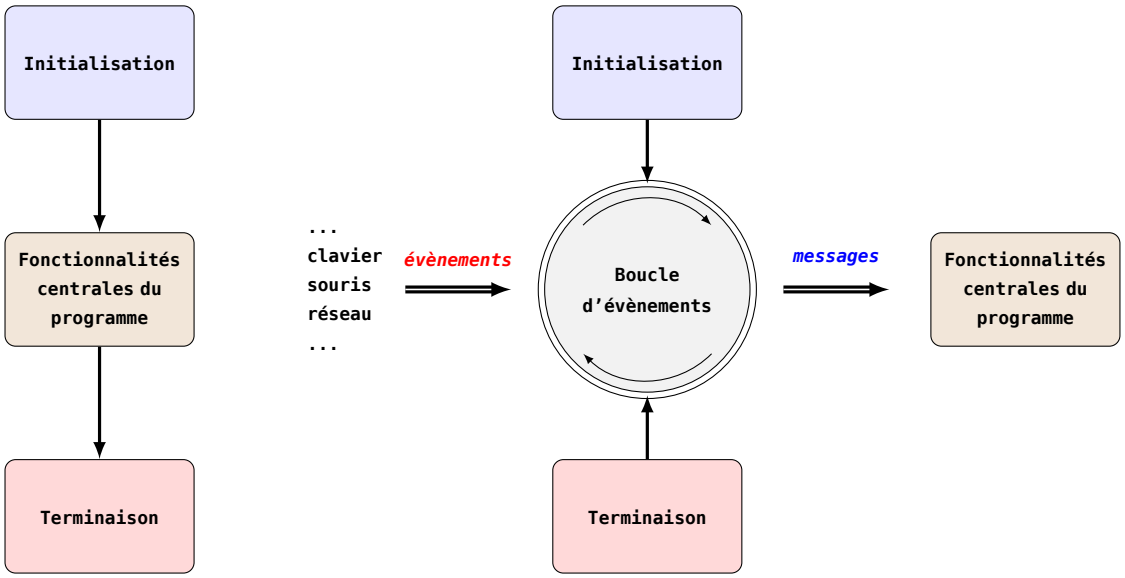
8.2.1 Présentation

C'est une bibliothèque assez simple qui provient de l'extension graphique, `Tk`, du langage `Tcl` [1]. Cette extension a largement essaimé hors de `Tcl/Tk` et on peut l'utiliser en Perl, Python, Ruby, etc. Dans le cas de Python, l'extension a été renommée `tkinter`.

Parallèlement à `Tk`, des extensions ont été développées dont certaines sont utilisées en Python. Par exemple le module standard `Tix` met une quarantaine de widgets à la disposition du développeur.

De son côté, le langage `Tcl/Tk` a largement évolué. La version 8.5 actuelle offre une bibliothèque appelée `Ttk` qui permet d'« habiller » les widgets avec différents thèmes ou styles. Ce module est également disponible à partir de Python 3.1.1.

1. Langage développé en 1988 par John K. Ousterhout de l'Université de Berkeley.



(a) Programmation séquentielle

(b) Programmation pilotée par une boucle d'évènements

FIGURE 8.1 – Deux styles de programmation.

Un exemple `tkinter` simple (☞ Fig. 8.2)

```
import tkinter

# création d'un widget affichant un simple message textuel
widget = tkinter.Label(None, text='Bonjour monde graphique !')
widget.pack() # positionnement du label
widget.mainloop() # lancement de la boucle d'événements
```

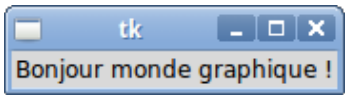


FIGURE 8.2 – Un exemple simple : l'affichage d'un `Label`

8.2.2 Les widgets de `tkinter`

Définition

✍ On appelle widget (mot valise, contraction de *window* et *gadget*) les composants graphiques de base d'une bibliothèque.

Liste des principaux widgets de tkinter :

Tk fenêtre de plus haut niveau ;

Frame contenant pour organiser d'autres widgets ;

Label zone de message ;

Button bouton d'action avec texte ou image ;

Message zone d'affichage multi-lignes ;

Entry zone de saisie ;

Checkbutton bouton à deux états ;

Radiobutton bouton à deux états exclusifs par groupe de boutons ;

Scale glissière à plusieurs positions ;

PhotoImage sert à placer des images (GIF et PPM/PGM) sur des widgets ;

BitmapImage sert à placer des bitmaps (*X11 bitmap data*) sur des widgets ;

Menu menu déroulant associé à un `Menubutton` ;

Menubutton bouton ouvrant un menu d'options ;

Scrollbar ascenseur ;

Listbox liste à sélection pour des textes ;

Text édition de texte simple ou multi-lignes ;

Canvas zone de dessins graphiques ou de photos ;

OptionMenu liste déroulante ;

ScrolledText widget `Text` avec ascenseur ;

PanedWindow interface à onglets ;

LabelFrame contenant pour organiser d'autres widgets, avec un cadre et un titre ;

Spinbox un widget de sélection multiple.

8.2.3 Le positionnement des widgets

`tkinter` possède trois gestionnaires de positionnement :

Le **packer** dimensionne et place chaque widget dans un widget conteneur selon l'espace requis par chacun d'eux ;

Le **gridder** dimensionne et positionne chaque widget dans les cellules d'un tableau d'un widget conteneur ;

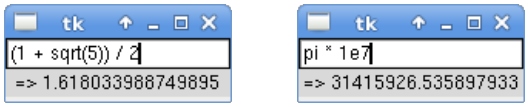
Le **placer** dimensionne et place chaque widget dans un widget conteneur selon l'espace explicitement demandé. C'est un placement absolu (usage peu fréquent).

8.3 Trois exemples

8.3.1 Une calculette

Cette application [1] implémente une calculette simple, mais complète (☞ Fig. 8.3).

```
from tkinter import *
from math import *

def evaluer(event):
   chaine.configure(text = '=> ' + str(eval(entree.get())))

# Programme principal ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
fenetre = Tk()
entree = Entry(fenetre)
entree.bind('<Return>', evaluer)
chaine = Label(fenetre)
entree.pack()
chaine.pack()

fenetre.mainloop()
```



(a) φ, le nombre d'or

(b) Nombre de secondes en un an

FIGURE 8.3 – Une calculette graphique minimale.

1. Adapté de [1], p. 127.

La fonction `evaluer()` est exécutée chaque fois que l'utilisateur appuie sur la touche `Entrée` après avoir entré une expression mathématique. Cette fonction utilise la méthode `configure()` du *widget* `chaine` pour modifier son attribut `text`. Pour cela, `evaluer()` exploite les caractéristiques de la fonction intégrée `eval()` qui analyse et évalue son argument, une chaîne d'expression mathématique. Par exemple :

```
>>> eval('(25 + 8) / 3')
11.0
```

Le programme principal se compose de l'instanciation d'une fenêtre `Tk()` contenant un *widget* `chaine` de type `Label()` et un *widget* `entree` de type `Entry()`. Nous associons l'évènement `<Return>` au *widget* `entree` de façon qu'un appui de la touche `Entrée` déclenche la fonction `evaluer()`. Enfin, après avoir positionné les deux *widget* à l'aide de la méthode `pack()`, on active la boucle d'évènement `mainloop()`.

8.3.2 `tkPhone`, un exemple sans menu

On se propose de créer un script de gestion d'un carnet téléphonique. L'aspect de l'application est illustré ☞ Fig. 8.4

Notion de *callback*

Nous avons vu que la programmation d'interface graphique passe par une boucle principale chargée de traiter les différents évènements qui se produisent.

Cette boucle est généralement gérée directement par la bibliothèque d'interface graphique utilisée, il faut donc pouvoir spécifier à cette bibliothèque quelles fonctions doivent être appelées dans quels cas. Ces fonctions sont nommées des *callbacks* ()ou rappels), car elles sont appelées directement par la bibliothèque d'interface graphique lorsque des évènements spécifiques se produisent.

Conception graphique

La conception graphique va nous aider à choisir les bons widgets.

En premier lieu, il est prudent de commencer par une conception manuelle ! En effet rien ne vaut un papier, un crayon et une gomme pour se faire une idée de l'aspect que l'on veut obtenir.

Dans notre cas on peut concevoir trois zones :

1. Une zone supérieure, dédiée à l'affichage ;
2. Une zone médiane est une liste alphabétique ordonnée ;
3. Une zone inférieure est formée de boutons de gestion de la liste ci-dessus.

Chacune de ces zones est codée par une instance de `Frame()`, positionnée l'une sous l'autre grâce au `packer`, et toutes trois incluses dans une instance de `Tk()` (cf. conception ☞ Fig. 8.4).

Le code de l'interface graphique

Méthodologie : on se propose de séparer le codage de l'interface graphique de celui des *callbacks*. Pour cela on propose d'utiliser l'héritage entre une classe parente chargée de gérer l'aspect graphique et une classe enfant chargée de gérer l'aspect fonctionnel de l'application contenu dans les callbacks. Comme nous l'avons vu précédemment (cf. § 7.6 p. 71), c'est un cas de polymorphisme de dérivation.

Voici donc dans un premier temps le code de l'interface graphique. L'initialisateur crée l'attribut `phoneList`, une liste qu'il remplit avec le contenu du fichier contenant les données (si le fichier n'existe pas, il est créé), crée la fenêtre de base `root` et appelle la méthode `makeWidgets()`.

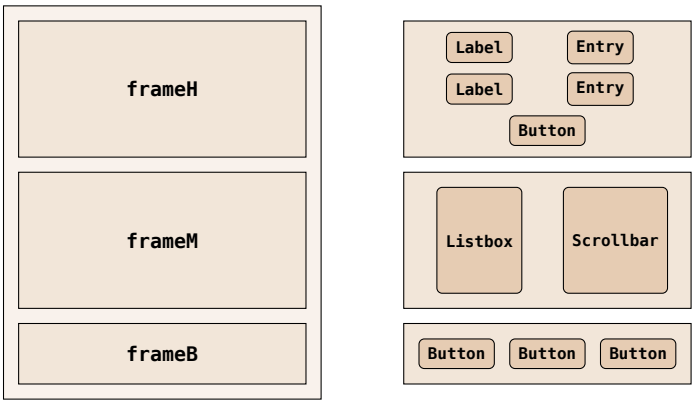
Cette méthode, suit la conception graphique exposée ci-dessus et remplit chacun des trois *frames*.

Les *callbacks* sont vides (instruction `pass` minimale).

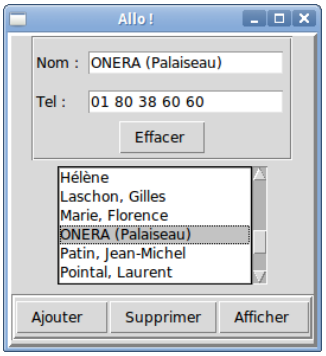
Comme tout bon module, un auto-test permet de vérifier le bon fonctionnement (ici le bon aspect) de l'interface :

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

# Bob Cordeau
# tkPhone_IHM.py
```


> Retour *Table des matières*



(a) Conception générale (b) Détails des `Frame`



(c) L'interface graphique.

FIGURE 8.4 – tkPhone.

```python
import tkinter as tk
from os.path import isfile

# class
class Allo_IHM:
   """IHM de l'application 'répertoire téléphonique'."""
   def __init__(self, fic):
      """Initialisateur/lanceur de la fenêtre de base"""
      self.phoneList = []
      self.fic = fic
      if isfile(self.fic):
         with open(self.fic) as f:
            for line in f:
               self.phoneList.append(line[:-1].split('*'))
      else:
         with open(self.fic, "w", encoding="utf8"):
            pass

      self.phoneList.sort()
      self.root = tk.Tk()
      self.root.title("Allo !")
      self.root.config(relief=tk.RAISED, bd=3)
      self.makeWidgets()
      self.nameEnt.focus()
      self.root.mainloop()

   def makeWidgets(self):
      "Configure et positionne les widgets"
      # frame "saisie" (en haut avec bouton d'effacement)
      frameH = tk.Frame(self.root, relief=tk.GROOVE, bd=2)
      frameH.pack()

      tk.Label(frameH, text="Nom :").grid(row=0, column=0, sticky=tk.W)
      self.nameEnt = tk.Entry(frameH)
      self.nameEnt.grid(row=0, column=1, sticky=tk.W, padx=5, pady=10)
```

```
        tk.Label(frameH, text="Tel :").grid(row=1, column=0, sticky=tk.W)
        self.phoneEnt = tk.Entry(frameH)
        self.phoneEnt.grid(row=1, column=1, sticky=tk.W, padx=5, pady=2)

        b = tk.Button(frameH, text="Effacer ", command=self.clear)
        b.grid(row=2, column=0, columnspan=2, pady=3)

        # frame "liste" (au milieu)
        frameM = tk.Frame(self.root)
        frameM.pack()

        self.scroll = tk.Scrollbar(frameM)
        self.select = tk.Listbox(frameM, yscrollcommand=self.scroll.set, height=6)
        self.scroll.config(command=self.select.yview)
        self.scroll.pack(side=tk.RIGHT, fill=tk.Y, pady=5)
        self.select.pack(side=tk.LEFT, fill=tk.BOTH, expand=1, pady=5)
        ## remplissage de la Listbox
        for i in self.phoneList:
           self.select.insert(tk.END, i[0])
        self.select.bind("<Double-Button-1>", lambda event: self.afficher(event))

        # frame "boutons" (en bas)
        frameB = tk.Frame(self.root, relief=tk.GROOVE, bd=3)
        frameB.pack(pady=3)

        b1 = tk.Button(frameB, text="Ajouter ", command=self.ajouter)
        b2 = tk.Button(frameB, text="Supprimer", command=self.supprimer)
        b3 = tk.Button(frameB, text="Afficher ", command=self.afficher)
        b1.pack(side=tk.LEFT, pady=2)
        b2.pack(side=tk.LEFT, pady=2)
        b3.pack(side=tk.LEFT, pady=2)

    def ajouter(self):
        pass

    def supprimer(self):
        pass

    def afficher(self, event=None):
        pass

    def clear(self):
        pass

# auto-test ------------------------------------------------------------------
if __name__ == '__main__':
    # instancie l'IHM, callbacks inactifs
    app = Allo_IHM("./phones.txt")
```

Le code de l'application

On va maintenant utiliser le module de la façon suivante :

- on importe la classe `Allo_IHM` depuis le module précédent ;
- on crée une classe `Allo` qui en dérive ;
- son initialisateur appelle l'initialisateur de la classe de base pour hériter de toutes ses caractéristiques ;
- il reste à surcharger les callbacks.

Enfin, le script instancie l'application :

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

# Bob Cordeau
# tkPhone.py

# import ---------------------------------------------------------------------
from tkPhone_IHM import Allo_IHM

# définition de classe -------------------------------------------------------
```

```
class Allo(Allo_IHM):
   """Repertoire téléphonique."""

   def __init__(self, fic='phones.txt'):
      "Constructeur de l'IHM."
      super().__init__(fic)


   def ajouter(self):
      # maj de la liste
      ajout = ["", ""]
      ajout[0] = self.nameEnt.get()
      ajout[1] = self.phoneEnt.get()
      if (ajout[0] == "") or (ajout[1] == ""):
         return
      self.phoneList.append(ajout)
      self.phoneList.sort()
      # maj de la listebox
      self.select.delete(0, 'end')
      for i in self.phoneList:
         self.select.insert('end', i[0])
      self.clear()
      self.nameEnt.focus()
      # maj du fichier
      f = open(self.fic, "a")
      f.write("%s*%s\n" % (ajout[0], ajout[1]))
      f.close()


   def supprimer(self):
      self.clear()
      # maj de la liste
      retrait = ["", ""]
      retrait[0], retrait[1] = self.phoneList[int(self.select.curselection()[0])]
      self.phoneList.remove(retrait)
      # maj de la listebox
      self.select.delete(0, 'end')
      for i in self.phoneList:
         self.select.insert('end', i[0])
      # maj du fichier
      f = open(self.fic, "w")
      for i in self.phoneList:
         f.write("%s*%s\n" % (i[0], i[1]))
      f.close()


   def afficher(self, event=None):
      self.clear()
      name, phone = self.phoneList[int(self.select.curselection()[0])]
      self.nameEnt.insert(0, name)
      self.phoneEnt.insert(0, phone)


   def clear(self):
      self.nameEnt.delete(0, 'end')
      self.phoneEnt.delete(0, 'end')


# programme principal ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
app = Allo() # instancie l'application
```

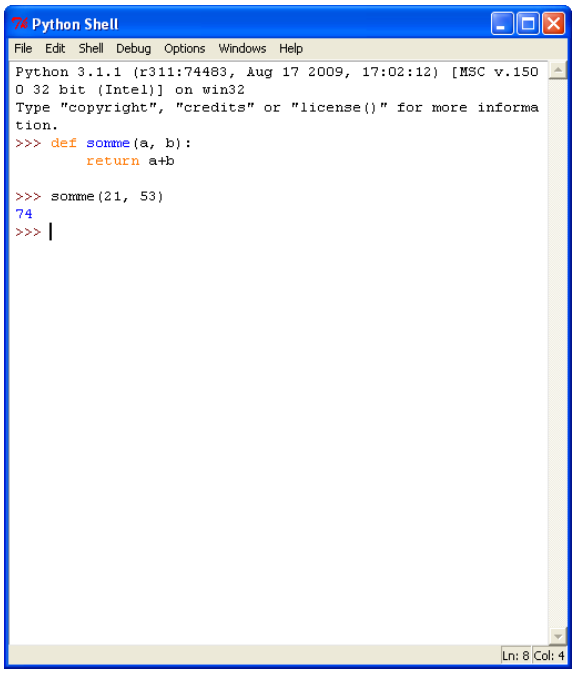
8.3.3 `IDLE`, un exemple avec menu

Généralement les distributions Python comportent l'application `IDLE`, l'interpréteur/éditeur écrit en Python par Guido van Rossum [1]. Cette application se présente sous l'aspect d'une interface graphique complète (☞ Fig. 8.5), avec menu.

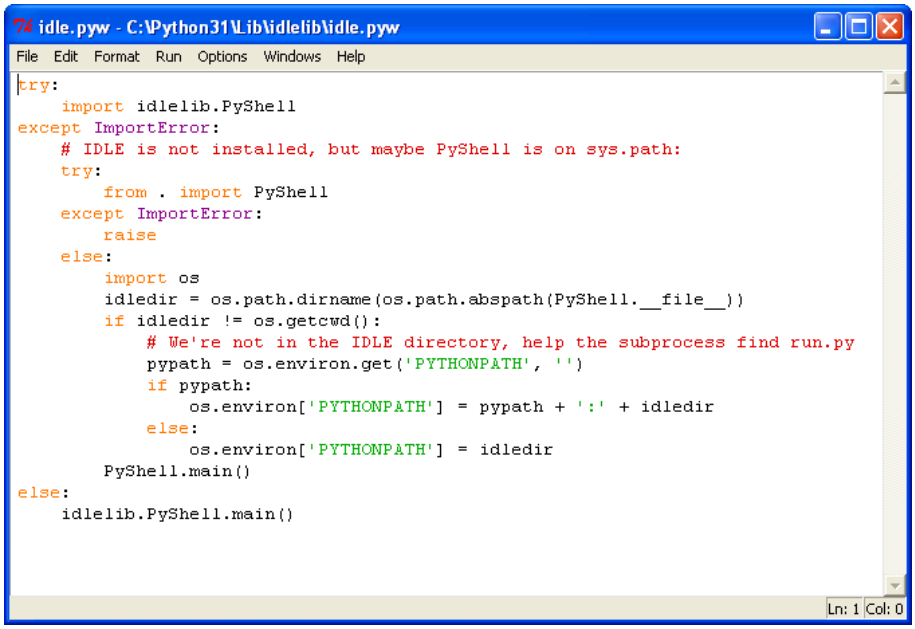
C'est un source Python dont le code est librement disponible [2] et constitue à lui seul un cours complet

1. Dans certaines distributions GNU/Linux, `IDLE` est un package particulier.
2. Mais il est trop volumineux pour être reproduit dans ces notes…

à `tkinter` [1].



(a) L'interpréteur d'IDLE



(b) L'éditeur d'IDLE

FIGURE 8.5 – IDLE.

1. Signalons `IDLEX` une extension à `IDLE` : `http://idlex.sourceforge.net/`.

> Retour *Table des matières*

Chapitre 9

Quelques techniques avancées de programmation

Ce chapitre présente quelques exemples de techniques avancées dans les trois paradigmes que supporte Python, les programmations procédurale, objet et fonctionnelle.

9.1 Techniques procédurales

9.1.1 Le pouvoir de l'introspection

C'est un des atouts de Python. On entend par *introspection* la possibilité d'obtenir des informations sur les objets manipulés par le langage.

La fonction `help()`

On peut tout d'abord utiliser la fonction prédéfinie `help()`.
Cette fonction est auto-documentée :

```
>>> help()

Welcome to Python 3.4 help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.4/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules. To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics". Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help> quit

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)". Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
```

L'interpréteur IPython propose une aide encore plus efficace grâce à son mécanisme d'autocomplétion contextuelle (touche TAB) :

```
In [1]: seq = []

In [2]: seq.
seq.append seq.copy seq.extend seq.insert seq.remove seq.sort
seq.clear seq.count seq.index seq.pop seq.reverse

In [2]: seq.sort?
Type: builtin_function_or_method
String form: <built-in method sort of list object at 0xb5e0c1ac>
Docstring: L.sort(key=None, reverse=False) -> None -- stable sort *IN PLACE*
```

La fonction utilitaire `printInfo()` filtre les méthodes disponibles de son argument ne commençant pas par « _ » et affiche les *docstrings* associées sous une forme plus lisible que `help()` :

```
def printInfo(object):
   """Filtre les méthodes disponibles de <object>."""
   methods = [method for method in dir(object)
      if callable(getattr(object, method)) and not method.startswith('_')]

   for method in methods:
      print(getattr(object, method).__doc__)
```

Par exemple, l'appel `printInfo([])` affiche la documentation :

```
  L.append(object) -- append object to end
  L.count(value) -> integer -- return number of occurrences of value
  L.extend(iterable) -- extend list by appending elements from the iterable
  L.index(value, [start, [stop]]) -> integer -- return first index of value.
  Raises ValueError if the value is not present.
  L.insert(index, object) -- insert object before index
  L.pop([index]) -> item -- remove and return item at index (default last).
  Raises IndexError if list is empty or index is out of range.
  L.remove(value) -- remove first occurrence of value.
  Raises ValueError if the value is not present.
  L.reverse() -- reverse *IN PLACE*
  L.sort(cmp=None, key=None, reverse=False) -- stable sort *IN PLACE*;
  cmp(x, y) -> -1, 0, 1
```

Les fonctions `type()`, `dir()` et `id()`

Ces fonctions fournissent respectivement le type, tous les noms définis dans l'espace de noms et la localisation mémoire (unique) d'un objet :

```
>>> li = [1, 2, 3]

>>> type(li)
<class 'list'>

>>> dir(li)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
 '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__',
 'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

>>> id(li)
3074801164
```

Les fonctions `locals()` et `globals()`

Comme nos l'avons déjà vu [cf. p. 48], ces fonctions retournent respectivement le dictionnaire des noms locaux (globaux) au moment de leur appel.

Le module sys

Ce module fournit nombre d'informations générales, entre autres :

```
>>> import sys

>>> sys.executable
'/usr/bin/python3'

>>> sys.platform
'linux2'

>>> sys.version
'3.2.3 (default, Oct 19 2012, 20:13:42) \n[GCC 4.6.3]'

>>> sys.argv
['']

>>> sys.path
['', '/usr/lib/python3.2', '/usr/lib/python3.2/plat-linux2', '/usr/lib/python3.2/lib-dynload', '/usr/local/lib/python3
    .2/dist-packages', '/usr/lib/python3/dist-packages']

>>> sys.modules
{'reprlib': <module 'reprlib' from '/usr/lib/python3.2/reprlib.py'>, 'heapq': <module 'heapq' from '/usr/lib/python3
    .2/heapq.py'>,
'sre_compile': <module 'sre_compile' from '/usr/lib/python3.2/sre_compile.py'>,
...
```

9.1.2 Gestionnaire de contexte (ou bloc gardé)

Utiliser une ressource dans un bloc de code puis terminer par un appel spécifique pour en fermer proprement l'accès est un motif récurrent. L'instruction `with` gère élégamment ce problème.

Cette syntaxe simplifie le code en assurant que certaines opérations sont exécutées avant et après un bloc d'instructions donné. Illustrons ce mécanisme sur un exemple classique où il importe de fermer le fichier utilisé :

```
# au lieu de ce code :
try:
   fh = open(filename)
   for line in fh:
      process(line)
finally:
   fh.close()


# il est plus simple d'écrire :
with open(filename) as fh:
   for line in fh:
      process(line)
```

9.1.3 Utiliser un dictionnaire pour lancer des fonctions ou des méthodes

L'idée est d'exécuter différentes parties de code en fonction de la valeur d'une variable de contrôle. On peut se servir de cette technique pour implémenter un menu textuel :

```
animaux = []
nombre_de_felins = 0

def gererChat():
   global nombre_de_felins
   print("Miaou")
   animaux.append("félin")
   nombre_de_felins += 1

def gererChien():
   print("Ouah")
   animaux.append("canidé")

def gererOurs():
```

```
   print("Attention au *OUILLE* !")
   animaux.append("plantigrade")

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

dico = {
    "chat" : gererChat,
    "chien" : gererChien,
    "ours" : gererOurs
    }

betes = ["chat", "ours", "chat", "chien"] # une liste d'animaux rencontrés

for bete in betes:
   dico[bete]() # appel de la fonction correspondante

nf = nombre_de_felins
print("nous avons rencontré {} félin(s)".format(nf))
print("Les animaux rencontrés sont : {}".format(', '.join(animaux), end=" ")

"""
Miaou
Attention au *OUILLE* !
Miaou
Ouah
nous avons rencontré 2 félin(s)
Les animaux rencontrés sont : félin, plantigrade, félin, canidé
"""
```

9.1.4 Les fonctions récursives

Définition

✍ Une fonction récursive comporte un appel à elle-même.

Plus précisément, une fonction récursive doit respecter les trois propriétés suivantes :

1. Une fonction récursive contient un cas de base ;
2. Une fonction récursive doit modifier son état pour se ramener au cas de base ;
3. Une fonction récursive doit s'appeler elle-même.

Par exemple, trier un tableau de N éléments par ordre croissant, c'est extraire le plus petit élément puis trier le tableau restant à $N-1$ éléments.

Un algorithme classique très utile est la méthode de Horner qui permet d'évaluer efficacement un polynôme de degré n en une valeur donnée x_0, en remarquant que cette réécriture ne contient plus que n multiplications :

$$p(x_0) = ((\cdots((a_n x_0 + a_{n-1})x_0 + a_{n-2})x_0 + \cdots)x_0 + a_1)x_0 + a_0$$

Voici une implémentation récursive de l'algorithme de Horner dans laquelle le polynôme p est représenté par la liste de ses coefficients $[a_0, \cdots, a_n]$:

```
>>> def horner(p, x):
... if len(p) == 1:
... return p[0]
... p[-2] += x * p[-1]
... return horner(p[:-1], x)
...

>>> horner([5, 0, 2, 1], 2) # x**3 + 2*x**2 + 5, en x = 2
21
```

Les fonction récursives sont souvent utilisées pour traiter les structures arborescentes comme les répertoires dans les systèmes de fichiers des disques durs.

Voici l'exemple d'une fonction qui affiche récursivement les fichiers d'un répertoire fourni en paramètre :

```
#-*- coding: utf8 -*-

from os import listdir
```

```python
from os.path import isdir, join

def listeFichiersPython(repertoire):
   """Affiche récursivement les fichiers Python à partir de <repertoire>."""
   noms = listdir(repertoire)
   for nom in noms:
      if nom in (".", ".."):
         continue
      nom_complet = join(repertoire, nom)
      if isdir(nom_complet):
         listeFichiersPython(nom_complet)
      elif nom.endswith(".py") or nom.endswith(".pyw"):
         print("Fichier Python :", nom_complet)

listeFichiersPython("/home/bob/Tmp")
```

Dans cette définition, on commence par constituer dans la variable `noms` la liste des fichiers et répertoires du répertoire donné en paramètre. Puis, dans une boucle `for`, tant que l'élément examiné est un répertoire, on ré-appelle la fonction sur lui pour descendre dans l'arborescence de fichiers tant que la *condition terminale* (`if nom in (".", ".."):`) est fausse.

Le résultat produit est :

```
Fichier Python : /home/bob/Tmp/parfait_chanceux.py
Fichier Python : /home/bob/Tmp/recursif.py
Fichier Python : /home/bob/Tmp/parfait_chanceux_m.py
Fichier Python : /home/bob/Tmp/verif_m.py
Fichier Python : /home/bob/Tmp/Truc/Machin/tkPhone_IHM.py
Fichier Python : /home/bob/Tmp/Truc/Machin/tkPhone.py
Fichier Python : /home/bob/Tmp/Truc/calculate.py
Fichier Python : /home/bob/Tmp/Truc/tk_variable.py
```

La récursivité terminale

Définition

✍ On dit qu'une fonction `f` est récursive terminale, si tout appel récursif est de la forme `return f(…)`. On parle alors d'*appel terminal*.

Python permet la récursivité mais n'optimise pas automatique les appels terminaux. Il est donc possible [1] d'atteindre la limite arbitraire fixée à 1 000 appels.

On peut pallier cet inconvénient de deux façons. Nous allons illustrer cette stratégie sur un exemple canonique, la factorielle.

La première écriture est celle qui découle directement de la définition de la fonction :

```python
def factorielle(n):
   """Version récursive non terminale."""
   if n == 0:
      return 1
   else:
      return n * factorielle(n-1)
```

On remarque immédiatement (`return n * factorielle(n-1)`) qu'il s'agit d'une fonction récursive **non terminale**. Or une fonction récursive terminale est en théorie plus efficace (mais souvent moins facile à écrire) que son équivalent non terminale : il n'y a qu'une phase de descente et pas de phase de remontée.

La méthode classique pour transformer cette fonction en un appel récursif terminal est d'ajouter un argument d'appel jouant le rôle d'accumulateur. D'où le code :

```python
def factorielleTerm(n, accu=1):
   """Version récursive terminale."""
   if n == 0:
      return accu
   else:
      return factorielleTerm(n-1, n*accu)
```

1. voire *incontournable* si on en croit la loi de Murphy…

La seconde stratégie est d'essayer de transformer l'écriture récursive de la fonction par une écriture itérative. La théorie de la calculabilité montre qu'une telle transformation est toujours possible à partir d'une fonction récursive terminale, ce qu'on appelle l'opération de *dérécursivation*. D'où le code :

```
def factorielleDerec(n, accu=1):
   """Version dérécursivée."""
   while n > 0:
      accu *= n
      n -= 1
   return accu
```

9.1.5 Les listes définies en compréhension

Les listes définies « en compréhension », souvent appelées « compréhension de listes », permettent de générer ou de modifier des collections de données par une écriture lisible, simple et performante.

Cette construction syntaxique se rapproche de la notation utilisée en mathématiques :
$\{x^2 | x \in [2,11[\} \Leftrightarrow$ `[x**2 for x in range(2, 11)]` $\Rightarrow$ `[4, 9, 16, 25, 36, 49, 64, 81, 100]`

Définition

✍ Une liste en compréhension est équivalente à une boucle `for` qui construirait la même liste en utilisant la méthode `append()`.

Les listes en compréhension sont utilisables sous trois formes.

Première forme expression d'une liste simple de valeurs :

```
result1 = [x+1 for x in une_seq]
# a le même effet que :
result2 = []
for x in une_seq:
   result2.append(x+1)
```

Deuxième forme expression d'une liste de valeurs avec filtrage :

```
result3 = [x+1 for x in une_seq if x > 23]
# a le même effet que :
result4 = []
for x in une_seq:
   if x > 23:
      result4.append(x+1)
```

Troisième forme expression d'une combinaison de listes de valeurs :

```
result5 = [x+y for x in une_seq for y in une_autre]
# a le même effet que :
result6 = []
for x in une_seq:
   for y in une_autre:
      result6.append(x+y)
```

Exemples utilisations très *pythoniques* :

```
valeurs_s = ["12", "78", "671"]
# conversion d'une liste de chaînes en liste d'entiers
valeurs_i = [int(i) for i in valeurs_s] # [12, 78, 671]

# calcul de la somme de la liste avec la fonction intégrée sum
print(sum([int(i) for i in valeurs_s])) # 761

# a le même effet que :
s = 0
for i in valeurs_s:
   s = s + int(i)
print(s) # 761

# Initialisation d'une liste 2D
multi_liste = [[0]*2 for ligne in range(3)]
print(multi_liste) # [[0, 0], [0, 0], [0, 0]]
```

Autre exemple :

```
>>> C_deg = range(-20, 41, 5)

>>> F_deg = [(9.0/5)*c + 32 for c in C_deg]

>>> table = [C_deg, F_deg]

>>> for i in range(len(table[0])):
... print(table[0][i], "=>", table[1][i])
...
-20 => -4.0
-15 => 5.0
-10 => 14.0
-5 => 23.0
0 => 32.0
5 => 41.0
10 => 50.0
15 => 59.0
20 => 68.0
25 => 77.0
30 => 86.0
35 => 95.0
40 => 104.0
```

9.1.6 Les dictionnaires définis en compréhension

Comme pour les listes, on peut définir des dictionnaires en compréhension :

```
>>> {n : x**2 for n, x in enumerate(range(5))}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Notons l'utilisation des accolades et du deux-points caractéristiques de la syntaxe du dictionnaire.

9.1.7 Les ensembles définis en compréhension

De même, on peut définir des ensembles en compréhension :

```
>>> {n for n in range(5)}
set([0, 1, 2, 3, 4])
```

Dans ce cas les accolades sont caractéristiques de la syntaxe de l'ensemble.

9.1.8 Les générateurs et les expressions génératrices

Les générateurs

Définition

✍ Un générateur est une fonction [1] qui mémorise son état au moment de retourner une valeur.
La transmission d'une valeur s'effectue en utilisant le mot clé `yield`.

Les générateurs fournissent un moyen de générer des *exécutions paresseuses* [2], ce qui signifie qu'elles ne calculent que les valeurs réellement demandées. Ceci peut s'avérer beaucoup plus efficace (en termes de mémoire) que le calcul, par exemple, d'une énorme liste en une seule fois.

Techniquement, un générateur fonctionne en deux temps. D'abord, au lieu de retourner une valeur avec le mot clé `return`, la fonction qui doit servir de générateur produit une valeur *et se met en pause* dès qu'elle rencontre le mot clé `yield`.

Ensuite, à l'utilisation du générateur, le corps de la fonction est exécuté lors des appels explicites à la méthode `next()` ou implicites dans une boucle `for`.

Voici un exemple de compteur d'entiers qui décrémente l'argument du générateur jusqu'à zéro :

1. ou plutôt une *procédure* car un générateur ne peut retourner que la valeur `None`.
2. appelées aussi *appels par nécessité* ou *évaluations retardées*.

```
>>> def countDown(n):
   """génère un décompteur à partir de <n>.

   Un générateur ne peut retourner que None.
   """
   print('Mise à feu :')
   while n > 0:
      yield n
      n = n - 1

for val in countDown(5):
   print(val, end=" ")
...
Mise à feu :
5 4 3 2 1
```

Remarquons que le premier appel au générateur produit trois effets :

1. Création de l'objet générateur par l'appel à la fonction `countDown()` ;
2. Initialisation : la fonction `countDown()` se déroule séquentiellement (notons l'affichage `Mise à feu`) ;
3. Arrivée à l'instruction `yield n`, la fonction retourne la valeur de `n` puis se met en pause.

Les appels suivants déclenchent la reprise de l'exécution de la fonction jusqu'au prochain appel de l'instruction `yield n`. Le mécanisme itère jusqu'au retour de la fonction quand `n` vaut `0`.

Les expressions génératrices

Syntaxe

✎ Une expression génératrice possède une syntaxe presque identique à celle des listes en compréhension à la différence qu'une expression génératrice est entourée de parenthèses.

Utilisation

Les expressions génératrices (souvent appelée « genexp ») sont aux générateurs ce que les listes en compréhension sont aux fonctions. Bien qu'il soit transparent, le mécanisme du `yield` vu ci-dessus est encore en action.

Par exemple la liste en compréhension suivante génère la création d'un million de valeurs en mémoire *avant* de commencer la boucle :

```
for i in [x**2 for x in range(1000000)]:
```

Alors que dans l'expression suivante, la boucle commence *immédiatement* et ne génère les valeurs qu'au fur et à mesure des demandes :

```
for i in (x**2 for x in range(1000000)):
```

Voici un autre exemple : une *genexp* de recherche d'un motif dans un fichier, analogue à un `grep` simplifié [1] :

```
>>> with open('.bash_aliases') as f:
... file = f.readlines()
...

>>> lines = (line.strip() for line in file if "alias" in line)

>>> for line in lines:
... line
...
'# some more aliases'
"alias ll='ls -alF'"
"alias la='ls -A'"
"alias l='ls -CF'"
"alias c='clear'"
"alias p='~/ShellScripts/a2ps_UTF8'"
```

1. Célèbre utilitaire de recherche d'un motif dans le monde Unix.

```
"alias ipq='ipython qtconsole &'"
"alias nb='ipython notebook &'"
"alias qt='ipython qtconsole &'"
'alias kie=\'find ~/Phil/ -name "*.tex" | xargs grep\''
```

9.1.9 Fonctions incluses et fermetures

La syntaxe de définition des fonctions en Python permet tout à fait d'*emboîter* leur définition. Voici une fonction incluse simple :

```
>>> def print_msg(msg):
   """Fonction externe."""
   def printer():
      """Fonction incluse."""
      print(msg)
   printer() # appel à la fonction incluse
...

>>> print_msg('Hello')
Hello
```

Le subtil changement suivant définit une fermeture [1] :

```
>>> def print_msg(msg):
   """Fonction externe."""
   def printer():
      """Fonction incluse."""
      print(msg)
   return printer # retourne la fonction incluse
...

>>> fct = print_msg('Hello') # fct est une fonction

>>> fct()
Hello
```

Définition

Une fermeture réunit ces trois critères :

1. C'est une fonction qui doit comporter une fonction incluse
2. La fonction incluse doit utiliser une valeur définie dans la fonction externe
3. La fonction externe doit retourner la fonction incluse

Les fermetures évitent l'utilisation des variables globales. Quand une classe comporte peu de méthodes (une méthode dans la plupart des cas), la fermeture est une alternative élégante.

Fonction fabrique

Le besoin est de créer des instances de fonctions ou de classes suivant certaines conditions. Un bon moyen est d'implanter une création d'objet souple en utilisant une fonction *fabrique*.

Idiome de la fonction fabrique renvoyant une fermeture :

```
>>> def creer_plus(ajout):
... """Fonction 'fabrique'."""
... def plus(increment):
... """Fonction 'fermeture' : utilise des noms locaux à creer_plus()."""
... return increment + ajout
... return plus
...

>>> p = creer_plus(23)

>>> q = creer_plus(42)

>>> print("p(100) =", p(100))
```

1. en anglais *closure*.

```
('p(100) =', 123)

>>> print("q(100) =", q(100))
('q(100) =', 142)
```

Fonction fabrique renvoyant une classe :

```
>>> class CasNormal:
... def uneMethode(self):
... print("normal")
...

>>> class CasSpecial:
... def uneMethode(self):
... print("spécial")
...

>>> def casQuiConvient(estNormal=True):
... """Fonction fabrique renvoyant une classe."""
... if estNormal:
... return CasNormal()
... else:
... return CasSpecial()
...

>>> une_instance = casQuiConvient()

>>> une_instance.uneMethode()
normal

>>> une_instance = casQuiConvient(False)

>>> une_instance.uneMethode()
spécial
```

9.1.10 Les décorateurs

Les décorateurs permettent d'encapsuler un appel et donc d'effectuer des pré- ou des post-traitements lors de l'appel d'une fonction, d'une méthode ou d'une classe.

Syntaxe

Soit `deco()` un décorateur. Pour « décorer » une fonction on écrit :

```
def deco():
   ...

@deco
def fonction(arg1, arg2, ...):
   pass
```

Une fonction peut être multi-décorée :

```
def f1():
  ...

def f2():
  ...

def f3():
  ...

@f1 @f2 @f3
def g():
   pass
```

Voici un exemple simple :

```
def unDecorateur(f):
   cptr = 0
   def _interne(*args, **kwargs):
      nonlocal cptr
      cptr = cptr + 1
      print("Fonction décorée :", f.__name__, ". Appel numéro :", cptr)
      return f(*args, **kwargs)

   return _interne

@unDecorateur
def uneFonction(a, b):
   return a + b

def autreFonction(a, b):
   return a + b


# programme principal ============================================
## utilisation d'un décorateur
print(uneFonction(1, 2))
## utilisation de la composition de fonction
autreFonction = unDecorateur(autreFonction)
print(autreFonction(1, 2))

print(uneFonction(3, 4))
print(autreFonction(6, 7))
"""
Fonction décorée : uneFonction. Appel numéro : 1
3
Fonction décorée : autreFonction. Appel numéro : 1
3
Fonction décorée : uneFonction. Appel numéro : 2
7
Fonction décorée : autreFonction. Appel numéro : 2
13
"""
```

9.2 Techniques objets

Comme nous l'avons vu lors du chapitre précédent, Python est un langage complètement objet. Tous les types de base ou dérivés sont en réalité des types abstraits de données implémentés sous forme de classe.

9.2.1 Les *Functors*

En Python un objet fonction ou *functor* est une référence à tout objet « appelable » [1] : fonction, fonction anonyme lambda [2], méthode, classe. La fonction prédéfinie `callable()` permet de tester cette propriété :

```
>>> def maFonction():
... print('Ceci est "appelable"')
...

>>> callable(maFonction)
True

>>> chaine = 'Une chaîne'

>>> callable(chaine)
False
```

Il est possible de transformer les instances d'une classe en *functor* si la méthode spéciale `__call__()` est définie dans la classe :

1. *callable* en anglais.
2. Cette notion sera développée ultérieurement [cf. p. 98]

```
>>> class A:
... def __init__(self):
... self.historique = []
... def __call__(self, a, b):
... self.historique.append((a, b))
... return a + b
...

>>> a = A()

>>> a(1, 2)
3

>>> a(3, 4)
7

>>> a(5, 6)
11

>>> a.historique
[(1, 2), (3, 4), (5, 6)]
```

9.2.2 Les accesseurs

Le problème de l'encapsulation

Dans le paradigme objet, la *visibilités* d'un objet est **privée**, les autres objets n'ont pas le droit de le consulter ou de le modifier.

FIGURE 9.1 – En Python, tous les attributs (données, méthodes) sont publics !

On peut néanmoins remédier à cet état de fait.

L'état d'un objet est géré par des *accesseurs*. On distingue habituellement le *getter* pour la lecture, le *setter* pour la modification et le *deleter* pour la suppression.

Lorsqu'un nom est préfixé par un caractère souligné, il est conventionnellement réservé à un usage interne (privé). Mais Python n'oblige à rien [1], c'est au développeur de respecter la convention !

On peut également préfixer un nom par deux caractères souligné, ce qui permet d'éviter les collisions de noms dans le cas où un même attribut serait défini dans une sous-classe. Ce renommage [2] a comme effet de bord de rendre l'accès à cet attribut plus difficile de l'extérieur de la classe qui le définit, quoique cette protection reste déclarative et n'offre pas une sécurité absolue.

La solution `property`

Le principe de l'encapsulation est mis en œuvre par la notion de propriété.

Définition

✍ Une propriété (`property`) est un attribut d'instance possédant des fonctionnalités spéciales.

Les *property* utilisent la syntaxe des décorateurs. Bien remarquer que, dans l'exemple suivant, on utilise `artist` et `title` comme des attributs simples :

```
class Oeuvre:

   def __init__(self, artist, title):
      self.__artist = artist
      self.__title = title

   @property
   def artist(self):
```

1. Slogan des développeurs Python : « We're all consenting adults here » (nous sommes entre adultes consentants).
2. Le *name mangling*.

```
        return self.__artist

    @artist.setter
    def artist(self, artist):
        self.__artist = artist

    @property
    def title(self):
        return self.__title

    @title.setter
    def title(self, title):
        self.__title = title

    def __str__(self):
        return "{:s} : '{:s}' de {:s}".format(self.__class__.__name__,
                                    self.__title, self.__artist)

if __name__ == '__main__':
    items = []

    items.append(Oeuvre('François Rabelais', 'Gargantua'))
    items.append(Oeuvre('Charles Baudelaire', 'Les Fleurs du mal'))

    for item in items:
        print("{} : '{}'".format(item.artist, item.title))
"""
François Rabelais : 'Gargantua'
Charles Baudelaire : 'Les Fleurs du mal'
"""
```

Un autre exemple : la classe `Cercle`

Schéma de conception : nous allons tout d'abord définir une classe `Point` que nous utiliserons comme classe de base de la classe `Cercle`.

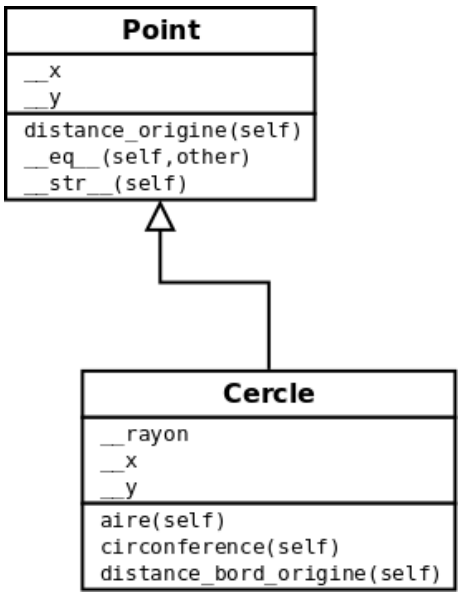


FIGURE 9.2 – Conception UML de la classe `Cercle`.

Voici le code de la classe `Point` :

```
class Point:

    def __init__(self, x=0, y=0):
        self.__x, self.__y = x, y

    @property
```

```
    def distance_origine(self):
        return math.hypot(self.__x, self.__y)

    def __eq__(self, other):
        return self.__x == other.__x and self.__y == other.__y

    def __str__(self):
        return "({}, {})".format(self.__x, self.__y)
```

L'utilisation de `property` permet un accès en *lecture seule* au résultat de la méthode `distance_origine()` considérée alors comme un simple attribut (car on l'utilise sans parenthèse) :

```
p1, p2 = Point(), Point(3, 4)
print(p1 == p2) # False
print(p2, p2.distance_origine) # (3, 4) 5.0
```

De nouveau, les méthodes renvoyant un simple flottant seront utilisées comme des attributs grâce à `property` :

```
class Cercle(Point):
    def __init__(self, rayon, x=0, y=0):
        super().__init__(x, y)
        self.__rayon = rayon

    @property
    def aire(self): return math.pi * (self.__rayon ** 2)

    @property
    def circonference(self): return 2 * math.pi * self.__rayon

    @property
    def distance_bord_origine(self):
        return abs(self.distance_origine - self.__rayon)
```

Voici la syntaxe permettant d'utiliser la méthode rayon comme un attribut en *lecture-écriture*. Remarquez que la méthode `rayon()` retourne l'attribut protégé : `__rayon` qui sera modifié par le *setter* (la méthode modificatrice) :

```
    @property
    def rayon(self):
        return self.__rayon

    @rayon.setter
    def rayon(self, rayon):
        assert rayon > 0, "rayon strictement positif"
        self.__rayon = rayon
```

Exemple d'utilisation des instances de `Cercle` :

```
    def __eq__(self, other):
        return (self.rayon == other.rayon
                        and super().__eq__(other))

    def __str__(self):
        return ("{0.__class__.__name__}({0.rayon!s}, {0.x!s}, "
              "{0.y!s})".format(self))

if __name__ == "__main__":
    c1 = Cercle(2, 3, 4)
    print(c1, c1.aire, c1.circonference)
    # Cercle(2, 3, 4) 12.5663706144 12.5663706144
    print(c1.distance_bord_origine, c1.rayon) # 3.0 2
    c1.rayon = 1 # modification du rayon
    print(c1.distance_bord_origine, c1.rayon) # 4.0 1
```

9.2.3 Le *duck typing* et les annotations

Il existe un style de programmation très pythonique appelé *duck typing* :

« S'il marche comme un canard et cancane comme un canard, alors c'est un canard ! ».

Cela signifie que Python ne s'intéresse qu'au *comportement* des objets. Si des objets offrent la même API (interface de programmation), l'utilisateur peut employer les mêmes méthodes :

Continuons l'exemple de la classe `Oeuvre` :

```
from property3 import Oeuvre

class Peinture(Oeuvre):
   """Peinture hérite d'Oeuvre."""
   def __init__(self, artist, title):
      super().__init__(artist, title)

class Sculpture(Oeuvre):
   """Sculpture hérite d'Oeuvre."""
   def __init__(self, artist, title):
      super().__init__(artist, title)

class Infos:
   """
   Exemple de 'duck typing'.

   Utilise les méthodes artist() et title() d'une classe quelconque.
   """
   def __init__(self, artist, title):
      self.__artist = artist
      self.__title = title

   def artist(self):
      return self.__artist

   def title(self):
      return self.__title

if __name__ == '__main__':
   items = []

   items.append(Oeuvre('François Rabelais', 'Gargantua'))
   items.append(Peinture('Claude Monet', 'Impression'))
   items.append(Sculpture('Auguste Rodin', 'Le Baiser'))

   for item in items:
      print(item)
"""
Oeuvre : 'Gargantua' de François Rabelais
Peinture : 'Impression' de Claude Monet
Sculpture : 'Le Baiser' de Auguste Rodin
"""
```

Python est un langage très souple : la nature des données est dynamiquement découverte à l'exécution, ce qui offre plusieurs avantages :

— on peut utiliser le « duck typing » : le code est court et clair ;
— le langage est très facile à apprendre.

Mais quand le programme reçoit un type inattendu, il s'arrête en erreur à l'exécution. C'est bien là la différence avec les langages à typage statique où toute erreur de type est décelée dès la compilation. Les *annotations* ont été pensées pour pallier ce problème.

Syntaxe

✎ Exemple d'annotation :

```
def pgcd(a:int, b:int) -> int:
   while b:
      a,b = b, a%b
   return a
```

Les annotations permettent de fournir des informations supplémentaires. Or, c'est important, ces informations *optionnelles* n'ont aucun impact sur l'exécution du code, mais des outils tierces parties [1] pourront les utiliser pour par exemple :

1. En particulier le projet mypy auquel GvR participe activement.

— faire de la vérification statique de type utile dans certains cas (gros projets, nombreux développeurs, aide à la documentation et au débogage complexe…) ;
— fournir une aide aux éditeurs de codes ;
— offrir un complément à la documentation des *docstrings* ;
— …

9.3 Techniques fonctionnelles

9.3.1 Directive `lambda`

Issue de langages fonctionnels (comme OCaml), la directive `lambda` permet de définir un objet *fonction anonyme* dont le bloc d'instructions est limité à une expression dont l'évaluation fournit la valeur de retour de la fonction.

Syntaxe

✎ `lambda [parameters]: expression`

Par exemple cette fonction retourne « s » si son argument est différent de 1, une chaîne vide sinon :

```
>>> s = lambda x: "" if x == 1 else "s"

>>> s(3)
's'

>>> s(1)
''
```

Le code suivant reprend l'exemple des fonctions composées (cf. § 5.2 p. 45), mais en utilisant une directive `lambda` :

```
>>> def f(x):
... return 2*x + 3
...

>>> def g(x):
... return x**2
...

>>> def compose(f1, f2):
... return lambda x: f2(f1(x))
...

>>> print(compose(f, g)(2.0)) # g[f(x)] = (2*x + 3)**2
49.0

>>> print(compose(g, f)(2.0)) # f[g(x)] = 2*x**2 + 3
11.0
```

9.3.2 Les fonctions `map`, `filter` et `reduce`

La programmation fonctionnelle est un paradigme de programmation qui considère le calcul en tant qu'évaluation de fonctions mathématiques. Elle souligne l'application des fonctions, contrairement au modèle de programmation impérative qui met en avant les changements d'état [1]. Elle repose sur trois concepts : *mapping*, *filtering* et *reducing* qui sont implémentés en Python par trois fonctions : `map()`, `filter()` et `reduce()`.

La fonction `map()` :

`map()` applique une fonction à chaque élément d'une séquence et retourne un itérateur :

```
>>> map(lambda x:x, range(10))
<map object at 0x7f3a80104f50>

>>> list(map(lambda x:x, range(10)))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

1. cf. Wikipedia

On remarque que `map()` peut être remplacée par un générateur en compréhension.

La fonction `filter()` :

`filter()` construit et renvoie un itérateur sur une liste qui contient tous les éléments de la séquence initiale répondant au critère : `function(element) == True` :

```
>>> list(filter(lambda x: x > 4, range(10)))
[5, 6, 7, 8, 9]
```

De même `filter()` peut être remplacée par un générateur en compréhension.

La fonction `reduce()` :

`reduce()` est une fonction du module `functools`. Elle applique de façon cumulative une fonction de deux arguments aux éléments d'une séquence, de gauche à droite, de façon à réduire cette séquence à une seule valeur qu'elle renvoie :

```
>>> def somme(x, y):
... print x, '+', y
... return x + y
...

>>> reduce(somme, [1, 2, 3, 4])
1 + 2
3 + 3
6 + 4
10

>>> sum([1, 2, 3, 4])
10
```

La fonction `reduce()` peut être remplacée par une des fonctions suivantes : `all()`, `any()`, `max()`, `min()` ou `sum()`.

9.3.3 Les applications partielles de fonctions

Issue de la programmation fonctionnelle, une PFA (application partielle de fonction) de n paramètres prend le premier argument comme paramètre fixe et retourne un objet fonction (ou instance) utilisant les n-1 arguments restants.

Les PFA sont utiles dans les fonctions de calcul comportant de nombreux paramètres. On peut en fixer certains et ne faire varier que ceux sur lesquels on veut agir :

```
>>> from functools import partial

>>> def f(m, c, d, u):
... return 1000*m + 100*c + 10*d + u
...

>>> f(1, 2, 3, 4)
1234

>>> g = partial(f, 1, 2, 3)

>>> g(4)
1234

>>> h = partial(f, 1, 2)

>>> h(3, 4)
1234
```

Elles sont aussi utiles pour fournir des modèles partiels de widgets, qui ont souvent de nombreux paramètres. Dans l'exemple suivant, on redéfinit la classe `Button` en fixant certains de ses attributs (qui peuvent toujours être surchargés) :

FIGURE 9.3 – PFA appliquée à un *widget*

```
from functools import partial
import tkinter as tk

root = tk.Tk()
# instanciation partielle de classe :
MonBouton = partial(tk.Button, root, fg='blue', bg='yellow')
MonBouton(text="Bouton 1").pack()
MonBouton(text="Bouton 2").pack()
MonBouton(text="QUITTER", bg='orange', fg='black',
command=root.quit).pack(fill=tk.X, expand=True)
root.title("PFA !")
root.mainloop()
```

Ce résultat est illustré ☞ Fig. 9.3.

9.4 La persistance et la sérialisation

Définition

✍ La **persistance** consiste à sauvegarder des données afin qu'elles survivent à l'arrêt de l'application.
On peut distinguer deux étapes :
— la sérialisation et la désérialisation ;
— le stockage et l'accès.
La **sérialisation** est le processus de conversion d'un ensemble d'objets en un flux d'octets.
La **désérialisation** est le processus inverse qui recrée les données d'origine.
Le **stockage** utilise soit des fichiers, soit des bases de données.

Examinons des exemples simples.

9.4.1 Sérialisation avec `pickle` et `json`

Le module `pickle`

L'intérêt du module `pickle` est sa simplicité. Par contre, ce n'est pas un format utilisé dans d'autres langages, il n'est utile que tant que l'on reste dans le monde Python.

On peut utiliser une chaîne pour sérialiser, mais l'usage le plus commun est d'utiliser un fichier ouvert en mode binaire (contrairement au mode texte que l'on a déjà vu cf. § 4.7 p. 36), avec le mode `"wb"`. Par exemple pour un dictionnaire :

```
import pickle

favorite_color = {"lion": "jaune", "fourmi": "noire", "caméléon": "variable"}
# stocke ses données dans un fichier
pickle.dump(favorite_color, open("save.p", "wb"))

# retrouver ses données : pickle recrée un dictionnaire
dico = pickle.load(open("save.p", "rb"))
print(dico)
```

La lecture du fichier `save.p` produit :

```
{'fourmi': 'noire', 'lion': 'jaune', 'caméléon': 'variable'}
```

Le module `json`

Le module `json` permet d'encoder et de décoder des informations au format `json`[1]. C'est un format d'échange très utile, implémenté dans un grand nombre de langages, plus lisible que `XML` mais moins puissant.

On utilise la même syntaxe qu'avec `pickle`, à savoir `dumps()` et `loads()` pour une chaîne, `dump()` et `load()` pour un fichier, mais cette fois, un fichier textuel :

```
import json

# encodage dans un fichier
with open("json_tst", "w") as f:
   json.dump(['foo', {'bar':('baz', None, 1.0, 2)}], f)

# décodage
with open("json_tst") as f:
   print(json.load(f))
```

La lecture du fichier `json_tst` produit :

```
['foo', {'bar': ['baz', None, 1.0, 2]}]
```

9.4.2 Stockage avec `sqlite3`

Le module `sqlite3` est une bibliothèque écrite en C qui implémente une base de données relationnelle légère qui utilise des fichiers (ou même la mémoire).

Cette utilisation convient bien à de petits volumes de données et permet de maquetter le passage à des bases de données plus importantes.

Exemple de stockage d'une table :

```
import sqlite3

conn = sqlite3.connect('/home/bob/Tmp/tel_bd') # création du connecteur
with conn as c: # création du curseur
   # création de la table
   c.execute("""create table tel (nom text, prenom text, numero integer)""")
   # insertion d'une ligne de données
   c.execute("""insert into tel values ('Barthod', 'Stéphane', '86451')""")
   c.execute("""insert into tel values ('Bourrely', 'Jean', '86617')""")
   c.execute("""insert into tel values ('Brunel', 'Michaël', '86619')""")
   c.execute("""insert into tel values ('Canal', 'Bruno', '86604')""")
   c.execute("""insert into tel values ('Cordeau', 'Bob', '86624')""")
   c.execute("""insert into tel values ('Donath', 'Therèse', '86600')""")
   c.execute("""insert into tel values ('Marie', 'Florence', '86602')""")
```

Le fichier `tel_bd` produit peut être visualisé par le programme *SQLite database browser* (☞ Fig. 9.4).

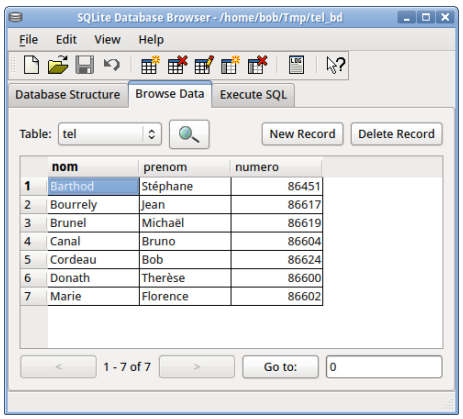


FIGURE 9.4 – Visualisation d'un fichier de base de données sqlite3

1. *JavaScript Object Notation.*

9.5 Les tests

Dès lors qu'un programme dépasse le stade du petit script, le problème des erreurs et donc des tests se pose inévitablement [1].

Définition

✍ Un test consiste à appeler la fonctionnalité spécifiée dans la documentation, avec un scénario qui correspond à un cas d'utilisation, et à vérifier que cette fonctionnalité se comporte comme prévu.

9.5.1 Tests unitaires et tests fonctionnels

On distingue deux familles de test :

Tests unitaires : validations isolées du fonctionnement d'une classe, d'une méthode ou d'une fonction ;

Tests fonctionnels : prennent l'application complète comme une « boîte noire » et la manipulent comme le ferait l'utilisateur final. Ces tests doivent passer par les mêmes interfaces que celles fournies aux utilisateurs, c'est pourquoi ils sont spécifiques à la nature de l'application et plus délicats à mettre en œuvre.

Dans cette introduction, nous nous limiterons à une courte présentation des tests unitaires.

9.5.2 Module `unittest`

Le module standard `unittest` fournit l'outil `PyUnit`, outil que l'on retrouve dans d'autres langages : `JUnit` (Java), `NUnit` (.Net), `JSUnit` (JavaScript), tous dérivés d'un outil initialement développé pour le langage SmallTalk : `SUnit`.

Par convention, chaque module est associé à un module de tests unitaires, placé dans un répertoire `tests` du paquet. Par exemple, un module nommé `calculs.py` aura un module de tests nommé `tests/test_calculs.py`.

`PyUnit` propose une classe de base, `TestCase`. Chaque méthode implémentée dans une classe dérivée de `TestCase`, et préfixée de `test_`, sera considérée comme un test unitaire [2] :

```python
"""Module de calculs."""

# fonctions
def moyenne(*args):
    """Renvoie la moyenne."""
    length = len(args)
    sum = 0
    for arg in args:
        sum += arg
    return sum/length

def division(a, b):
    """Renvoie la division."""
    return a/b
```

```python
"""Module de test du module de calculs."""

# import ------------------------------------------------------------------
import sys
import unittest
from os.path import abspath, dirname
# on enrichit le path pour ajouter le répertoire absolu du source à tester :
sys.path.insert(0, dirname(dirname((abspath(__file__)))))
from calculs import moyenne, division

# définition de classe et de fonction -------------------------------------
class CalculTest(unittest.TestCase):

    def test_moyenne(self):
        self.assertEquals(moyenne(1, 2, 3), 2)
```

1. cf. la loi de Murphy.
2. cf. [7], p. 127.

```
      self.assertEquals(moyenne(2, 4, 6), 4)

   def test_division(self):
      self.assertEquals(division(10, 5), 2)
      self.assertRaises(ZeroDivisionError, division, 10, 0)

def test_suite():
   tests = [unittest.makeSuite(CalculTest)]
   return unittest.TestSuite(tests)

# auto-test ==========================================================
if __name__ == '__main__':
   unittest.main()
```

L'exécution du test produit :

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Pour effectuer une « campagne de tests », il reste à créer un script qui :
- recherche tous les modules de test : leurs noms commencent par `test_` et ils sont contenus dans un répertoire `tests` ;
- récupère la `suite`, renvoyée par la fonction globale `test_suite` ;
- crée une suite de suites et lance la campagne.

9.6 La documentation des sources

Durant la vie d'un projet, on distingue plusieurs types de documentation :
- les documents de spécification (ensemble explicite d'exigences à satisfaire) ;
- les documents techniques attachés au code ;
- les manuels d'utilisation et autres documents de haut niveau.

Les documents techniques évoluent au rythme du code et peuvent donc être traités comme lui : ils doivent pouvoir être lus et manipulés avec un simple éditeur de texte.

Il existe deux outils majeurs pour concevoir des documents pour les applications Python :

reStructuredText (ou reST) : un format enrichi ;

les doctests : compatibles avec le format `reST`. Ils permettent de combiner les textes applicatifs avec les tests.

9.6.1 Le format reST

Le format `reStructuredText`, communément appelé `reST` est un système de balises utilisé pour formater des textes.

À la différence de LaTeX ou d'HTML il enrichit le document de manière « non intrusive », c'est-à-dire que les fichiers restent directement lisibles.

docutils

Le projet `docutils`, qui inclut l'interpréteur `reST`, fournit un jeu d'utilitaires :

`rst2html` génère un rendu HTML avec une feuille de style css intégrée ;

`rst2latex` crée un fichier LaTeX équivalent ;

`rst2s5` construit une présentation au format s5, qui permet de créer des présentations interactives en HTML.

Sphinx

Sphinx est un logiciel libre de type générateur de documentation. Il s'appuie sur des fichiers au format reStructuredText, qu'il convertit en HTML, PDF, man, et autres formats.

De nombreux projets utilisent Sphinx pour leur documentation officielle, cf. `http://sphinx-doc.org/examples.html`

rst2pdf

Par ailleurs, le programme **rst2pdf** génère directement une documentation au format PDF.

Voici un exemple [1] simple de fichier texte au format reST.
On remarque entre autres que :

— la principale balise est la **ligne blanche** qui sépare les différentes structures du texte ;
— la structuration se fait en soulignant les titres des sections de différents niveaux avec des caractères de ponctuation (= - _ : , etc.). À chaque fois qu'il rencontre un texte ainsi souligné, l'interpréteur associe le caractère utilisé à un niveau de section ;
— un titre est généralement souligné et surligné avec le même caractère, comme dans l'exemple suivant :

```
======================
Fichier au format reST
======================

Section 1
=========
On est dans la section 1.

Sous-section
~~~~~~~~~~~~
Ceci est une sous-section.

Sous-sous-section
.................
Ceci est une sous-sous-section.

.. et ceci un commentaire

Section 2
=========
La section 2 est ‘‘beaucoup plus‘‘ **intéressante** que la section 1.

Section 3
=========
La section 2 est un peu vantarde : la section 1 est *très bien*.

Une image au format ”png”
~~~~~~~~~~~~~~~~~~~~~~~~~
.. figure:: helen.png
    :scale: 30%
```

L'utilitaire `rst2pdf`, appliqué à ce fichier, produit le fichier de même nom (☞ Fig. 9.5), mais avec l'extension `.pdf`.

9.6.2 Le module `doctest`

Le principe du *literate programming* (ou programmation littéraire) de Donald Knuth consiste à mêler dans le source le code et la documentation du programme.

Ce principe été repris en Python pour documenter les API via les chaînes de documentation (*docstring*). Des programmes comme `Epydoc` peuvent alors les extraire des modules pour composer une documentation séparée.

Il est possible d'aller plus loin et d'inclure dans les chaînes de documentation des exemples d'utilisation, écrits sous la forme de session interactive.

Examinons deux exemples.

Pour chacun, nous donnerons d'une part le source muni de sa chaîne de documentation dans lequel le module standard `doctest` permet d'extraire, puis de lancer ces sessions pour vérifier qu'elles fonctionnent et, d'autre part un résultat de l'exécution.

1. cf. [7], p. 127

FIGURE 9.5 – Exemple de sortie au format PDF.

Premier exemple : `documentation1.py`

```
# -*- coding: utf-8 -*-
"""Module d'essai de doctest."""

import doctest

def somme(a, b):
    """Renvoie a + b.

    >>> somme(2, 2)
    4
    >>> somme(2, 4)
    6
    """
    return a+b

if __name__ == '__main__':
    print("{:-^40}".format(" Mode silencieux "))
    doctest.testmod()
    print("Si tout va bien, on n'a rien vu !")

    print("\n{:-^40}".format(" Mode détaillé "))
    doctest.testmod(verbose=True)
```

L'exécution de ce fichier donne :

```
---------- Mode silencieux ------------
Si tout va bien, on n'a rien vu !

---------- Mode détaillé ------------
Trying:
    somme(2, 2)
Expecting:
    4
ok
```

```
Trying:
    somme(2, 4)
Expecting:
    6
ok
1 items had no tests:
    __main__
1 items passed all tests:
   2 tests in __main__.somme
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

Deuxième exemple : `documentation2.py`

```
# -*- coding: UTF-8 -*-
"""Module d'essai de doctest."""

# fonctions
def accentEtrange(texte):
    """Ajoute un accent étrange à un texte.

    Les 'r' sont Triplés, les 'e' suivi d'un 'u'

    Exemple :

    >>> texte = "Est-ce que tu as regardé la télé hier soir ? Il y avait un théma sur les ramasseurs d'escargots en
        Laponie, ils en bavent..."
    >>> accentEtrange(texte)
    Est-ceu queu tu as rRreugarRrdé la télé hieurRr soirRr ? Il y avait un théma surRr leus rRramasseuurRrs d'
        euscarRrgots eun Laponieu, ils eun baveunt...

    Cette technique permet 'dinternationaliser les applications
    pour les rendre compatibles avec certaines régions françaises.
    """
    texte = texte.replace('r', 'rRr')
    print(texte.replace('e', 'eu'))

def _test():
    import doctest
    doctest.testmod(verbose=True)

if __name__ == '__main__':
    _test()
```

L'exécution de ce fichier donne :

```
Trying:
    texte = "Est-ce que tu as regardé la télé hier soir? Il y avait un théma sur les ramasseurs d'escargots en Laponie,
ils en bavent..."
Expecting nothing
ok
Trying:
    accentEtrange(texte)
Expecting:
    Est-ceu queu tu as rRreugarRrdé la télé hieurRr soirRr? Il y avait un théma surRr leus rRramasseuurRrs d'euscarRrgots
 eun Laponieu, ils eun baveunt...
ok
2 items had no tests:
    __main__
    __main__._test
1 items passed all tests:
   2 tests in __main__.accentEtrange
2 tests in 3 items.
2 passed and 0 failed.
Test passed.
```

9.6.3 Le développement dirigé par la documentation

Comme on peut le voir, la documentation intégrée présente néanmoins un défaut : quand la documentation augmente, on ne voit plus le code !

La solution est de déporter cette documentation : la fonction `doctest.testfile()` permet d'indiquer le nom du fichier de documentation.

Qui plus est, on peut écrire ce fichier au format reST, ce qui permet de faire coup double. D'une part, on dispose des tests intégrés à la fonction (ou à la méthode) et, d'autre part, le même fichier fournit une documentation à jour.

Exemple : `test_documentation2.py`

Fichier de documentation [1] :

```
Le module ''accent''
====================

Test de la fonction ''accentEtrange''
-------------------------------------

Ce module fournit une fonction ''accentEtrange''.
On peut ainsi ajouter un accent à un texte :

        >>> from doctest2 import accentEtrange
        >>> texte = "Est-ce que tu as regardé la télé hier soir ? Il y avait un théma sur
        les ramasseurs d'escargots en Laponie, ils en bavent..."
        >>> accentEtrange(texte)
        Est-ceu queu tu as rRreugarRrdé la télé hieurRr soirRr ? Il y avait un théma surRr
        leus rRramasseuurRrs d'euscarRrgots eun Laponieu, ils eun baveunt...

Les ''r'' sont triplés et les ''e'' épaulés par des ''u''. Cette technique permet
de se passer de systèmes de traductions complexes pour faire fonctionner
les logiciels dans certaines régions.
```

Source du module :

```
import doctest
doctest.testfile("test_documentation2.txt", verbose=True)
```

Nous produisons la documentation HTML par la commande :

```
rst2html test_documentation2.txt test_documentation2.html
```

Elle est illustrée ☞ Fig. 9.6.

9.6.4 Pour aller plus loin

La programmation avec IPython…

1. cf. [7], p. 127.

FIGURE 9.6 – Documentation au format html du script `test_documentation2.py`.

> Retour *Table des matières*

Annexe A

Interlude

Le Zen de Python [1]

Préfère
la beauté à la laideur,
l'explicite à l'implicite,
le simple au complexe,
le complexe au compliqué,
le déroulé à l'imbriqué,
l'aéré au compact.
Prends en compte la lisibilité.
Les cas particuliers ne le sont jamais assez pour violer les règles.
Mais, à la pureté, privilégie l'aspect pratique.
Ne passe pas les erreurs sous silence,
Ou bâillonne-les explicitement.
Face à l'ambiguïté, à deviner ne te laisse pas aller.
Sache qu'il ne devrait avoir qu'une et une seule façon de procéder.
Même si, de prime abord, elle n'est pas évidente, à moins d'être Néerlandais.
Mieux vaut maintenant que jamais.
Cependant jamais est souvent mieux qu'immédiatement.
Si l'implémentation s'explique difficilement, c'est une mauvaise idée.
Si l'implémentation s'explique aisément, c'est peut-être une bonne idée.
Les espaces de noms, sacrée bonne idée ! Faisons plus de trucs comme ça !

1. Tim Peters (PEP nº 20), traduction Cécile Trevian et Bob Cordeau.
Retour chap.1, p. 1

Le Graal de Python [1] !

Arthur

Lancelot ! Lancelot ! Lancelot !

[mégaphone de police]

Lancelooooooooot !

Lancelot

Bloody hell, mais que se passe-t-il donc, mon Roi ?

Arthur

Bevedere, explique-lui !

Bevedere

Nous devons te parler d'un nouveau langage de programmation : Python

Lancelot

Nouveau ? Cela fait bien dix ans qu'il existe, et je ne vois pas en quoi cela va nous aider à récupérer le Saint-Graal !

Bevedere

Saint-Graal, Saint-Graal...

[soupir]

Tu ne peux pas penser à des activités plus saines que cette quête stupide de temps en temps ?

Arthur

[sort une massue et assomme Bevedere avec]

Son explication était mal partie de toute manière.

Gardes français

Est-ce que ces messieurs les Anglais peuvent aller s'entre-tuer plus loin ?

Ne voyez-vous pas que nous sommes concentrés sur notre jeu en ligne ?

Arthur

Ce tunnel sous la Manche, quelle hérésie !

[racle sa gorge]

Lancelot, assieds-toi, et écoute-moi. (et ferme ce laptop, bloody hell !)

Lancelot

[rabat l'écran de son laptop]

1. cf. [6], p. 127.

Arthur

La quête a changé. Tu dois maintenant apprendre le langage Python, et découvrir pourquoi il est de plus en plus prisé par mes sujets.

Lancelot

Mais…

Arthur

Il n'y a pas de mais !

[menace Lancelot avec sa massue]

Je suis ton Roi. dot slash.

Prends ce livre, et au travail !

Gardes français

Oui, au travail, et en silence !

Annexe B

Passer du problème au programme

Lorsqu'on a un problème à résoudre par un programme, la difficulté est de savoir :

1. Par où commencer ?
2. Comment concevoir l'algorithme ?

Prérequis

Au fur et à mesure que l'on acquière de l'expérience, on découvre et on apprend à utiliser les bibliothèques de modules et paquets qui fournissent des types de données et des services avancés, évitant d'avoir à re-créer, coder et déboguer une partie de la solution.

Réutiliser

La première chose à faire est de vérifier qu'il n'existe pas déjà une solution (même partielle) au problème que l'on pourrait reprendre *in extenso* ou dont on pourrait s'inspirer. On peut chercher dans les nombreux modules standard installés avec le langage, dans les dépôts institutionnels de modules tiers (le *Python Package Index* [1] par exemple), ou encore utiliser les moteurs de recherche sur l'Internet. Si on ne trouve pas de solution existante dans notre langage préféré, on peut trouver une solution dans un autre langage, qu'il n'y aura plus qu'à adapter.

L'analyse qui permet de créer un algorithme et la programmation ensuite, sont deux phases qui nécessitent de la pratique avant de devenir « évidentes » ou « faciles ».

Réfléchir à un algorithme

Pour démarrer, il faut partir d'éléments réels, mais sur un échantillon du problème comportant peu de données, un cas que l'on est capable de traiter « à la main ».

Il est fortement conseillé de démarrer sur papier ou sur un tableau (le papier ayant l'avantage de laisser plus facilement des traces des différentes étapes).

On identifie tout d'abord quelles sont les données que l'on a à traiter en entrée et quelles sont les données que l'on s'attend à trouver en sortie. Pour chaque donnée, on essaie de préciser quel est son domaine, quelles sont ses limites, quelles contraintes la lient aux autres données.

Résoudre « à la main »

On commence par une résolution du problème, en réalisant les transformations et calculs sur notre échantillon de problème, en fonctionnant par étapes.

À chaque étape, on note :

— quelles sont les étapes pertinentes, sur quels critères elles ont été choisies ;
— quelles sont les séquences d'opérations que l'on a répétées.

Lorsque l'on tombe sur des étapes complexes, on découpe en sous-étapes, éventuellement en les traitant séparément comme un algorithme de résolution d'un sous-problème. Le but est d'arriver à un niveau de détails suffisamment simple ; soit qu'il s'agisse d'opérations très basiques (opération sur un texte, expression de calcul numérique…), soit que l'on pense/sache qu'il existe déjà un outil pour traiter ce sous-problème (calcul de sinus pour un angle, opération de tri sur une séquence de données…).

Lors de ce découpage, il faut éviter de considérer des opérations comme « implicites » ou « évidentes », il faut préciser d'où proviennent les informations et ce que l'on fait des résultats. Par exemple on ne

1. http://pypi.python.org/

considère pas « un élément » mais « le nom traité est l'élément suivant de la séquence de noms » ou encore « le nom traité est le x^eélément de la séquence de noms ».

Normalement, au cours de ces opérations, on a commencé à nommer les données et les étapes au fur et à mesure qu'on en a eu besoins.

Formaliser

Une fois qu'on a un brouillon des étapes, il faut commencer à mettre en forme et à identifier les constructions algorithmiques connues et les données manipulées :

- boucles (sur quelles informations, condition d'arrêt) ;
- tests (quelle condition) ;
- informations en entrée, quel est leur type, quelles sont les contraintes pour qu'elles soient valides et utilisables, d'où viennent-elles :
  - déjà présentes en mémoire,
  - demandées à l'utilisateur,
  - lues dans des fichiers ou récupérées ailleurs (sur l'Internet par exemple) ;
- calculs et expressions :
  - quel genre de données sont nécessaires, y a-t-il des éléments constants à connaître, des résultats intermédiaires à réutiliser,
  - on peut identifier ici les contrôles intermédiaires possibles sur les valeurs qui puissent permettre de vérifier que l'algorithme se déroule bien ;
- stockage des résultats intermédiaires ;
- résultat final — à quel moment l'a-t-on, qu'en fait-on :
  - retourné dans le cadre d'une fonction,
  - affiché à l'utilisateur,
  - sauvegardé dans un fichier.

Factoriser

Le but est d'identifier les séquences d'étapes qui se répètent en différents endroits, séquences qui seront de bons candidats pour devenir des fonctions ou des classes. Ceci peut être fait en même temps que l'on formalise.

Passer de l'idée au programme

Le passage de l'idée puis de l'algorithme au code dans un programme, est relativement facile en Python, car celui-ci est très proche d'un langage d'algorithmique.

- Les ***noms*** des choses que l'on a manipulées vont nous donner des **`variables`**.
- Les ***tests*** vont se transformer en **`if condition:`**
- Les ***boucles sur des séquences*** d'informations vont se transformer en **`for variable in séquence:`**
- Les ***boucles avec expression*** de condition vont se transformer en **`while conditions:`**
- Les ***séquences d'instructions qui se répètent*** en différents endroits vont se transformer en **`fonctions`**.
- Le ***retour de résultat*** d'une séquence (fonction) va se traduire en **`return variable`**.
- Les ***conditions sur les données*** nécessaires pour un traitement vont identifier des tests d'erreurs et des levées d'exception.

Annexe C

Jeux de caractères et encodage

Position du problème

Nous avons vu que l'ordinateur code toutes les informations qu'il manipule en *binaire*. Pour coder les nombres entiers un changement de base suffit, pour les flottants, on utilise une norme (IEEE 754), mais la situation est plus complexe pour représenter les caractères.

Tous les caractères que l'on peut écrire à l'aide d'un ordinateur sont représentés en mémoire par des nombres. On parle d'*encodage*. Le « a » minuscule par exemple est représenté, ou encodé, par le nombre 97. Pour pouvoir afficher ou imprimer un caractère lisible, leurs dessins, appelés *glyphes*, sont stockés dans des catalogues appelés *polices de caractères*. Les logiciels informatiques parcourent ces catalogues pour rechercher le glyphe qui correspond à un nombre. Suivant la police de caractères, on peut ainsi afficher différents aspects du même « a » (97).

Les 128 premiers caractères comprennent les caractères de l'alphabet latin (non altérés [1]), les majuscules et les minuscules, les chiffres arabes, et quelques signes de ponctuation, c'est la fameuse table ASCII [2] (☞ Fig. C.1). Chaque pays a ensuite complété ce jeu initial suivant les besoins de sa propre langue, créant ainsi son propre système d'encodage.

Cette méthode a un fâcheux inconvénient : le caractère « à » français peut alors être représenté par le même nombre que le caractère « å » scandinave dans les deux encodages, ce qui rend impossible l'écriture d'un texte bilingue avec ces deux caractères !

Pour écrire un document en plusieurs langues, le standard nommé Unicode a été développé et maintenu par un consortium [3]. Il permet d'unifier une grande table de correspondance internationale, sans chevauchement entre les caractères. Les catalogues de police se chargent ensuite de fournir des glyphes correspondants.

n°	char	n°	char	n°	char	n°	char
32		56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	"	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	'	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	-	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	_	119	w
48	0	72	H	96	`	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	|
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g	127	△

Figure C.1 – Table ASCII.

1. C'est-à-dire sans signe diacritique, par exemple les accents, le tréma, la cédille…
2. American Standard Code for Information Interchange
3. Le *Consortium Unicode*.

L'encodage UTF-8

Comme il s'agit de différencier plusieurs centaines de milliers de caractères (on compte plus de 6000 langues dans le monde), il n'est évidemment pas possible de les encoder sur un seul octet.

En fait, la norme Unicode ne fixe aucune règle concernant le nombre d'octets ou de bits à réserver pour l'encodage, mais spécifie seulement la valeur numérique de l'identifiant associé à chaque caractère (☞ Fig. C.2).

Comme la plupart des textes produits en Occident utilisent essentiellement la table ASCII qui correspond justement à la partie basse de la table Unicode, l'encodage le plus économique est l'UTF-8 [1] :

— pour les codes 0 à 127 (cas les plus fréquents), l'UTF-8 utilise l'octet de la table ASCII ;
— pour les caractères spéciaux (codes 128 à 2047), quasiment tous nos signes diacritiques, l'UTF-8 utilise 2 octets ;
— pour les caractères spéciaux encore moins courants (codes 2048 à 65535), l'UTF-8 utilise 3 octets ;
— enfin pour les autres (cas rares), l'UTF-8 en utilise 4.

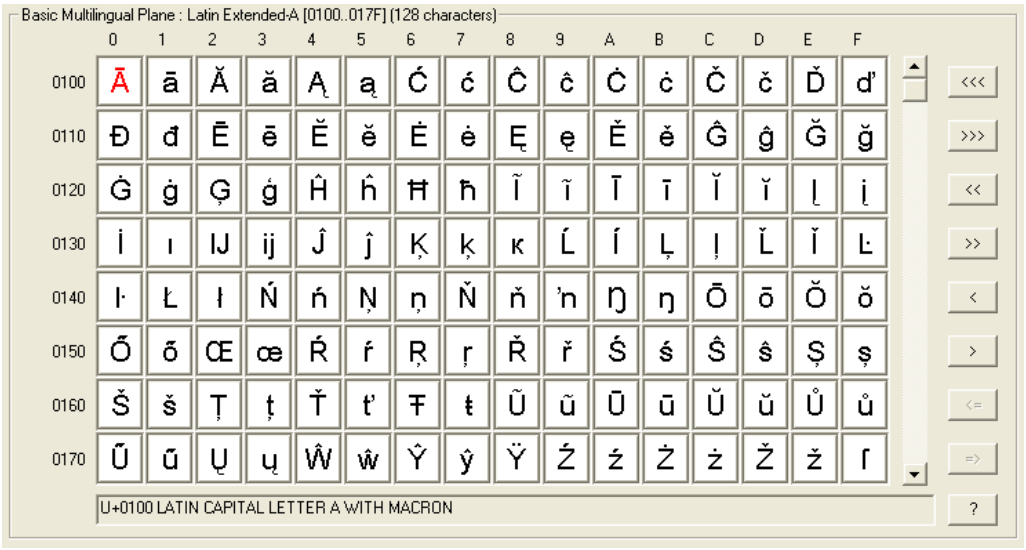


FIGURE C.2 – Extrait de la table Unicode.

Exemple de l'encodage UTF-8 du caractère Unicode « é » :

Symbole	Code décimal	Code hexadécimal	Encodage UTF-8
é	233	e9	C3 A9

Voici trois exemples de caractères spéciaux codés en notation hexadécimale et séparés par le caractère d'échappement de la tabulation :

```
>>> print("\u00e9 \t \u0110 \t \u0152")
é	Đ	Œ
```

Applications aux scripts Python

En Python 3, les chaînes de caractères (le type `str()`) sont des chaînes Unicode. Par ailleurs, puisque les scripts Python que l'on produit avec un éditeur sont eux-mêmes des textes, ils sont suceptibles d'être encodés suivant différents formats. Afin que Python utilise le bon, il est important d'indiquer l'encodage de caractères utilisé. On le précise obligatoirement en 1re ou 2e ligne des sources.

Les encodages les plus fréquents sont [2] :

```
# -*- coding: utf8 -*-
```

ou :

```
# -*- coding: latin1 -*-
```

1. Retour chap. 2, p. 7
2. Notons que `utf8` et `latin1` sont des alias de `utf-8` et `latin-1`.

Annexe D

Les expressions régulières

Les expressions régulières [a] fournissent une notation générale très puissante permettant de décrire abstraitement des éléments textuels. Il s'agit d'un vaste domaine dont nous ne proposons qu'une introduction.

a. Ici, l'adjectif *régulier* est employé au sens de *qui obéit à des règles.*

Introduction

Dès les débuts de l'informatique, les concepteurs des systèmes d'exploitation eurent l'idée d'utiliser des *métacaractères* permettant de représenter des modèles généraux. Par exemple, dans un shell Linux ou dans une fenêtre de commande Windows, le symbole * [1] remplace une série de lettres, ainsi `*.png` indique tout nom de fichier contenant l'extension `png`. Python offre en standard les modules `glob` et `fnmatch` pour utiliser la notations des métacaractères.

Depuis ces temps historiques, les informaticiens [2] ont voulu généraliser ces notations. On distingue classiquement trois stades dans l'évolution des expressions régulières :
— les expressions régulières de base (BRE, *Basic Regular Expressions*) ;
— les expressions régulières étendues (ERE, *Extended Regular Expressions*) ;
— les expressions régulières avancées (ARE, *Advanced Regular Expressions*).

Trois stades auxquels il convient d'ajouter le support de l'encodage Unicode.

Python supporte toutes ces évolutions.

Les expressions régulières

Une expression régulière [3] se lit (et se construit) de gauche à droite. Elle constitue ce qu'on appelle traditionnellement un *motif* de recherche [4].

Les expressions régulières de base

Elles utilisent six symboles qui, dans le contexte des expressions régulières, acquièrent les significations suivantes :

Le point `.` représente une seule instance de n'importe quel caractère sauf le caractère de fin de ligne. Ainsi l'expression `t.c` représente toutes les combinaisons de trois lettres commençant par `t` et finissant par `c`, comme *tic*, *tac*, *tqc* ou *t9c*, alors que `b.l..` pourrait représenter *bulle*, *balai* ou *bêler* ;

La paire de crochets `[ ]` représente une occurrence quelconque des caractères qu'elle contient. Par exemple `[aeiouy]` représente une voyelle, et `Duran[dt]` désigne *Durand* ou *Durant*.

Entre les crochets, on peut noter un intervalle en utilisant le tiret. Ainsi, `[0-9]` représente les chiffres de 0 à 9, et `[a-zA-Z]` représente une lettre minuscule ou majuscule.

On peut de plus utiliser l'accent circonflexe en première position dans les crochets pour indiquer « le contraire de ». Par exemple `[^a-z]` représente autre chose qu'une lettre minuscule, et `[^'"]` n'est ni une apostrophe ni un guillemet ;

1. Appelé aussi *joker* (ou *wildcard* en anglais).
2. En particulier le mathématicien Stephen Kleene (1909–1994).
3. Souvent abrégée en *regex*.
4. En anglais *search pattern*.

L'astérisque `*` est un *quantificateur*, il signifie aucune ou plusieurs occurrences du caractère ou de l'élément qui le précède immédiatement.

L'expression `ab*` signifie la lettre `a` suivie de zéro ou plusieurs lettres `b`, par exemple *ab*, *a* ou *abbb* et `[A-Z]*` correspond à zéro ou plusieurs lettres majuscules ;

L'accent circonflexe `^` est une *ancre*. Il indique que l'expression qui le suit se trouve en début de ligne.

L'expression `^Depuis` indique que l'on recherche les lignes commençant par le mot *Depuis* ;

Le symbole dollar `$` est aussi une *ancre*. Il indique que l'expression qui le précède se trouve en fin de ligne.

L'expression `suivante :$` indique que l'on recherche les lignes se terminant par *suivante :*

L'expression `^Les expressions régulières$` extrait les lignes ne contenant que la chaîne *Les expressions régulières*, alors que `^$` extrait les lignes vides ;

La contre-oblique `\` permet d'*échapper* à la signification des métacaractères.

Ainsi `\.` désigne un véritable point, `\*` un astérisque, `\^` un accent circonflexe, `\$` un dollar et `\\` une contre-oblique.

Les expressions régulières étendues

Elles ajoutent cinq symboles qui ont les significations suivantes :

La paire de parenthèses `( )` est utilisée à la fois pour former des sous-motifs et pour délimiter des sous-expressions, ce qui permettra d'extraire des parties d'une chaîne de caractères.

L'expression `(to)*` désignera *to*, *tototo*, etc…

Le signe plus `+` est un *quantificateur* comme `*`, mais il signifie une ou plusieurs occurrences du caractère ou de l'élément qui le précède immédiatement.

L'expression `ab+` signifie la lettre `a` suivie d'une ou plusieurs lettres `b` ;

Le point d'interrogation `?` troisième *quantificateur*, il signifie zéro ou une instance de l'expression qui le précède.

Par exemple `écran(s)?` désigne *écran* ou *écrans* ;

La paire d'accolades `{ }` précise le nombre d'occurrences permises pour le motif qui le précède.

Par exemple `[0-9]{2,5}` attend entre deux et cinq nombres décimaux.

Les variantes suivantes sont disponibles : `[0-9]{2,}` signifie au minimum deux occurrences d'entiers décimaux et `[0-9]{2}` deux occurrences exactement ;

La barre verticale `|` représente des choix multiples dans un sous-motif.

L'expression `Duran[dt]` peut aussi s'écrire `(Durand|Durant)`. On pourrait utiliser l'expression `(lu|ma|me|je|ve|sa|di)` dans l'écriture d'une date.

Dans de nombreux outils et langages (dont Python), la syntaxe étendue comprend aussi une série de séquences d'échappement :

Séquences d'échappement

`\`	symbole d'échappement
`\e`	séquence de contrôle *escape*
`\f`	saut de page
`\n`	fin de ligne
`\r`	retour-chariot
`\t`	tabulation horizontale
`\v`	tabulation verticale
`\d`	classe des nombres entiers
`\s`	classe des caractères d'espacement
`\w`	classe des caractères alphanumériques
`\b`	délimiteurs de début ou de fin de mot
`\D`	négation de la classe `\d`
`\S`	négation de la classe `\s`
`\W`	négation de la classe `\w`
`\B`	négation de la classe `\b`

Les expressions régulières avec Python

Le module `re` permet d'utiliser les expressions régulières dans les scripts Python. Les scripts devront donc comporter la ligne :

```
import re
```

Pythonismes

Le module `re` utilise la notation objet. Les motifs et les correspondances de recherche seront des objets de la classe `re` auxquels on pourra appliquer des méthodes.

Utilisation des *raw strings*

La syntaxe des motifs comprend souvent le caractère contre-oblique qui doit être lui-même échappé dans une chaîne de caractères, ce qui alourdit la notation. On peut éviter cet inconvénient en utilisant des « chaînes brutes ». Par exemple au lieu de :

```
"\\d\\d? \\w+ \\d{4}"
```

on écrira :

```
r"\d\d? \w+ \d{4}"
```

Les options de compilation

Grâce à un jeu d'*options de compilation*, il est possible de piloter le comportement des expressions régulières. On utilise pour cela la syntaxe `(?...)` avec les drapeaux suivants :

a correspondance ASCII (Unicode par défaut) ;

i correspondance non sensible à la casse ;

L les correspondances utilisent la *locale*, c'est-à-dire les particularité du pays ;

m correspondance dans des chaînes multilignes ;

s modifie le comportement du métacaractère point `.` qui représentera alors aussi le saut de ligne ;

u correspondance Unicode (par défaut) ;

x mode verbeux.

Voici un exemple de recherche non sensible à la casse :

```
>>> import re

>>> case = re.compile(r"[a-z]+")

>>> ignore_case = re.compile(r"(?i)[a-z]+")

>>> print(case.search("Bastille").group())
astille

>>> print(ignore_case.search("Bastille").group())
Bastille
```

Les motifs nominatifs

Python possède une syntaxe qui permet de nommer des parties de motif délimitées par des parenthèses, ce qu'on appelle un « motif nominatif » :

— syntaxe de création d'un motif nominatif : `(?P<nom_du_motif>)` ;

— syntaxe permettant de s'y référer : `(?P=nom_du_motif)` ;

— syntaxe à utiliser dans un motif de remplacement ou de substitution : `\g<nom_du_motif>`.

Exemples

On propose plusieurs exemples d'extraction de dates historiques telles que `"14 juillet 1789"`.

Extraction simple

Cette chaîne peut être décrite par le motif `\d\d? \w+ \d{4}` que l'on peut expliciter ainsi : « un ou deux entiers décimaux suivi d'un blanc suivi d'une chaîne d'au moins un caractère suivi d'un blanc suivi de quatre entiers décimaux ».

Détaillons le script :

```
import re

motif_date = re.compile(r"\d\d? \w+ \d{4}")
corresp = motif_date.search("Bastille le 14 juillet 1789")

print(corresp.group())
```

Après avoir importé le module `re`, la variable `motif_date` reçoit la forme compilée de l'expression régulière correspondant à une date historique. Puis on applique à ce motif compilé la méthode `search()` qui retourne la première position du motif dans la chaîne et l'affecte à la variable `corresp`. Enfin on affiche la correspondance complète (en ne donnant pas d'argument à `group()`.

Son exécution produit :

```
14 juillet 1789
```

Extraction des sous-groupes

On aurait pu affiner l'affichage du résultat en modifiant l'expression régulière de recherche de façon à pouvoir capturer les éléments du motif :

```
import re

motif_date = re.compile(r"(\d\d?) (\w+) (\d{4})")
corresp = motif_date.search("Bastille le 14 juillet 1789")

print("corresp.group() :", corresp.group())
print("corresp.group(1) :", corresp.group(1))
print("corresp.group(2) :", corresp.group(2))
print("corresp.group(3) :", corresp.group(3))
print("corresp.group(1,3) :", corresp.group(1,3))
print("corresp.groups() :", corresp.groups())
```

Ce qui produit à l'exécution :

```
corresp.group() : 14 juillet 1789
corresp.group(1) : 14
corresp.group(2) : juillet
corresp.group(3) : 1789
corresp.group(1,3) : ('14', '1789')
corresp.groups() : ('14', 'juillet', '1789')
```

Extraction des sous-groupes nommés

Une autre possibilité est l'emploi de la méthode `groupdict()` qui renvoie une liste comportant le nom et la valeur des sous-groupes trouvés (ce qui nécessite de nommer les sous-groupes).

```
import re

motif_date = re.compile(r"(?P<jour>\d\d?) (?P<mois>\w+) (\d{4})")
corresp = motif_date.search("Bastille le 14 juillet 1789")

print(corresp.groupdict())
print(corresp.group('jour'))
print(corresp.group('mois'))
```

Ce qui donne :

```
{'jour': '14', 'mois': 'juillet'}
14
juillet
```

Extraction d'une liste de sous-groupes

La méthode `findall` retourne une liste des occurrences trouvées. Si par exemple on désire extraire tous les nombres d'une chaîne, on peut écrire :

```
>>> import re

>>> nbr = re.compile(r"\d+")

>>> print(nbr.findall("Bastille le 14 juillet 1789"))
['14', '1789']
```

Scinder une chaîne

La méthode `split()` permet de scinder une chaîne à chaque occurrence du motif. Si on ajoute un paramètre numérique `n` (non nul), la chaîne est scindée en au plus `n` éléments :

```
>>> import re

>>> nbr = re.compile(r"\d+")

>>> print("Une coupure à chaque occurrence :", nbr.split("Bastille le 14 juillet 1789"))
Une coupure à chaque occurrence : ['Bastille le ', ' juillet ', '']

>>> print("Une seule coupure :", nbr.split("Bastille le 14 juillet 1789", 1))
Une seule coupure : ['Bastille le ', ' juillet 1789']
```

Substitution au sein d'une chaîne

La méthode `sub()` effectue des substitutions dans une chaîne. Le remplacement peut être une chaîne ou une fonction. Comme on le sait, en Python, les chaînes de caractères sont non modifiables et donc les substitutions produisent de nouvelles chaînes.

Exemples de remplacement d'une chaîne par une autre et des valeurs décimales en leur équivalent hexadécimal :

```
import re

def int2hexa(match):
   return hex(int(match.group()))

anniv = re.compile(r"1789")
print("Premier anniversaire :", anniv.sub("1790", "Bastille le 14 juillet 1789"))

nbr = re.compile(r"\d+")
print("En hexa :", nbr.sub(int2hexa, "Bastille le 14 juillet 1789"))
```

Ce qui affiche :

```
Premier anniversaire : Bastille le 14 juillet 1790
En hexa : Bastille le 0xe juillet 0x6fd
```

Les deux notations suivantes sont disponibles pour les substitutions :

Séquences de substitution

&	contient toute la chaîne recherchée par un motif
\n	contient la sous-expression capturée par la n^e paire de parenthèses du motif de recherche ($1 \leq n \leq 9$)

Annexe E

Utilisation des notebooks

Lorsqu'on a un problème à résoudre par un programme, la difficulté est de savoir :

1. Par où commencer ?
2. Comment concevoir l'algorithme ?

Prérequis

Au fur et à mesure que l'on acquière de l'expérience, on découvre et on apprend à utiliser les bibliothèques de modules et paquets qui fournissent des types de données et des services avancés, évitant d'avoir à re-créer, coder et déboguer une partie de la solution.

Réutiliser

La première chose à faire est de vérifier qu'il n'existe pas déjà une solution (même partielle) au problème que l'on pourrait reprendre *in extenso* ou dont on pourrait s'inspirer. On peut chercher dans les nombreux modules standard installés avec le langage, dans les dépôts institutionnels de modules tiers (le *Python Package Index* [1] par exemple), ou encore utiliser les moteurs de recherche sur l'Internet. Si on ne trouve pas de solution existante dans notre langage préféré, on peut trouver une solution dans un autre langage, qu'il n'y aura plus qu'à adapter.

1. http://pypi.python.org/

Annexe F

Les messages d'erreur de l'interpréteur

Lorsqu'on a un problème à résoudre par un programme, la difficulté est de savoir :

1. Par où commencer ?
2. Comment concevoir l'algorithme ?

Prérequis

Au fur et à mesure que l'on acquière de l'expérience, on découvre et on apprend à utiliser les bibliothèques de modules et paquets qui fournissent des types de données et des services avancés, évitant d'avoir à re-créer, coder et déboguer une partie de la solution.

Réutiliser

La première chose à faire est de vérifier qu'il n'existe pas déjà une solution (même partielle) au problème que l'on pourrait reprendre *in extenso* ou dont on pourrait s'inspirer. On peut chercher dans les nombreux modules standard installés avec le langage, dans les dépôts institutionnels de modules tiers (le *Python Package Index* [1] par exemple), ou encore utiliser les moteurs de recherche sur l'Internet. Si on ne trouve pas de solution existante dans notre langage préféré, on peut trouver une solution dans un autre langage, qu'il n'y aura plus qu'à adapter.

1. http://pypi.python.org/

Bibliographie et Webographie

[1] Swinnen, Gérard, *Apprendre à programmer avec Python 3*, Eyrolles, 2010.

[2] Summerfield, Mark, *Programming in Python 3*, Addison-Wesley, 2e édition, 2009.

[3] Martelli, Alex, *Python en concentré*, O'Reilly, 2004.

[4] Martelli, Alex, Martelli Ravenscroft, Anna, Ascher, David, *Python par l'exemple*, O'Reilly, 2006.

[5] Lutz, Mark et Bailly, Yves, *Python précis et concis*, O'Reilly, 2e édition, 2005.

[6] Ziadé, Tarek, *Programmation Python. Conception et optimisation*, Eyrolles, 2e édition, 2009.

[7] Ziadé, Tarek, *Python : Petit guide à l'usage du développeur agile*, Dunod, 2007.

[8] Hellmann, Doug, *The Python Standard Library by Example*, Addison-Wesley, 2011.

[9] Langtangen, Hans Petter, *A Primer on Scientific Programming with Python*, Springer, 2011.

[10] Beazley, David M., *Python. Essential Reference*, Addison Wesley, 4e édition, 2009.

[11] Younker, Jeff, *Foundations of Agile Python Development*, Apress, 2008.

[12] Rouquette Maïeul, *LaTeX appliqué aux sciences humaines*, Atramenta, 2012.

[13] Chevalier Céline et collectif, *LaTeX pour l'impatient*, H & K, 3e édition, 2009.

[14] Carella, David, *Règles typographiques et normes. Mise en pratique avec LaTeX*, Vuibert, 2006.

[15] Rossant, Cyrille, *Learning IPython for Interactive Computing and Data Visualization*, Packt Publishing, 2013.

- Les sites généraux :
  www.python.org
  pypi.python.org/pypi
  code.google.com/p/pythonxy/wiki/Downloads
- Interpréteur et EDI spécialisés :
  ipython.org
  code.google.com/p/spyderlib
  www.wingware.com/downloads/wingide-101
  www.scintilla.org/SciTEDownload.html
  code.google.com/p/pyscripter
- Les outils :
  matplotlib.org
  www.inkscape-fr.org
  code.google.com/p/rst2pdf
  www.texniccenter.org
- L'encodage :
  sametmax.com/cours-et-tutos
- Une petite référence `NumPy` :
  mathprepa.fr/python-project-euler-mpsi
- Le lien des liens :
  perso.limsi.fr/pointal

Mémento Bases Python 3

Types de base

entier, flottant, booléen, chaîne

```
  int 783    0    -192
float 9.23   0.0   -1.7e-6        (e-6 = 10^-6)
 bool True   False
  str "Un\nDeux"   'L\'âme'
```

- `\n` : retour à la ligne
- `'` : échappé
- multiligne : `"""X\tY\tZ 1\t2\t3"""` (`\t` : tabulation)
- *non modifiable, séquence ordonnée de caractères*

Types Conteneurs

- séquences ordonnées, accès index rapide, valeurs répétables

```
  list [1,5,9]   ["x",11,8.9]   ["mot"]   []
 tuple (1,5,9)   11,"y",7.4     ("mot",)  ()
```

non modifiable → `str` en tant que séquence ordonnée de caractères

expression juste avec des virgules

- sans ordre *a priori*, clé unique, accès par clé rapide ; clés = types de base ou tuples

```
  dict {"clé":"valeur"}                          {}
dictionnaire
couples clé/valeur {1:"un",3:"trois",2:"deux",3.14:"π"}
ensemble
   set {"clé1","clé2"}   {1,9,3,0}   set()
```

Identificateurs

pour noms de variables, fonctions, modules, classes…

`a..zA..Z_` suivi de `a..zA..Z_0..9`

- accents possibles mais à éviter
- mots clés du langage interdits
- distinction casse min/MAJ

```
☺ a toto x7 y_max BigOne
☹ 8y and
```

Affectation de variables

```
x = 1.2+8+sin(0)
```

valeur ou expression de calcul

nom de variable (identificateur)

```
y,z,r = 9.2,-7.6,"bad"
```

noms de variables — conteneur de plusieurs valeurs (ici un tuple)

```
x+=3     incrémentation
x-=2     décrémentation
x=None   valeur constante « non défini »
```

Conversions

`type(`*expression*`)`

```
int("15")        on peut spécifier la base du nombre entier en 2nd paramètre
int(15.56)       troncature de la partie décimale (round(15.56) pour entier arrondi)
float("-11.24e8")
str(78.3)        et pour avoir la représentation littérale → repr("Texte")
```

voir au verso le formatage de chaînes, qui permet un contrôle fin

`bool` → utiliser des comparateurs (avec `==`, `!=`, `<`, `>`, …), résultat logique booléen

```
list("abc") → ['a','b','c']          (utilise chaque élément de la séquence en paramètre)
dict([(3,"trois"),(1,"un")]) → {1:'un',3:'trois'}
set(["un","deux"]) → {'un','deux'}   (utilise chaque élément de la séquence en paramètre)
":".join(['toto','12','pswd']) → 'toto:12:pswd'
```

`":"` : chaîne de jointure ; `['toto','12','pswd']` : séquence de chaînes

```
"des mots espacés".split() → ['des','mots','espacés']
"1,4,8,2".split(",") → ['1','4','8','2']
```

`","` : chaîne de séparation

Indexation des séquences

pour les listes, tuples, chaînes de caractères,…

```
index négatif   -6   -5   -4    -3    -2    -1
index positif    0    1    2     3     4     5
lst=           [11,  67,  "abc", 3.14, 42,  1968]
tranche positive 0    1    2     3     4     5     6
tranche négative -6   -5   -4    -3    -2    -1
```

`len(lst)` → `6`

accès individuel aux éléments par `[`*index*`]`

- `lst[1]` → `67`
- `lst[0]` → `11` *le premier*
- `lst[-2]` → `42`
- `lst[-1]` → `1968` *le dernier*

accès à des sous-séquences par `[`*tranche début*`:`*tranche fin*`:`*pas*`]`

```
lst[:-1] → [11,67,"abc",3.14,42]
lst[1:-1] → [67,"abc",3.14,42]
lst[::2] → [11,"abc",42]
lst[:] → [11,67,"abc",3.14,42,1968]
lst[1:3] → [67,"abc"]
lst[-3:-1] → [3.14,42]
lst[:3] → [11,67,"abc"]
lst[4:] → [42,1968]
```

Indication de tranche manquante → à partir du début / jusqu'à la fin.

Sur les séquences modifiables, utilisable pour suppression `del lst[3:5]` *et modification par affectation* `lst[1:4]=['hop',9]`

Logique booléenne

Comparateurs: `<` `>` `<=` `>=` `==` `!=` (≤ ≥ = ≠)

- `a and b` et logique — *les deux en même temps*
- `a or b` ou logique — *l'un ou l'autre ou les deux*
- `not a` non logique
- `True` valeur constante vrai
- `False` valeur constante faux

Blocs d'instructions

instruction parente :

indentation !

bloc d'instructions 1…

instruction parente :

bloc d'instructions 2…

instruction suivante après bloc 1

Instruction conditionnelle

bloc d'instructions exécuté uniquement si une condition est vraie

```
if expression logique :
    bloc d'instructions
```

combinable avec des sinon si, sinon si... et un seul sinon final, exemple :

```
if x==42:
    # bloc si expression logique x==42 vraie
    print("vérité vraie")
elif x>0:
    # bloc sinon si expression logique x>0 vraie
    print("positivons")
elif bTermine:
    # bloc sinon si variable booléenne bTermine vraie
    print("ah, c'est fini")
else:
    # bloc sinon des autres cas restants
    print("ça veut pas")
```

Maths

nombres flottants... valeurs approchées ! — *angles en radians*

Opérateurs: `+` `-` `*` `/` `//` `%` `**` (× ÷ ÷ entière, reste ÷, a^b)

```
(1+5.3)*2→12.6
abs(-3.2)→3.2
round(3.57,1)→3.6
from math import sin,pi…
sin(pi/4)→0.707…
cos(2*pi/3)→-0.4999…
acos(0.5)→1.0471…
sqrt(81)→9.0      √
log(e**2)→2.0     etc. (cf doc)
```

Instruction boucle conditionnelle

bloc d'instructions exécuté tant que la condition est vraie

```
while expression logique:
    bloc d'instructions
```

```
s = 0
i = 1
```
*initialisations **avant** la boucle*

condition avec au moins une valeur variable (ici `i`)

```
while i <= 100:
    # bloc exécuté tant que i ≤ 100
    s = s + i**2
    i = i + 1
```
faire varier la variable de condition !

```
print("somme:",s)
```
résultat de calcul après la boucle

attention aux boucles sans fin !

$$s = \sum_{i=1}^{i=100} i^2$$

Contrôle de boucle

`break` *sortie immédiate*

`continue` *itération suivante*

Instruction boucle itérative

bloc d'instructions exécuté pour chaque élément d'un conteneur ou d'un itérateur

```
for variable in séquence:
    bloc d'instructions
```

Parcours des **valeurs** de la séquence

```
s = "Du texte"
cpt = 0
```
*initialisations **avant** la boucle*

variable de boucle, valeur gérée par l'instruction `for`

```
for c in s:
    if c == "e":
        cpt = cpt + 1
print("trouvé",cpt,"'e'")
```
Comptage du nombre de `e` dans la chaîne.

boucle sur dict/set = boucle sur séquence des clés

utilisation des tranches pour parcourir un sous-ensemble de la séquence

Parcours des **index** de la séquence

- changement de l'élément à la position
- accès aux éléments autour de la position (avant/après)

```
lst = [11,18,9,12,23,4,17]
perdu = []
for idx in range(len(lst)):
    val = lst[idx]
    if val> 15:
        perdu.append(val)
        lst[idx] = 15
print("modif:",lst,"-modif:",perdu)
```
Bornage des valeurs supérieures à 15, mémorisation des valeurs perdues.

Parcours simultané **index** et **valeur** de la séquence:

```
for idx,val in enumerate(lst):
```

Affichage / Saisie

```
print("v=",3,"cm :",x,",",y+4)
```

éléments à afficher : valeurs littérales, variables, expressions

Options de `print`:

- `sep=" "` (séparateur d'éléments, défaut espace)
- `end="\n"` (fin d'affichage, défaut fin de ligne)
- `file=f` (print vers fichier, défaut sortie standard)

```
s = input("Directives:")
```

`input` retourne toujours une **chaîne**, la convertir vers le type désiré (cf encadré *Conversions* au recto).

Opérations sur conteneurs

`len(c)` → nb d'éléments

`min(c)` `max(c)` `sum(c)`

`sorted(c)` → *copie* triée

`val in c` → booléen, opérateur `in` de test de présence (`not in` d'absence)

`enumerate(c)` → *itérateur* sur (index,valeur)

*Spécifique aux **conteneurs de séquences** (listes, tuples, chaînes) :*

`reversed(c)` → *itérateur* inversé `c*5` → duplication `c+c2` → concaténation

`c.index(val)` → position `c.count(val)` → nb d'occurences

*Note: Pour dictionnaires et ensembles, ces opérations travaillent sur les **clés**.*

Opérations sur listes

modification de la liste originale

`lst.append(item)`	ajout d'un élément à la fin
`lst.extend(seq)`	ajout d'une séquence d'éléments à la fin
`lst.insert(idx,val)`	insertion d'un élément à une position
`lst.remove(val)`	suppression d'un élément à partir de sa valeur
`lst.pop(idx)`	suppression de l'élément à une position et retour de la valeur
`lst.sort()` `lst.reverse()`	tri / inversion de la liste *sur place*

Opérations sur dictionnaires

`d[clé]=valeur` `d.clear()`

`d[clé]`→*valeur* `del d[clé]`

`d.update(d2)` *mise à jour/ajout des couples*

`d.keys()` `d.values()` `d.items()` *vues sur les clés, valeurs, couples*

`d.pop(clé)`

Opérations sur ensembles

Opérateurs:

`|` → union (caractère barre verticale)

`&` → intersection

`-` `^` → différence/diff symétrique

`<` `<=` `>` `>=` → relations d'inclusion

`s.update(s2)`

`s.add(clé)` `s.remove(clé)`

`s.discard(clé)`

Fichiers

stockage de données sur disque, et relecture

```
f = open("fic.txt","w",encoding="utf8")
```

variable fichier pour les opérations

nom du fichier sur le disque (+chemin…)

mode d'ouverture
- `'r'` lecture (read)
- `'w'` écriture (write)
- `'a'` ajout (append)…

encodage des caractères pour les fichiers textes: `utf8` `ascii` `latin1` …

cf fonctions des modules `os` et `os.path`

en **écriture**

```
f.write("coucou")
```

en **lecture**

```
s = f.read(4)
```
si nb de caractères pas précisé, lit tout le fichier

chaîne vide si fin de fichier

```
s = f.readline()
```
lecture ligne suivante

*fichier texte → lecture / écriture de **chaînes** uniquement, convertir de/vers le type désiré*

```
f.close()
```
ne pas oublier de refermer le fichier après son utilisation !

Fermeture automatique Pythonesque : `with open(…) as f:`

très courant : boucle itérative de lecture des lignes d'un fichier texte :

```
for ligne in f :
    bloc de traitement de la ligne
```

Génération de séquences d'entiers

très utilisé pour les boucles itératives for

```
range([début,] fin [,pas])
```
par défaut 0 — non compris

```
range(5)      → 0 1 2 3 4
range(3,8)    → 3 4 5 6 7
range(2,12,3) → 2 5 8 11
```

`range` retourne un « générateur », faire une conversion en liste pour voir les valeurs, par exemple:

```
print(list(range(4)))
```

Définition de fonction

nom de la fonction (identificateur)

paramètres nommés

```
def nomfct(p_x,p_y,p_z):
    """documentation"""
    # bloc instructions, calcul de res, etc.
    return res
```
valeur résultat de l'appel. si pas de résultat calculé à retourner : `return None`

les paramètres et toutes les variables de ce bloc n'existent que *dans* le bloc et *pendant* l'appel à la fonction (« *boite noire* »)

Appel de fonction

```
r = nomfct(3,i+2,2*i)
```
un argument par paramètre

récupération du résultat retourné (si nécessaire)

Formatage de chaînes

directives de formatage — valeurs à formater

```
"modele{} {} {}".format(x,y,r) → str
"{sélection:formatage!conversion}"
```

- **Sélection** :
  `2`
  `x`
  `0.nom`
  `4[clé]`
  `0[2]`
- **Formatage** :

Exemples:

```
"{:+2.3f}".format(45.7273)
→'+45.727'
"{1:>10s}".format(8,"toto")
→'      toto'
"{!r}".format("L'ame")
→'"L\'ame"'
```

car-rempl. *alignement* *signe* *larg.mini*`.`*précision~larg.max* *type*

`<` `>` `^` `=` `+` `-` *espace* `0` au début pour remplissage avec des 0

entiers: `b` binaire, `c` caractère, `d` décimal (défaut), `o` octal, `x` ou `X` hexa…

flottant: `e` ou `E` exponentielle, `f` ou `F` point fixe, `g` ou `G` approprié (défaut)

`%` pourcentage

chaîne : `s` …

- **Conversion** : `s` (texte lisible) ou `r` (représentation littérale)

Abrégé Dense Python 3.2

Abrégé nécessairement incomplet pour tenir sur une feuille, voir sur http://docs.python.org/py3k.

Symbolique de l'Abrégé

♂ instructions optionnelles, ↱ instruction répétables, ⚷ valeur immutable (non modifiable), --> conteneur ordonné (⇝ non ordonné), `constante`, *variable*, `type`, `fonction` & `.méthode`, *paramètre*, *[,paramètre optionnel]*, `mot_clé`, `littéral`, `module`, *fichier*.

Introspection & Aide

`help`(*[objet ou "sujet"]*)
`id`(*objet*) `dir`(*[objet]*) `vars`(*[objet]*)
`locals`() `globals`()

Accès Qualifiés

Séparateur `.` entre un **espace de noms** et un **nom** dans cet espace. Espaces de noms : objet, classe, fonction, module, package…. Exemples :

```
math.sin(math.pi)        f.__doc__
MaClasse.nbObjets()
point.x                  rectangle.largeur()
```

Types de Base

non défini⚷ : `None`

Booléen ⚷: `bool` `True` / `False`
`bool`(*x*) ➙ `False` si *x* nul ou vide

Entier ⚷: `int` `0` `165` `-57`
binaire:`0b101` octal:`0o700` hexa:`0xf3e`
`int`(*x[,base]*) `.bit_length`()

Flottant ⚷: `float` `0.0` `-13.2e-4`
`float`(*x*) `.as_integer_ratio`()

Complexe ⚷: `complex` `0j` `-1.2e4+9.4j`
`complex`(*re[,img]*) `.real` `.imag`
`.conjugate`()

Chaîne ⚷-->: `str` `''` `'toto'` `"toto"`
`"""multiligne toto"""`
`str`(*x*) `repr`(*x*)

Identificateurs, Variables & Affectation

Identificateurs : `[a-zA-Z_]` suivi d'un ou plusieurs `[a-zA-Z0-9_]`, accents et caractères alphabétiques non latins autorisés (mais à éviter).

nom `=` *expression*
nom1`,`*nom2*…`,`*nomN* `=` *séquence*
☞ *séquence* contenant N éléments
nom1 `=` *nom2…* `=` *nomX* `=` *expression*
☞ éclatement séquence: *premier*`,`*`*suite`*`=`*séquence*
☞ incrémentation : *nom*`=`*nom*`+`*expression*
⬌ affectation augmentée : *nom*`+=`*expression* (avec les autres opérateurs aussi)
☞ suppression : `del` *nom*

Conventions Identificateurs

Détails dans PEP 8 "Style Guide for Python"

`UNE_CONSTANTE`	majuscules
`unevarlocale`	minuscules sans _
`une_var_globale`	minuscules avec _
`une_fonction`	minuscules avec _
`une_methode`	minuscules avec _
`UneClasse`	titré
`UneExceptionError`	titré avec Error à la fin
`unmodule`	minuscules plutôt sans _
`unpackage`	minuscules plutôt sans _

Éviter `l O I` (l min, o maj, i maj) seuls.

`_xxx`	usage interne
`__xxx`	transformé _Classe__xxx
`__xxx__`	nom spécial réservé

Opérations Logiques

a`<`*b* *a*`<=`*b* *a*`>=`*b* *a*`>`*b* a=b➙*a*`==`*b* a≠b➙*a*`!=`*b*
`not` *a* *a* `and` *b* *a* `or` *b* (*expr*)
☞ combinables : `12<`*x*`<=34`

Maths

`-`*a* *a*`+`*b* *a*`-`*b* *a*`*`*b* *a*`/`*b* a^b➙*a*`**`*b* (*expr*)
division euclidienne a=b.q+r ➙ *q*=*a*`//`*b* et *r*=*a*`%`*b* et *q,r*=`divmod`(*a,b*)
|x|➙`abs`(*x*) x^y%z➙`pow`(*x,y[,z]*) `round`(*x[,n]*)
☞ fonctions/données suivantes dans le module `math`
e *pi* `ceil`(x) `floor`(*x*) `trunc`(*x*)
e^x➙`exp`(*x*) `log`(*x*) √➙`sqrt`(*x*)
`cos`(*x*) `sin`(*x*) `tan`(*x*) `acos`(*x*) `asin`(*x*)
`atan`(*x*) `atan2`(*x,y*) `hypot`(*x,y*)
`cosh`(*x*) `sinh`(*x*)…
☞ fonctions suivantes dans le module `random`
`seed`(*[x]*) `random`() `randint`(*a,b*)
`randrange`(*[déb],fin[,pas]*) `uniform`(*a,b*)
`choice`(*seq*) `shuffle`(*x[,rnd]*) `sample`(*pop,k*)

Manipulations de bits

(sur les entiers) *a*`<<`*b* *a*`>>`*b* *a*`&`*b* *a*`|`*b* *a*`^`*b*

Chaîne

Échappements : `\`
`\\` ➙ `\` `\'` ➙ `'` `\"` ➙ `"`
`\n` ➙ nouvelle ligne `\t` ➙ tabulation
`\N{`*nom*`}` ➙ unicode *nom*
`\x`*hh* ➙ *hh* hexa `\0`*oo* ➙ *oo* octal
`\u`*hhhh* et `\U`*hhhhhhhh* ➙ unicode hexa *hhhh*
☞ préfixe `r`, désactivation du `\` : `r"\n"` ➙ `\n`

Formatage : `"{`*modèle*`}".format`(*données…*)

```
"{} {}".format(3,2)
"{1} {0} {0}".format(3,9)
"{x} {y}".format(y=2,x=5)
"{0!r} {0!s}".format("texte\n")
"{0:b}{0:o}{0}{0:x}".format(100)
"{0:0.2f}{0:0.3g}{0:.1e}".format(1.45)
```

Opérations
s`*`*n* (répétition) *s1*`+`*s2* (concaténation) `*= +=`
`.split`(*[sep[,n]]*) `.join`(*iterable*)
`.splitlines`(*[keepend]*) `.partition`(*sep*)
`.replace`(*old,new[,n]*) `.find`(*s[, déb[,fin]]*)
`.count`(*s[, déb[,fin]]*) `.index`(*s[, déb[,fin]]*)
`.isdigit`() & Co `.lower`() `.upper`()
`.strip`(*[chars]*)
`.startswith`(*s[,déb[,fin]]*)
`.endsswith`(*s[,start[,end]]*)
`.encode`(*[enc[, err]]*)
`ord`(*c*) `chr`(*i*)

Expression Conditionnelle

Évaluée comme une valeur.
expr1 `if` *condition* `else` *expr2*

Contrôle de Flux

☞ blocs d'instructions délimités par l'indentation (idem fonctions, classes, méthodes). Convention 4 espaces - régler l'éditeur.

Alternative Si
```
if condition1:
    # bloc exécuté si condition1 est vraie
elif condition2: ♂↱
    # bloc exécuté si condition2 est vraie
else: ♂
    # bloc exécuté si toutes conditions fausses
```

Boucle Parcours De Séquence
```
for var in itérable:
    # bloc exécuté avec var valant tour à tour
    # chacune des valeurs de itérable
else: ♂
    # exécuté après, sauf si sortie du for par break
```
☞ *var* à plusieurs variables: `for` *x,y,z* `in`…
☞ *var* index,valeur: `for` *i,v* `in enumerate`(…)
☞ *itérable : voir Conteneurs & Itérables*

Boucle Tant Que
```
while condition:
    # bloc exécuté tant que condition est vraie
else: ♂
    # exécuté après, sauf si sortie du while par break
```

Rupture De Boucle : `break`
Sortie immédiate de la boucle, sans passer par le bloc else.

Saut De Boucle : `continue`
Saut immédiat en début de bloc de la boucle pour exécuter l'itération suivante.

Traitement D'erreurs: Exceptions
```
try:
    # bloc exécuté dans les cas normaux
except exc as e: ↱
    # bloc exécuté si une erreur de type exc est
    # détectée
else:
    # bloc exécuté en cas de sortie normale du try
finally:
    # bloc exécuté dans tous les cas
```
☞ *exc* pour n types : `except` (*exc1, exc2…, excn*)
☞ `as` *e* optionnel, récupère l'exception
⚠ détecter des exceptions précises (ex. ValueError) et non génériques (ex. Exception).

Levée D'exception (situation d'erreur)
`raise` *exc* *[args]*
`raise` ➙ ⚠ propager l'exception
Quelques classes d'exceptions : `Exception` - `ArithmeticError` - `ZeroDivisionError` - `IndexError` - `KeyError` - `AttributeError` - `IOError` - `ImportError` - `NameError` - `SyntaxError` - `TypeError` - `NotImplementedError`…

Contexte Géré
`with` *garde*() `as` *v* ♂`:`
    # Bloc exécuté dans un contexte géré

Définition et Appel de Fonction

```
def nomfct(x,y=4,*args,**kwargs):
    # le bloc de la fonction ou à défaut pass
    return ret_expression ♂
```

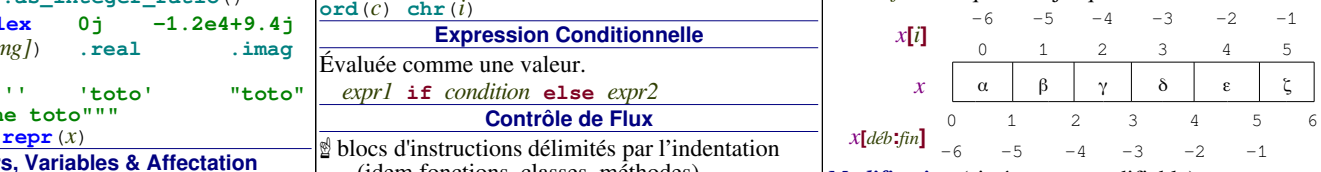
x: paramètre simple
y: paramètre avec valeur par défaut
args: paramètres variables par ordre (`tuple`)
kwargs: paramètres variables nommés (`dict`)
ret_expression: `tuple` ➙ retour de plusieurs valeurs

Appel
res = *nomfct*(*expr*`,`*param*`=`*expr*`,`*`*tuple*`,`*`**`*dict*)

Fonctions Anonymes
`lambda` *x,y*`:` *expression*

Séquences & Indexation

☞ pour tout conteneur ordonné à accès direct.
i^e ***Élément*** : *x*`[`*i*`]`
Tranche (slice) : *x*`[`*déb*`:`*fin*`]` *x*`[`*déb*`:`*fin*`:`*pas*`]`
☞ *i*, *déb*, *fin*, *pas* entiers positifs ou négatifs
☞ *déb*/*fin* manquant ➙ jusqu'au bout



Modification (si séquence modifiable)
x`[`*i*`]=`*expression* *x*`[`*déb*`:`*fin*`]=`*itérable*
`del` *x*`[`*i*`]` `del` *x*`[`*déb*`:`*fin*`]`

Conteneurs & Itérables

Un *itérable* fournit les valeurs l'une après l'autre. Ex : conteneurs, vues sur dictionnaires, objets itérables, fonctions générateurs…

Générateurs (calcul des valeurs lorsque nécessaire)
`range`(*[déb,]fin[,pas]*)
`(` *expr* `for` *var* `in` *iter* ↱ `if` *cond* ♂ `)`

Opérations Génériques
v `in` *conteneur* *v* `not in` *conteneur*
`len`(*conteneur*) `enumerate`(*iter[,déb]*)
`iter`(*o[,sent]*) `all`(*iter*) `any`(*iter*)
`filter`(*fct,iter*) `map`(*fct,iter,…*)
`max`(*iter*) `min`(*iter*) `sum`(*iter[,déb]*)
`reversed`(*seq*) `sorted`(*iter[,k][,rev]*)
Sur séquences : `.count`(*x*) `.index`(*x[,i[,j]]*)

Chaîne ⚷--> : (séquence de caractères)
☞ cf. types `bytes`, `bytearray`, `memoryview` pour manipuler des octets (+notation `b"octets"`).

Liste --> : `list` `[]` `[1,'toto',3.14]`
`list`(*iterable*) `.append`(*x*)
`.extend`(*iterable*) `.insert`(*i,x*) `.pop`(*[i]*)
`.remove`(*x*) `.reverse`() `.sort`()
`[` *expr* `for` *var* `in` *iter* ↱ `if` *cond* ♂ `]`

Tuple ⚷--> : `tuple` `()` `(9,'x',36)` `(1,)`
`tuple`(*iterable*) `9,'x',36` `1,`

Ensemble ⇝ : `set` `{1,'toto',42}`
`set`(*iterable*) ⚷⇝ : `frozenset`(*iterable*)
`.add`(*x*) `.remove`(*x*) `.discard`(*x*)
`.copy`() `.clear`() `.pop`()
∪➙`|`, ∩➙`&`, diff➙`-`, diff.sym➙`^`, ⊂…➙`<`…
`|= &= -= ^=` …

Dictionnaire (tableau associatif, map) ⇝ : `dict`
`{}` `{1:'one',2:'two'}`
`dict`(*iterable*) `dict`(*a*`=`2`,`*b*`=`4)
`dict.fromkeys`(*seq[,val]*)
d`[`*k*`]=`*expr* *d*`[`*k*`]` `del` *d*`[`*k*`]`
`.update`(*iter*) `.keys`() `.values`()
`.items`() `.pop`(*k[,def]*) `.popitem`()
`.get`(*k[,def]*) `.setdefault`(*k[,def]*)
`.clear`() `.copy`()
☞ `items`, `keys`, `values` "vues" itérables

Entrées/Sorties & Fichiers

`print`("x=",*x[,y…][,sep=…][,end=…][,file=…]*)
`input`(`"Age ? "`) ➙ `str`
☞ transtypage explicite en `int` ou `float` si besoin.

Fichier : *f*=`open`(*nom*[,*mode*][,*encoding*=…])
mode : `'r'` lecture (défaut) `'w'` écriture `'a'` ajout
`'+'` lecture écriture `'b'` mode binaire…
encoding : `'utf-8'` `'latin1'` `'ascii'`…
`.write`(*s*) `.read`([*n*]) `.readline`()
`.flush`() `.close`() `.readlines`()
Boucle sur lignes : `for` *line* `in` *f* :…
Contexte géré (close) : `with open`(…) `as` *f*:
dans le module ***os*** (voir aussi ***os.path***):
`getcwd`() `chdir`(*chemin*) `listdir`(*chemin*)
Paramètres ligne de commande dans ***sys***.*argv*

Modules & Packages

Module : fichier script extension *.py* (et modules compilés en C). Fichier *toto.py* ➡ module `toto`.
Package : répertoire avec fichier *__init__.py*. Contient des fichiers modules.
Recherchés dans le PYTHONPATH, voir liste ***sys***.*path*.

Modèle De Module :

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
"""Documentation module - cf PEP257"""
# Fichier: monmodule.py
# Auteur: Joe Student
# Import d'autres modules, fonctions…
import math
from random import seed,uniform
# Définitions constantes et globales
MAXIMUM = 4
lstFichiers = []
# Définitions fonctions et classes
def f(x):
    """Documentation fonction"""
    …
class Convertisseur(object):
    """Documentation classe"""
    nb_conv = 0 # var de classe
    def __init__(self,a,b):
        """Documentation init"""
        self.v_a = a # var d'instance
        …
    def action(self,y):
        """Documentation méthode"""
        …
# Auto-test du module
if __name__ == '__main__':
    if f(2) != 4: # problème
    …
```

Import De Modules / De Noms

```
import monmodule
from monmodule import f,MAXIMUM
from monmodule import *
from monmodule import f as fct
```

Pour limiter l'effet `*`, définir dans ***monmodule*** :

```
__all__ = [ "f", "MAXIMUM"]
```

Import via package :

```
from os.path import dirname
```

Définition de Classe

Méthodes spéciales, noms réservées `__xxxx__`.

```
class NomClasse([claparent]) :
    # le bloc de la classe
    variable_de_classe = expression
    def __init__(self[,params…]) :
        # le bloc de l'initialiseur
        self.variable_d_instance = expression
    def __del__(self) :
        # le bloc du destructeur
    @staticmethod        # @ ↔ "décorateur"
    def fct([,params…]) :
        # méthode statique (appelable sans objet)
```

Tests D'appartenance

`isinstance`(*obj*,*classe*)
`isssubclass`(*sousclasse*,*parente*)

Création d'Objets

Utilisation de la classe comme une fonction, paramètres passés à l'initialiseur `__init__`.

obj = `NomClasse`(*params…*)

Méthodes spéciales Conversion

```
def __str__(self) :
    # retourne chaîne d'affichage
def __repr__(self) :
    # retourne chaîne de représentation
def __bytes__(self) :
    # retourne objet chaîne d'octets
def __bool__(self) :
    # retourne un booléen
def __format__(self,spécif_format) :
    # retourne chaîne suivant le format spécifié
```

Méthodes spéciales Comparaisons

Retournent `True`, `False` ou `NotImplemented`.

x<*y* ➡ `def __lt__`(*self*,*y*) :
x<=*y* ➡ `def __le__`(*self*,*y*) :
x==*y* ➡ `def __eq__`(*self*,*y*) :
x!=*y* ➡ `def __ne__`(*self*,*y*) :
x>*y* ➡ `def __gt__`(*self*,*y*) :
x>=*y* ➡ `def __ge__`(*self*,*y*) :

Méthodes spéciales Opérations

Retournent un nouvel objet de la classe, intégrant le résultat de l'opération, ou `NotImplemented` si ne peuvent travailler avec l'argument *y* donné.

x ➡ *self*

x+*y* ➡ `def __add__`(*self*,*y*) :
x−*y* ➡ `def __sub__`(*self*,*y*) :
*x***y* ➡ `def __mul__`(*self*,*y*) :
x/*y* ➡ `def __truediv__`(*self*,*y*) :
x//*y* ➡ `def __floordiv__`(*self*,*y*) :
x%*y* ➡ `def __mod__`(*self*,*y*) :
`divmod`(*x*,*y*) ➡ `def __divmod__`(*self*,*y*) :
*x****y* ➡ `def __pow__`(*self*,*y*) :
`pow`(*x*,*y*,*z*) ➡ `def __pow__`(*self*,*y*,*z*) :
x<<*y* ➡ `def __lshift__`(*self*,*y*) :
x>>*y* ➡ `def __rshift__`(*self*,*y*) :
x&*y* ➡ `def __and__`(*self*,*y*) :
x|*y* ➡ `def __or__`(*self*,*y*) :
x^*y* ➡ `def __xor__`(*self*,*y*) :
−*x* ➡ `def __neg__`(*self*) :
+*x* ➡ `def __pos__`(*self*) :
`abs`(*x*) ➡ `def __abs__`(*self*) :
~*x* ➡ `def __invert__`(*self*) :

Méthodes suivantes appelées ensuite avec *y* si *x* ne supporte pas l'opération désirée.

y ➡ *self*

x+*y* ➡ `def __radd__`(*self*,*x*) :
x−*y* ➡ `def __rsub__`(*self*,*x*) :
*x***y* ➡ `def __rmul__`(*self*,*x*) :
x/*y* ➡ `def __rtruediv__`(*self*,*x*) :
x//*y* ➡ `def __rfloordiv__`(*self*,*x*) :
x%*y* ➡ `def __rmod__`(*self*,*x*) :
`divmod`(*x*,*y*) ➡ `def __rdivmod__`(*self*,*x*) :
*x****y* ➡ `def __rpow__`(*self*,*x*) :
x<<*y* ➡ `def __rlshift__`(*self*,*x*) :
x>>*y* ➡ `def __rrshift__`(*self*,*x*) :
x&*y* ➡ `def __rand__`(*self*,*x*) :
x|*y* ➡ `def __ror__`(*self*,*x*) :
x^*y* ➡ `def __rxor__`(*self*,*x*) :

Méthodes spéciales Affectation augmentée

Modifient l'objet *self* auquel elles s'appliquent.

x ➡ *self*

x+=*y* ➡ `def __iadd__`(*self*,*y*) :
x−=*y* ➡ `def __isub__`(*self*,*y*) :
*x**=*y* ➡ `def __imul__`(*self*,*y*) :
x/=*y* ➡ `def __itruediv__`(*self*,*y*) :
x//=*y* ➡ `def __ifloordiv__`(*self*,*y*) :
x%=*y* ➡ `def __imod__`(*self*,*y*) :
*x***=*y* ➡ `def __ipow__`(*self*,*y*) :
x<<=*y* ➡ `def __ilshift__`(*self*,*y*) :
x>>=*y* ➡ `def __irshift__`(*self*,*y*) :
x&=*y* ➡ `def __iand__`(*self*,*y*) :
x|=*y* ➡ `def __ior__`(*self*,*y*) :
x^=*y* ➡ `def __ixor__`(*self*,*y*) :

Méthodes spéciales Conversion numérique

Retournent la valeur convertie.

x ➡ *self*

`complex`(*x*) ➡ `def __complex__`(*self*) :
`int`(*x*) ➡ `def __int__`(*self*) :
`float`(*x*) ➡ `def __float__`(*self*) :
`round`(*x*,*n*) ➡ `def __round__`(*self*,*n*) :
`def __index__`(*self*) :
    # retourne un entier utilisable comme index

Méthodes spéciales Accès aux attributs

Accès par *obj*.*nom*. Exception `AttributeError` si attribut non trouvé.

obj ➡ *self*

```
def __getattr__(self,nom) :
    # appelé si nom non trouvé en attribut existant,
def __getattribute__(self,nom) :
    # appelé dans tous les cas d'accès à nom
def __setattr__(self,nom,valeur) :
def __delattr__(self,nom) :
def __dir__(self) : # retourne une liste
```

Accesseurs

Property

```
class C(object) :
  def getx(self) : …
  def setx(self,valeur) : …
  def delx(self) : …
  x = property(getx,setx,delx,"docx")
  # Plus simple, accesseurs à y, avec des décorateurs
  @property
  def y(self) :    # lecture
    """docy"""
  @y.setter
  def y(self,valeur) :    # modification
  @y.deleter
  def y(self) :    # suppression
```

Protocole Descripteurs

o.x ➡ `def __get__`(*self*,*o*,*classe_de_o*) :
o.*x*=*v* ➡ `def __set__`(*self*,*o*,*v*) :
`del` *o*.*x* ➡ `def __delete__`(*self*,*o*):

Méthode spéciale Appel de fonction

Utilisation d'un objet comme une fonction (callable) :

o(*params*) ➡ `def __call__`(*self*[,*params…*]):

Méthode spéciale Hachage

Pour stockage efficace dans `dict` et `set`.

`hash`(*o*) ➡ `def __hash__`(*self*) :

Définir à `None` si objet non *hachable*.

Méthodes spéciales Conteneur

o ➡ *self*

```
len(o) ➡ def __len__(self) :
o[clé] ➡ def __getitem__(self,clé) :
o[clé]=v ➡ def __setitem__(self,clé,v) :
del o[clé] ➡ def __delitem__(self,clé) :
for i in o: ➡ def __iter__(self) :
    # retourne un nouvel itérateur sur le conteneur
reversed(o) ➡ def __reversed__(self) :
x in o ➡ def __contains__(self,x) :
```

Pour la notation `[`*déb*`:`*fin*`:`*pas*`]`, un objet de type `slice` est donné comme valeur de *clé* aux méthodes conteneur.

Tranche : `slice`(*déb*,*fin*,*pas*)
.*start* .*stop* .*step* `.indices`(*longueur*)

Méthodes spéciales Itérateurs

```
def __iter__(self) :# retourne self
def __next__(self) :# retourne l'élément suivant
```

Si plus d'élément, levée exception `StopIteration`.

Méthodes spéciales Contexte Géré

Utilisées pour le `with`.

```
def __enter__(self) :
    # appelée à l'entrée dans le contexte géré
    # valeur utilisée pour le as du contexte
def __exit__(self,etype,eval,tb) :
    # appelée à la sortie du contexte géré
```

Méthodes spéciale Métaclasses

```
__prepare__ = callable
def __new__(cls[,params…]) :
    # allocation et retour d'un nouvel objet cls
```

`isinstance`(*o*,*cls*) ➡ `def __instancecheck__`(*cls*,*o*):
`isssubclass`(*sousclasse*,*cls*) ➡ `def __subclasscheck__`(*cls*,*sousclasse*):

Générateurs

Calcul des valeurs lorsque nécessaire (ex.: `range`).
Fonction générateur, contient une instruction
`yield`. `yield` *expression*
`yield from` ***séquence***
variable = (`yield` *expression*) transmission de valeurs au générateur.
Si plus de valeur, levée exception `StopIteration`.

Contrôle Fonction Générateur

générateur.`__next__`()
générateur.`send`(*valeur*)
générateur.`throw`(*type*[,*valeur*[,*traceback*]])
générateur.`close`()

Index

Glossaire
Lexique bilingue

>>>
Invite Python par défaut dans un shell interactif. Souvent utilisée dans les exemples de code extraits de sessions de l'interpréteur Python.

...
Invite Python par défaut dans un shell interactif, utilisée lorsqu'il faut entrer le code d'un bloc indenté ou à l'intérieur d'une paire de parenthèses, crochets ou accolades.

2to3
Un outil qui essaye de convertir le code Python 2.x en code Python 3.x en gérant la plupart des incompatibilités qu'il peut détecter.
`2to3` est disponible dans la bibliothèque standard `lib2to2` ; un point d'entrée autonome est `Tool/scripts/2to3`. Voir **2to3 – Automated Python 2 to 3 code translation**.

abstract base class *ABC* (classe de base abstraite)
Complète le *duck-typing* en fournissant un moyen de définir des interfaces alors que d'autres techniques (comme `hasattr()`) sont plus lourdes. Python fournit de base plusieurs *ABC* pour les structures de données (module `collections`), les nombres (module `numbers`) et les flux (module `io`). Vous pouvez créer votre propre *ABC* en utilisant le module `abc`.

argument (argument) [cf. p. 45]
Valeur passée à une fonction ou une méthode, affectée à une variable nommée locale au corps de la fonction. Une fonction ou une méthode peut avoir à la fois des arguments par position et avec des valeurs par défaut. Ces arguments peuvent être de multiplicité variable : * accepte ou fournit plusieurs arguments par position dans une liste, tandis que ** joue le même rôle en utilisant les valeurs par défaut dans un dictionnaire.
On peut passer toute expression dans la liste d'arguments, et la valeur évaluée est transmise à la variable locale.

attribute (attribut) [cf. p. 65]
Valeur associée à un objet référencé par un nom et une expression pointée. Par exemple, l'attribut `a` d'un objet `o` peut être référencé `o.a`.

BDFL *Benevolent Dictator For Life* (Dictateur Bienveillant à Vie) [cf. p. 1]
Amical surnom de Guido van Rossum, le créateur de Python.

bytecode (bytecode ou langage intermédiaire) [cf. p. 3]
Le code source Python est compilé en bytecode, représentation interne d'un programme Python dans l'interpréteur. Le bytecode est également rangé dans des fichiers `.pyc` et `.pyo`, ainsi l'exécution d'un même fichier est plus rapide les fois ultérieures (la compilation du source en bytecode peut être évitée). On dit que le bytecode tourne sur une **machine virtuelle** qui, essentiellement, se réduit à une collection d'appels des routines correspondant à chaque code du bytecode.

class (classe) [cf. p. 65]
Modèle permettant de créer ses propres objets. Les définitions de classes contiennent normalement des définitions de méthodes qui opèrent sur les instances de classes.

coercion (coercition ou transtypage) [cf. p. 19]
Conversion implicite d'une instance d'un type dans un autre type dans une opération concernant deux arguments de types compatibles. Par exemple, `int(3.15)` convertit le nombre flottant `3.15` en l'entier `3`, mais dans `3+4.5`, chaque argument est d'un type différent (l'un `int` et l'autre `float`) et tous deux doivent être convertis dans le même type avant d'être additionnés, sinon une exception `TypeError` sera lancée. Sans coercition, tous les arguments, même de types compatibles, doivent être normalisés à la même valeur par le programmeur, par exemple, `float(3)+4.5` au lieu de simplement `3+4.5`.

complex number (nombre complexe) [cf. p. 12]

Une extension du système familier des nombres réels dans laquelle tous les nombres sont exprimés comme la somme d'une partie réelle et une partie imaginaire. Les nombres imaginaires sont des multiples réels de l'unité imaginaire (la racine carrée de -1), souvent écrite *i* par les mathématiciens et *j* par les ingénieurs. Python a un traitement incorporé des nombres complexes, qui sont écrits avec cette deuxième notation ; la partie imaginaire est écrite avec un suffixe `j`, par exemple `3+1j`. Pour avoir accès aux équivalents complexes des éléments du module `math` utilisez le module `cmath`. L'utilisation des nombres complexes est une possibilité mathématique assez avancée. Si vous n'êtes pas certain d'en avoir besoin vous pouvez les ignorer sans risque.

context manager (gestionnaire de contexte) [cf. p. 85]

Objet qui contrôle l'environnement indiqué par l'instruction `with` et qui définit les méthodes `__enter__()` et `__exit__()`. Voir la PEP 343.

CPython (Python classique) [cf. p. 4]

Implémentation canonique du langage de programmation Python. Le terme *CPython* est utilisé dans les cas où il est nécessaire de distinguer cette implémentation d'autres comme Jython ou IronPython.

decorator (décorateur) [cf. p. 92]

Fonction retournant une autre fonction habituellement appliquée comme une transformation utilisant la syntaxe `@wrapper`.

`classmethod()` et `staticmethod()` sont des exemples classiques de décorateurs.
Les deux définitions de fonctions suivantes sont sémantiquement équivalentes :

```
def f(...):
    ...
f = staticmethod(f)
```

```
@staticmethod
def f(...):
    ...
```

Un concept identique existe pour les classes, mais est moins utilisé. Voir la documentation **function definition** et **class definition** pour plus de détails sur les décorateurs.

descriptor (descripteur)

Tout objet qui définit les méthodes `__get__()`, `__set__()` ou `__delete__()`. Lorsqu'un attribut d'une classe est un descripteur, un comportement spécifique est déclenché lors de la consultation de l'attribut. Normalement, écrire `a.b` consulte l'objet `b` dans le dictionnaire de la classe de `a`, mais si `b` est un descripteur, la méthode `__get__()` est appelée. Comprendre les descripteurs est fondamental pour la compréhension profonde de Python, car ils sont à la base de nombreuses caractéristiques, comme les fonctions, les méthodes, les propriétés, les méthodes de classe, les méthodes statiques et les références aux super-classes.
Pour plus d'informations sur les méthodes des descripteurs, voir Implementing Descriptors.

dictionary (dictionnaire) [cf. p. 34]

Une table associative, dans laquelle des clés arbitraires sont associées à des valeurs. L'utilisation des objets `dict` ressemble beaucoup à celle des objets `list`, mais les clés peuvent être n'importe quels objets ayant une fonction `__hash__()`, non seulement des entiers. Ces tables sont appelées `hash` en Perl.

docstring (chaîne de documentation) [cf. p. 43]

Chaîne littérale apparaissant comme première expression d'une classe, d'une fonction ou d'un module. Bien qu'ignorée à l'exécution, elle est reconnue par le compilateur et incluse dans l'attribut `__doc__` de la classe, de la fonction ou du module qui la contient. Depuis qu'elle est disponible via l'introspection, c'est l'endroit canonique pour documenter un objet.

duck-typing (typage « comme un canard ») [cf. p. 96]

Style de programmation pythonique dans lequel on détermine le type d'un objet par inspection de ses méthodes et attributs plutôt que par des relations explicites à des types (« s'il ressemble à un canard et fait *coin-coin* comme un canard alors ce doit être un canard »). En mettant l'accent sur des interfaces plutôt que sur des types spécifiques, on améliore la flexibilité du code en permettant la substitution polymorphe. Le *duck-typing* évite les tests qui utilisent `type()` ou `isinstance()` (notez

cependant que le *duck-typing* doit être complété par l'emploi des classes de base abstraites). À la place, il emploie des tests comme `hasattr()` et le style de programmation *EAFP*.

EAFP (*Easier to ask for forgiveness than permission*, ou « plus facile de demander pardon que la permission ») Ce style courant de programmation en Python consiste à supposer l'existence des clés et des attributs nécessaires à l'exécution d'un code et à attraper les exceptions qui se produisent lorsque de telles hypothèses se révèlent fausses. C'est un style propre et rapide, caractérisé par la présence de nombreuses instructions `try` et `except`. Cette technique contraste avec le style *LBYL*, courant dans d'autres langages comme le C.

expression (expression) [cf. p. 8] Fragment de syntaxe qui peut être évalué. Autrement dit, une expression est une accumulation d'éléments d'expression comme des littéraux, des noms, des accès aux attributs, des opérateurs ou des appels à des fonctions retournant une valeur. À l'inverse de beaucoup d'autres langages, toutes les constructions de Python ne sont pas des expressions. Les instructions ne peuvent pas être utilisées comme des expressions (par exemple `if`). Les affectations sont aussi des instructions, pas des expressions.

extension module (module d'extension) Module écrit en C ou en C++, utilisant l'API C de Python, qui interagit avec le cœur du langage et avec le code de l'utilisateur.

finder Objet qui essaye de trouver le *loader* (chargeur) d'un module. Il doit implémenter une méthode nommée `find_module()`. Voir la PEP 302 pour des détails et **importlib.abc.Finder** pour une classe de base abstraite.

floor division (division entière) [cf. p. 9] Division mathématique qui laisse tomber le reste. L'opérateur de division entière est `//`. Par exemple, l'expression `11//4` est évaluée à `2`, par opposition à la division flottante qui retourne `2.75`.

function (fonction) [cf. p. 43] Suite d'instructions qui retourne une valeur à l'appelant. On peut lui passer zéro ou plusieurs arguments qui peuvent être utilisés dans le corps de la fonction. Voir aussi **argument** et **method**.

__future__ Un pseudo-module que les programmeurs peuvent utiliser pour permettre les nouvelles fonctionnalités du langage qui ne sont pas compatibles avec l'interpréteur couramment employé.
En important `__future__` et en évaluant ses variables, vous pouvez voir à quel moment une caractéristique nouvelle a été ajoutée au langage et quand elle est devenue la fonctionnalité par défaut :

```
>>> import __future__

>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection (gestion automatique de la mémoire) Processus de libération de la mémoire quand elle n'est plus utilisée. Python exécute cette gestion en comptant les références et en détectant et en cassant les références cycliques.

generator (fonction générateur) [cf. p. 89] Une fonction qui renvoie un itérateur. Elle ressemble à une fonction normale, excepté que la valeur de la fonction est rendue à l'appelant en utilisant une instruction `yield` au lieu d'une instruction `return`. Les fonctions générateurs contiennent souvent une ou plusieurs boucles `for` ou `while` qui « cèdent » des éléments à l'appelant. L'exécution de la fonction est stoppée au niveau du mot-clé `yield`, en renvoyant un résultat, et elle est reprise lorsque l'élément suivant est requis par un appel de la méthode `next()` de l'itérateur.

generator expression (expression générateur) [cf. p. 90] Une expression qui renvoie un générateur. Elle ressemble à une expression normale suivie d'une expression `for` définissant une variable de contrôle, un intervalle et une expression `if` facultative. Toute cette expression combinée produit des valeurs pour une fonction englobante :

```
>>> sum(i*i for i in range(10)) # somme des carrs 0+1+4+ ... 81 = 285
285
```

GIL

cf. **global interpreter lock**.

global interpreter lock (verrou global de l'interpréteur)

Le verrou utilisé par les *threads* Python pour assurer qu'un seul *thread* tourne dans la **machine virtuelle CPython** à un instant donné. Il simplifie Python en garantissant que deux processus ne peuvent pas accéder en même temps à une même mémoire. Bloquer l'interpréteur tout entier lui permet d'être *multi-thread* aux frais du parallélisme du système environnant. Des efforts ont été faits par le passé pour créer un interpréteur *free-threaded* (où les données partagées sont verrouillées avec une granularité fine), mais les performances des programmes en souffraient considérablement, y compris dans le cas des programmes *mono-thread*.

hashable (hachable)

Un objet est hachable s'il a une valeur de hachage constante au cours de sa vie (il a besoin d'une méthode `__hash__()`) et s'il peut être comparé à d'autres objets (il a besoin d'une méthode `__eq__()`). Les objets hachables comparés égaux doivent avoir la même valeur de hachage.
L'« hachabilité » rend un objet propre à être utilisé en tant que clé d'un dictionnaire ou membre d'un ensemble (`set`), car ces structures de données utilisent la valeur de hachage de façon interne.
Tous les objets de base Python non modifiables (*immutable*) sont hachables, alors que certains conteneurs comme les listes ou les dictionnaires sont modifiables. Les objets instances des classes définies par l'utilisateur sont hachables par défaut ; ils sont tous inégaux (différents) et leur valeur de hachage est leur `id()`.

IDLE [cf. p. 81]

Un environnement de développement intégré pour Python. IDLE est un éditeur basique et un environnement d'interprétation ; il est donné avec la distribution standard de Python. Excellent pour les débutants, il peut aussi servir d'exemple pas trop sophistiqué pour tous ceux qui doivent implémenter une application avec interface utilisateur graphique multi-plate-forme.

immutable (immuable) [cf. p. 15]

Un objet avec une valeur fixe. Par exemple, les nombres, les chaînes, les tuples. De tels objets ne peuvent pas être altérés ; pour changer de valeur un nouvel objet doit être créé. Les objets immuables jouent un rôle important aux endroits où une valeur de hash constante est requise, par exemple pour les clés des dictionnaires.

importer [cf. p. 51]

Objet qui à la fois trouve et charge un module. C'est à la fois un objet *finder* et un objet *loader*.

interactive (interactif) [cf. p. 7]

Python possède un interpréteur interactif, ce qui signifie que vous pouvez essayer vos idées et voir immédiatement les résultats. Il suffit de lancer `python` sans argument (éventuellement en le sélectionnant dans un certain menu principal de votre ordinateur). C'est vraiment un moyen puissant pour tester les idées nouvelles ou pour inspecter les modules et les paquetages (pensez à `help(x)`).

interpreted (interprété) [cf. p. 3]

Python est un langage interprété, par opposition aux langages compilés, bien que cette distinction puisse être floue à cause de la présence du compilateur de bytecode. Cela signifie que les fichiers source peuvent être directement exécutés sans avoir besoin de créer préalablement un fichier binaire exécuté ensuite. Typiquement, les langages interprétés ont un cycle de développement et de mise au point plus court que les langages compilés, mais leurs programmes s'exécutent plus lentement. Voir aussi **interactive**.

iterable (itérable) [cf. p. 25]

Un objet conteneur capable de renvoyer ses membres un par un. Des exemples d'*iterable* sont les types séquences (comme les `list`, les `str`, et les `tuple`) et quelques types qui ne sont pas des séquences, comme les objets `dict`, les objets `file` et les objets de n'importe quelle classe que vous définissez avec une méthode `__iter__()` ou une méthode `__getitem__()`. Les *iterables* peuvent être utilisés dans les boucles `for` et dans beaucoup d'autres endroits où une séquence est requise (`zip()`, `map()`, ...). Lorsqu'un objet *iterable* est passé comme argument à la fonction incorporée `iter()` il renvoie un itérateur. Cet itérateur est un bon moyen pour effectuer un parcours d'un ensemble de valeurs. Lorsqu'on utilise des *iterables*, il n'est généralement pas nécessaire d'appeler la fonction

`iter()` ni de manipuler directement les valeurs en question. L'instruction `for` fait cela automatiquement pour vous, en créant une variable temporaire sans nom pour gérer l'itérateur pendant la durée de l'itération. Voir aussi **iterator**, **sequence**, et **generator**.

iterator (itérateur)
Un objet représentant un flot de données. Des appels répétés à la méthode `__next__()` de l'itérateur (ou à la fonction de base `next()`) renvoient des éléments successifs du flot. Lorsqu'il n'y a plus de données disponibles dans le flot, une exception `StopIteration` est lancée. À ce moment-là, l'objet itérateur est épuisé et tout appel ultérieur de la méthode `next()` ne fait que lancer encore une exception `StopIteration`. Les itérateurs doivent avoir une méthode `__iter__()` qui renvoie l'objet itérateur lui-même. Ainsi un itérateur est itératif et peut être utilisé dans beaucoup d'endroits où les *iterables* sont acceptés ; une exception notable est un code qui tenterait des itérations multiples. Un objet conteneur (comme un objet `list`) produit un nouvel itérateur à chaque fois qu'il est passé à la fonction `iter()` ou bien utilisé dans une boucle `for`. Si on fait cela avec un itérateur on ne récupérera que le même itérateur épuisé utilisé dans le parcours précédent, ce qui fera apparaître le conteneur comme s'il était vide.

keyword argument (argument avec valeur par défaut) [cf. p. 47]
Argument précédé par `variable_name=` dans l'appel. Le nom de la variable désigne le nom local dans la fonction auquel la valeur est affectée. ** est utilisé pour accepter ou passer un dictionnaire d'arguments avec ses valeurs. Voir **argument**.

lambda [cf. p. 98]
Fonction anonyme en ligne ne comprenant qu'une unique expression évaluée à l'appel. Syntaxe de création d'une fonction lambda :

```
lambda[arguments]: expression
```

LBYL (*Look before you leap* ou « regarder avant d'y aller »)
Ce style de code teste explicitement les pré-conditions avant d'effectuer un appel ou une recherche. Ce style s'oppose à l'approche *EAFP* et est caractérisé par la présence de nombreuses instructions `if`.

list (liste) [cf. p. 29]
Séquence Python de base. En dépit de son nom, elle ressemble plus au tableau d'autres langages qu'à une liste chaînée puisque l'accès à ses éléments est en $O(1)$.

list comprehension (liste en compréhension) [cf. p. 88]
Une manière compacte d'effectuer un traitement sur un sous-ensemble d'éléments d'une séquence en renvoyant une liste avec les résultats. Par exemple :

```
result = ["0x%02x" % x for x in range(256) if x % 2 == 0]
```

engendre une liste de chaînes contenant les écritures hexadécimales des nombres pairs de l'intervalle de `0` à `255`. La clause `if` est facultative. Si elle est omise, tous les éléments de l'intervalle `range(256)` seront traités.

loader (chargeur)
Objet qui charge un module. Il doit posséder une méthode `load_module()`. un *loader* est typiquement fourni par un *finder*. Voir la PEP 302 pour les détails et voir **importlib.abc.Loader** pour une classe de base abstraite.

mapping (liste associative) [cf. p. 34]
Un objet conteneur (comme `dict`) qui supporte les recherches par des clés arbitraires en utilisant la méthode spéciale `__getitem__()`.

metaclass
La classe d'une classe. La définition d'une classe crée un nom de classe, un dictionnaire et une liste de classes de base. La métaclasse est responsable de la création de la classe à partir de ces trois éléments. Beaucoup de langages de programmation orientés objets fournissent une implémentation par défaut. Une originalité de Python est qu'il est possible de créer des métaclasses personnalisées. Beaucoup d'utilisateurs n'auront jamais besoin de cela mais, lorsque le besoin apparaît, les métaclasses fournissent des solutions puissantes et élégantes. Elles sont utilisées pour enregistrer les accès aux attributs, pour ajouter des *threads* sécurisés, pour détecter la création d'objets, pour implémenter des singletons et pour bien d'autres tâches.
Plus d'informations peuvent être trouvées dans Customizing class creation.

method (méthode) [cf. p. 68]

Fonction définie dans le corps d'une classe. Appelée comme un attribut d'une instance de classe, la méthode prend l'instance d'objet en tant que premier argument (habituellement nommé `self`). Voir **function** et **nested scope**.

mutable (modifiable)

Les objets modifiables peuvent changer leur valeur tout en conservant leur `id()`. Voir aussi **immutable**.

named tuple (tuple nommé) [cf. p. 55]

Toute classe de pseudo-tuples dont les éléments indexables sont également accessibles par des attributs nommés (par exemple `time.localtime()` retourne un objet pseudo-tuple où l'année est accessible soit par un index comme `t[0]` soit par un attribut nommé comme `t.tm_year`.
Un tuple nommé peut être un type de base comme `time.struct_time` ou il peut être créé par une définition de classe ordinaire. Un tuple nommé peut aussi être créé par la fonction fabrique `collections.nametuple()`. Cette dernière approche fournit automatiquement des caractéristiques supplémentaires comme une représentation auto-documentée, par exemple :

```
>>> Employee(name='jones', title='programmer')
```

namespace (espace de noms) [cf. p. 48]

L'endroit où une variable est conservée. Les espaces de noms sont implémentés comme des dictionnaires. Il y a des espaces de noms locaux, globaux et intégrés et également imbriqués dans les objets. Les espaces de noms contribuent à la modularité en prévenant les conflits de noms. Par exemple, les fonctions `__builtin__.open()` et `os.open()` se distinguent par leur espace de noms. Les espaces de noms contribuent aussi à la lisibilité et la maintenablité en clarifiant quel module implémente une fonction. Par exemple, en écrivant `random.seed()` ou `itertools.izip()` on rend évident que ces fonctions sont implémentées dans les modules `random` et `itertools` respectivement.

nested scope (portée imbriquée) [cf. p. 49]

La possibilité de faire référence à une variable d'une définition englobante. Par exemple, une fonction définie à l'intérieur d'une autre fonction peut faire référence à une variable de la fonction extérieure. Notez que le portées imbriquées fonctionnent uniquement pour la référence aux variables et non pour leur affectation, qui concerne toujours la portée imbriquée. Les variables locales sont lues et écrites dans la portée la plus intérieure ; les variables globales sont lues et écrites dans l'espace de noms global. L'instruction `nonlocal` permet d'écrire dans la portée globale.

new-style class (style de classe nouveau)

Vieille dénomination pour le style de programmation de classe actuellement utilisé. Dans les versions précédentes de Python, seul le style de classe nouveau pouvait bénéficier des nouvelles caractéristiques de Python, comme `__slots__`, les descripteurs, les propriétés, `__getattribute__()`, les méthodes de classe et les méthodes statiques.

object (objet) [cf. p. 65]

Toute donnée comprenant un état (attribut ou valeur) et un comportement défini (méthodes). Également la classe de base ultime du *new-style class*.

positional argument (argument de position)

Arguments affectés aux noms locaux internes à une fonction ou à une méthode, déterminés par l'ordre donné dans l'appel. La syntaxe * accepte plusieurs arguments de position ou fournit une liste de plusieurs arguments à une fonction. Voir **argument**.

property (propriété) [cf. p. 94]

Attribut d'instance permettant d'implémenter les principes de l'encapsulation.

Python3000

Surnom de la version 3 de Python (forgé il y a longtemps, quand la version 3 était un projet lointain). Aussi abrégé « Py3k ».

Pythonic (pythonique)

Idée ou fragment de code plus proche des idiomes du langage Python que des concepts fréquemment utilisés dans d'autres langages. Par exemple, un idiome fréquent en Python est de boucler sur les éléments d'un *iterable* en utilisant l'instruction `for`. Beaucoup d'autres langages n'ont pas ce type de construction et donc les utilisateurs non familiers avec Python utilisent parfois un compteur numérique :

```
for i in range(len(food)):
    print(food[i])
```

au lieu d'utiliser la méthode claire et pythonique :

```
for piece in food:
    print(piece)
```

reference count (nombre de références) [cf. p. 14]
Nombre de références d'un objet. Quand le nombre de références d'un objet tombe à zéro, l'objet est désalloué. Le comptage de références n'est généralement pas visible dans le code Python, mais c'est un élément clé de l'implémentation de *CPython*. Le module `sys` définit la fonction `getrefcount()` que les programmeurs peuvent appeler pour récupérer le nombre de références d'un objet donné.

__slots__
Une déclaration à l'intérieur d'une classe de style nouveau qui économise la mémoire en prédéclarant l'espace pour les attributs et en éliminant en conséquence les dictionnaires d'instance. Bien que populaire, cette technique est quelque peu difficile à mettre en place et doit être réservée aux rares cas où il y a un nombre important d'instances dans une application où la mémoire est réduite.

sequence (séquence) [cf. p. 29]
Un *iterable* qui offre un accès efficace aux éléments en utilisant des index entiers et les méthodes spéciales `__getitem__()` et `__len__()`. Des types séquences incorporés sont `list`, `str`, `tuple` et `unicode`. Notez que le type `dict` comporte aussi les méthodes `__getitem__()` et `__len__()`, mais est considéré comme une table associative plutôt que comme une séquence, car la recherche se fait à l'aide de clés arbitraires immuables au lieu d'index.

slice (tranche) [cf. p. 31]
Objet contenant normalement une partie d'une séquence. Une tranche est créée par une notation indexée utilisant des « : » entre les index quand plusieurs sont donnés, comme dans `variable_name[1:3:5]`. La notation crochet utilise les objets *slice* de façon interne.

special method (méthode spéciale) [cf. p. 69]
Méthode appelée implicitement par Python pour exécuter une certaine opération sur un type, par exemple une addition. Ces méthodes ont des noms commençant et finissant par deux caractères soulignés. Les méthodes spéciales sont documentées dans *Special method names*.

statement (instruction) [cf. p. 23]
Une instruction est une partie d'une suite, d'un « bloc » de code. Une instruction est soit une expression soit une ou plusieurs constructions utilisant un mot clé comme `if`, `while` ou `for`.

triple-quoted string (chaîne multi-ligne) [cf. p. 15]
Une chaîne délimitée par trois guillemets (") ou trois apostrophes ('). Bien qu'elles ne fournissent pas de fonctionnalités différentes de celles des chaînes simplement délimitées, elles sont utiles pour nombre de raisons. Elles permettent d'inclure des guillemets ou des apostrophes non protégés et elles peuvent s'étendre sur plusieurs lignes sans utiliser de caractère de continuation, et sont donc spécialement utiles pour rédiger des chaînes de documentation.

type (type) [cf. p. 9]
Le type d'un objet Python détermine de quelle sorte d'objet il s'agit ; chaque objet possède un type. Le type d'un objet est accessible grâce à son attribut `__class__` ou peut être retourné par la fonction `type(obj)`.

view (vue)
Les objets retournés par `dict.keys()`, `dict.values()` et `dict.items()` sont appelés des *dictionary views*. ce sont des « séquences paresseuses [1] » qui laisseront voir les modifications du dictionnaire sous-jacent. Pour forcer le *dictionary view* à être une liste complète, utiliser `list(dictview)`. Voir Dictionary view objects.

virtual machine (machine virtuelle) [cf. p. 3]
Ordinateur entièrement défini par un programme. La machine virtuelle Python exécute le bytecode généré par le compilateur.

1. L'évaluation paresseuse est une technique de programmation où le programme n'exécute pas de code avant que les résultats de ce code ne soient réellement nécessaires. Le terme « paresseux » (en anglais *lazzy evaluation*) étant connoté négativement en français on parle aussi d'évaluation « retardée ».

Zen of Python [cf. p. 109]

Une liste de principes méthodologiques et philosophiques utiles pour la compréhension et l'utilisation du langage Python. Cette liste peut être obtenue en tapant `import this` dans l'interpréteur Python.

Table des figures

Table des matières