



Robert CORDEAU

Introduction à Python 3

version 1.1



robert.cordeau@u-psud.fr

Informatique :

Rencontre de la logique formelle et du fer à souder.

Maurice NIVAT



Remerciements :

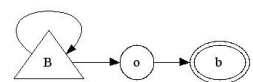
Sans les encouragements de Gérard SWINNEN, jamais je n'aurais osé me lancer dans l'aventure de l'enseignement de Python. Qu'il en soit remercié.

Ce document a bénéficié des corrections *impitoyables* de Laurent POINTAL (LIMSI), des lectures attentives de Michelle CORDEAU et Georges VINCENTS (IUT d'Orsay, M.P.).

Grand merci à ma fille Hélène pour ses illustrations ; les aventures de *Steven* le Python enchantent les têtes de paragraphes.

Merci à Tarek ZIADÉ pour les emprunts à ses publications, en particulier je remercie les éditions **Eyrolles** pour leur aimable autorisation de publier le dialogue de la page 102, ainsi que les éditions **Dunod** pour leur aimable autorisation de publier les exemples des pages 88, 90, 96 et 98.

Enfin il me faudrait saluer tous les auteurs que j'ai butiné sur internet... Qu'au moins, je n'oublie pas ceux à qui j'ai fait les plus grands emprunts dans les annexes : SEBSAUVAGE et Christian SCHINDELHAUER.



Avant-propos

La version 3 actuelle de Python abolit la compatibilité descendante avec la série des versions 2.x¹, dans le but d'éliminer les faiblesses originelles du langage. La ligne de conduite du projet était de « réduire la redondance dans le fonctionnement de Python par la suppression des méthodes obsolètes ».

À qui s'adresse ce cours ?

Ce cours prend la suite des « Notes de cours Python² » destiné aux étudiants de Mesures Physiques de l'IUT d'Orsay.

Bien qu'à ce jour l'offre des bibliothèques tierces ne soit pas encore riche (entre autre la bibliothèque *numpy* n'est pas disponible), il semble utile de disposer d'un cours généraliste en français consacré à la version 3 de Python. Nous en avons profité pour étoffer le texte de trois chapitres et pour proposer une forme plus pratique pour un texte susceptible d'être imprimé, tout en restant agréable à consulter à l'écran.

Outre ce cadre universitaire assez réduit, ce cours s'adresse à toute personne désireuse d'apprendre Python en tant que premier langage de programmation.

Ces notes de programmation reposent sur quelques partis pris :

- le choix du langage Python version 3;
- le choix de logiciels libres :
 - des éditeurs spécialisés : Wing IDE, eric 4, Scite³.
 - des outils *open source* : gnuplot, L^AT_EX dans sa distribution MiK_TE_X version 2.8, l'éditeur T_EXnicCenter...
- et sur l'abondance des ressources et de la documentation sur le Web.

1. C'est une grave décision, mûrement réfléchi : « Un langage qui bouge peu permet une industrie qui bouge beaucoup » (Bertrand MEYER)

2. disponibles à l'adresse <http://www.iut-orsay.fr/dptmphy/Pedagogie/coursPython.pdf>.

3. à réserver pour Windows...

Table des matières

1	Introduction	3
1.1	Principales caractéristiques du langage Python	3
1.2	Matériel et logiciel	4
1.2.1	L'ordinateur	4
1.2.2	Deux sortes de programmes	4
1.3	Les langages	5
1.3.1	Des langages de différents niveaux	5
1.3.2	Bref historique des langages	5
1.4	Production des programmes	5
1.4.1	Deux techniques de production des programmes	5
1.4.2	Technique de production de Python	6
1.4.3	La construction des programmes	6
1.5	Algorithme et programme	6
1.5.1	Définitions	6
1.5.2	La présentation des programmes	7
1.6	Les implémentations de Python	7
2	La calculatrice Python	9
2.1	Les modes d'exécution	9
2.1.1	Les deux modes d'exécution d'un code Python	9
2.2	Identifiants et mots clés	10
2.2.1	Identifiants	10
2.2.2	Styles de nommage	10
2.2.3	Les mots réservés de Python 3	10
2.3	Notion d'expression	11
2.4	Les types de données entiers	11
2.4.1	Le type <code>int</code>	11
2.4.2	Le type <code>bool</code>	12
2.5	Les types de données flottants	12
2.5.1	Le type <code>float</code>	12
2.5.2	Le type <code>complex</code>	13
2.6	Variables et affectation	13
2.6.1	Les variables	13
2.6.2	L'affectation	13
2.6.3	Affecter n'est pas comparer !	14
2.6.4	Les variantes de l'affectation	14
2.6.5	Les affectations (explications graphiques)	14
2.7	Les chaînes de caractères	14

2.7.1	Les chaînes de caractères : présentation	14
2.7.2	Les chaînes de caractères : opérations	15
2.7.3	Les chaînes de caractères : fonctions <i>vs</i> méthodes	15
2.7.4	Méthodes de test de l'état d'une chaîne ch	16
2.7.5	Méthodes retournant une nouvelle chaîne	16
2.7.6	Les chaînes de caractères : indiçage simple	17
2.7.7	Extraction de sous-chaînes	17
2.8	Les données binaires	18
2.9	Les entrées-sorties	19
2.9.1	Les entrées	20
2.9.2	Les sorties	20
2.9.3	Les séquences d'échappement	21
3	Le contrôle du flux d'instructions	23
3.1	Les instructions composées	23
3.2	Choisir	24
3.2.1	Choisir : if - [elif] - [else]	24
3.2.2	Syntaxe compacte d'une alternative	25
3.3	Boucler	25
3.3.1	Boucler : while	25
3.3.2	Parcourir : for	25
3.4	Ruptures de séquences	26
3.4.1	Interrompre une boucle : break	26
3.4.2	Court-circuiter une boucle : continue	26
3.4.3	Syntaxe complète des boucles	26
3.4.4	Exceptions	27
4	Les conteneurs standard	29
4.1	Les séquences	29
4.1.1	Qu'est-ce qu'une séquence ?	29
4.2	Les listes	29
4.2.1	Définition, syntaxe et exemples	29
4.2.2	Initialisations et tests	30
4.2.3	Méthodes	30
4.2.4	Manipulation des « tranche »	30
4.3	Les listes en intension	31
4.4	Les tuples	32
4.5	Retour sur les références	32
4.6	Les tableaux associatifs	33
4.6.1	Les types tableaux associatifs	33
4.6.2	Les dictionnaires (dict)	34
4.7	Les ensembles (set)	35
4.8	Les fichiers textuels	36
4.8.1	Les fichiers : introduction	36
4.8.2	Gestion des fichiers	36
4.9	Itérer sur les conteneurs	37
4.10	L'affichage formaté	38

5 Fonctions et espaces de noms	41
5.1 Définition et syntaxe	41
5.2 Passage des arguments	43
5.2.1 Mécanisme général	43
5.2.2 Un ou plusieurs paramètres, pas de retour	43
5.2.3 Un ou plusieurs paramètres, utilisation du retour	43
5.2.4 Passage d'une fonction en paramètre	44
5.2.5 Paramètres avec valeur par défaut	44
5.2.6 Nombre d'arguments arbitraire : passage d'un tuple	45
5.2.7 Nombre d'arguments arbitraire : passage d'un dictionnaire	45
5.3 Espaces de noms	46
5.3.1 Portée des objets	46
5.3.2 Résolution des noms : règle <i>LGI</i>	46
6 Modules et packages	49
6.1 Modules	49
6.1.1 Import d'un module	49
6.1.2 Exemples	50
6.2 Bibliothèque standard	51
6.2.1 La bibliothèque standard	51
6.3 Bibliothèques tierces	54
6.3.1 Une grande diversité	54
6.3.2 Un exemple : la bibliothèque Unum	54
6.4 Packages	55
7 La programmation Orientée Objet	57
7.1 Insuffisance de l'approche procédurale	57
7.2 Terminologie	58
7.3 Classes et instanciation d'objets	59
7.3.1 L'instruction class	59
7.3.2 L'instanciation et ses attributs	59
7.3.3 Retour sur les espaces de noms	59
7.4 Méthodes	60
7.5 Méthodes spéciales	60
7.5.1 Les méthodes spéciales	60
7.5.2 L'initialisateur	61
7.5.3 Surcharge des opérateurs	61
7.5.4 Exemple de surcharge	61
7.6 Héritage et polymorphisme	62
7.6.1 Héritage et polymorphisme	62
7.6.2 Exemple d'héritage et de polymorphisme	62
7.7 Retour sur l'exemple initial	62
7.7.1 La classe Cercle : conception	62
7.7.2 La classe Cercle	63
7.8 Notion de Conception Orientée Objet	64
7.8.1 Composition	64
7.8.2 Dérivation	65

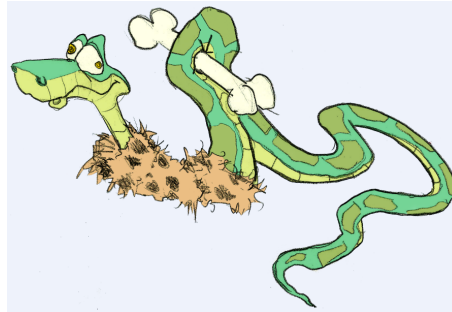
8	Techniques avancées	67
8.1	Techniques procédurales	67
8.1.1	Améliorer la documentation	67
8.1.2	Faire des menus avec un dictionnaire	68
8.1.3	Les fonctions récursives	69
8.1.4	Les générateurs et les expressions génératrices	71
8.1.5	Les fonctions incluses	72
8.1.6	Les décorateurs	73
8.2	Techniques objets	73
8.2.1	<code>__slots__</code> et <code>__dict__</code>	73
8.2.2	Functor	74
8.2.3	Les gestionnaires de contexte	74
8.3	Techniques fonctionnelles	75
8.3.1	Directive <code>lambda</code>	75
8.3.2	Les fonctions <code>map</code> , <code>filter</code> et <code>reduce</code>	75
8.3.3	Les applications partielles de fonctions	77
9	La programmation « OO » graphique	79
9.1	Programmes pilotés par des événements	79
9.2	La bibliothèque <code>tkinter</code>	79
9.2.1	Les widgets de <code>tkinter</code>	79
9.2.2	Le positionnement des widgets	81
9.3	Deux exemples	81
9.3.1	<code>tkPhone</code> , un exemple sans menu	81
9.3.2	<code>IDLE</code> , un exemple avec menu	85
10	Notion de développement agile	87
10.1	Les tests	87
10.1.1	Tests unitaires et tests fonctionnels	88
10.1.2	Le développement dirigé par les tests	88
10.2	La documentation	89
10.2.1	Le format <code>reST</code>	90
10.2.2	Le module <code>doctest</code>	93
10.2.3	Le développement dirigé par la documentation	96
A	Interlude	101
B	Jeux de caractères et encodage	104
C	Les fonctions logiques	108
D	Les bases arithmétiques	109
E	Les fonctions de hachage	110
F	Exercices corrigés	112
G	Licence associée à ce document	125
H	Ressources	131

Table des figures

1.1	Chaîne de compilation	5
1.2	Technique de l'interprétation	6
1.3	Interprétation du bytecode compilé	6
2.1	La boucle d'évaluation de IDLE	9
2.2	L'affectation illustrée.	15
2.3	L'indigage d'une chaîne.	18
2.4	Extraction de sous-chaînes.	18
2.5	Codes, glyphes, caractères et octets.	19
2.6	Les entrées-sorties.	20
2.7	Utilisation des séquences d'échappement	21
3.1	Les instructions composées.	24
4.1	Assignation augmentée d'un objet non modifiable.	33
4.2	Assignation augmentée d'un objet modifiable.	34
4.3	Opérations sur les ensembles	35
5.1	Les avantages de l'utilisation des fonctions	42
5.2	Passage des arguments par affectation.	43
5.3	Règle LGI	46
7.1	Conception UML de la classe <code>Cercle</code> .	63
8.1	Empilage/dépilage de 4!	70
8.2	Application partielle d'un widget	77
9.1	Deux styles de programmation.	80
9.2	Un exemple simple	80
9.3	<code>tkPhone</code> .	81
9.4	IDLE.	86
10.1	exemple de sortie au format HTML.	92
10.2	Exécution du script <code>doctest1.py</code> .	94
10.3	Exécution du script <code>doctest2.py</code> .	95
10.4	Exécution du script <code>example.py</code> .	97
10.5	Documentation du script <code>doctest2.py</code> .	98
10.6	Documentation du script <code>example.py</code> .	99
B.1	Table ASCII.	104
B.2	Extrait de la table Unicode.	105

B.3	Le jeu de caractères cp1252.	107
E.1	Le principe du hachage	110
E.2	Hachage des clés d'un dictionnaire.	111
G.1	Paternité – Pas d'Utilisation Commerciale – Partage Des Conditions Initiales À L'Identique	125

Introduction à l'informatique



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

1.1 Principales caractéristiques du langage Python

- **Historique**
 - 1991 : Guido van ROSSUM publie Python au CWI (Pays-Bas) à partir du langage ABC et du projet AMOEBA (système d'exploitation distribué)
 - 1996 : sortie de *Numerical Python*
 - 2001 : naissance de la PSF (Python Software Foundation)
 - Les versions se succèdent... Un grand choix de modules disponibles, des colloques annuels sont organisés, Python est enseigné dans plusieurs universités et est utilisé en entreprise...
 - Fin 2008 : sorties simultanées de Python 2.6 et de Python 3
- **Langage Open Source**
 - Licence Open Source CNRI, compatible GPL, mais sans la restriction *copyleft*. 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
 - Importantes documentations en ligne
 - Développement rapide et incrémentiel
 - Tests et débogage faciles
 - Analyse interactive de données
- **Langage interprété rapide**
 - Interprétation du *bytecode* compilé

- De nombreux modules sont disponibles à partir de bibliothèques optimisées écrites en C, C++ ou FORTRAN
- **Simplicité du langage** (cf. Zen of Python p. 101) :
 - Syntaxe claire et cohérente
 - Indentation significative
 - Gestion automatique de la mémoire (*garbage collector*)
 - Typage dynamique fort : pas de déclaration
- **Orienté objet**
 - Modèle objet puissant mais pas obligatoire
 - Structuration multifichier très facile 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

1.2 Environnements matériel et logiciel

1.2.1 L'ordinateur

On peut simplifier la définition de l'ordinateur de la façon suivante :

Définition



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 permanente (dite *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 Les 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, mais 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*.

1.3.2 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.

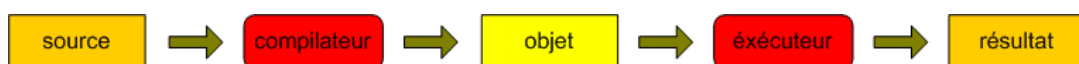


FIGURE 1.1 – Chaîne de compilation

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 exécution...



FIGURE 1.2 – Technique de l'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**.

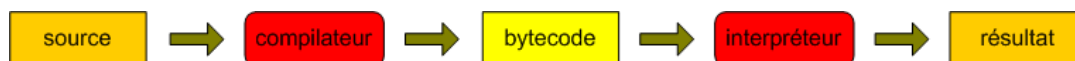


FIGURE 1.3 – Interprétation du bytecode compilé

1.4.3 La 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**. 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

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

Un algorithme se termine en un **temps fini**

Définition

i 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 La présentation des programmes

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

La signification de parties non triviales (et uniquement celles-là) doit être expliquée par un **commentaire**.

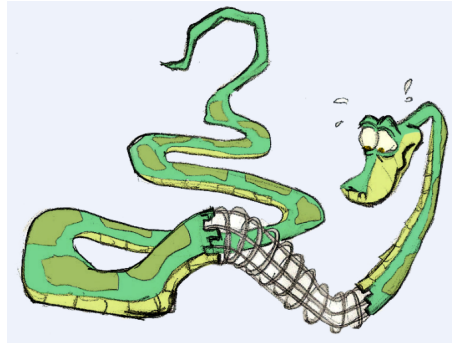
Un commentaire commence par le caractère **#** et s'étend jusqu'à la fin de la ligne :

```
#-----  
# Voici un commentaire  
#-----  
  
9 + 2 # En voici un autre
```

1.6 Les implémentations de Python

- **CPython** : *Classic Python*, codé en C, portable sur différents systèmes
- **Python3000** : *Python 3*, la nouvelle implémentation de CPython
- <+> **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 récuser tant que l'on veut)
- **Pypy** : projet de recherche européen d'un interpréteur Python écrit en Python

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 définis en mathématique.

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

Enfin, *last but not the least*, ce chapitre s'étend sur les notions non triviales de variables, de références 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 de commandes Python dans un fichier grâce à un éditeur (on parle alors d'un *script Python*) que l'on exécute par une touche du menu de l'éditeur ;
- soit on utilise un interpréteur (par exemple IDLE) pour obtenir un résultat immédiat grâce à l'interpréteur Python embarqué dans IDLE qui exécute la *boucle d'évaluation* (cf. Fig. 2.1)

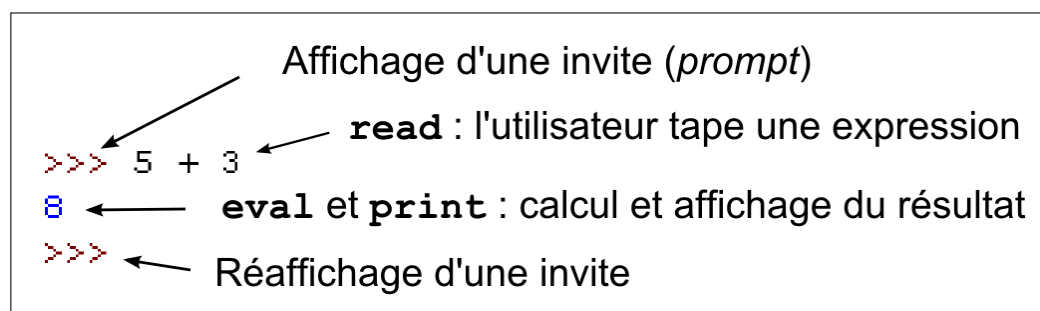


FIGURE 2.1 – La boucle d'évaluation de IDLE

2.2 Identifiants et mots clés

2.2.1 Identifiants

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

Définition

i Un identifiant Python valide est une suite non vide de caractères, de longueur quelconque, formée d'**un caractère de début** et de **zéro ou plusieurs caractères de continuation**

Sachant que :

- un *caractère de début* peut être n'importe quelle lettre UNICODE (cf. p. 104), ainsi que le caractère souligné (`_`).
- un *caractère de continuation* est un caractère de début, un chiffre ou un point.

Attention



Les identifiants sont sensibles à la casse et ne doivent pas être un mot clé.

2.2.2 Styles de nommage

Il est important d'utiliser une politique cohérente de nommage des identifiants. Voici les styles préconisés :

- **UPPERCASE** ou **UPPER_CASE** pour les constantes ;
- **TitleCase** pour les classes ;
- **camelCase** pour les fonctions, les méthodes et les interfaces graphiques ;
- **lowercase** ou **lower_case** pour tous les autres identifiants.

Exemples :

```
NB_ITEMS = 12           # UPPER_CASE
class MaClasse: pass    # TitleCase
def maFonction(): pass  # camelCase
mon_id = 5              # lower_case
```

2.2.3 Les mots réservés de Python 3

La version 3.1.1 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

i 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, d'identifiants et d'opérateurs.

Exemples de deux expressions simples et d'une expression complexe :

```
id1 = 15.3
id2 = maFonction(id1)
if id2 > 0:
    id3 = math.sqrt(id2)
else:
    id4 = id1 - 5.5*id2
```

2.4 Les types de données entiers

Python 3 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.

Les entiers littéraux sont décimaux par défaut, mais on peut aussi utiliser les bases suivantes (cf. p. 109) :

```
>>> 2009          # décimal
2009
>>> 0b11111011001 # binaire
2009
>>> 0o3731        # octal
2009
>>> 0x7d9         # hexadecimal
2009
```

Opérations arithmétiques

Les principales opérations :

```
20 + 3    # 23
20 - 3    # 17
20 * 3    # 60
20 ** 3   # 8000
20 / 3    # 6.666666666666667
20 // 3   # 6 (division entière)
20 % 3    # 2 (modulo)
abs(3 - 20) # valeur absolue
```

Bien remarquer le rôle des deux opérateurs de division :

`/` : produit une division flottante ;

`//` : produit une division entière.

2.4.2 Le type bool

- Deux valeurs possibles : `False`, `True`.
- Opérateurs de comparaison : `==`, `!=`, `>`, `>=`, `<` et `<=` :

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

- Opérateurs logiques (concept de *shortcut*) : `not`, `or` et `and`.
En observant les tables de vérité des opérateurs `and` et `or` (cf. p. 108), on remarque que :
 - dès qu'un premier membre à la valeur `False`, l'expression `False and expression2` vaudra `False`. On n'a donc pas besoin de l'évaluer ;
 - de même dès qu'un premier membre à la valeur `True`, l'expression `True or expression2` vaudra `True`.

Cette optimisation est appelée « principe du *shortcut* » :

```
(3 == 3) or (9 > 24)    # True (dès le premier membre)
(9 > 24) and (3 == 3)   # False (dès le premier membre)
```

- Les opérations logiques et de comparaisons sont évaluées afin de donner des résultats booléens dans `False`, `True`.

Les expressions booléennes

Une expression booléenne (cf. p. 108) a deux valeurs possibles : `False` ou `True`.

Python attribut à une expression booléennes la valeur `False` si c'est :

- la constante `False` ;
- la constante `None` ;
- une séquence ou une collection vide ;
- une donnée numérique de valeur 0.

Tout le reste vaut `True`.

2.5 Les types de données flottants

2.5.1 Le type float

- Un `float` est noté avec un point décimal ou en notation exponentielle :

```
2.718
.02
3e8
6.023e23
```

- Les flottants supportent les mêmes opérations que les entiers.
- Les `float` ont une précision finie indiquée dans `sys.float_info.epsilon`.
- L'import du module `math` autorise toutes les opérations mathématiques usuelles :

```
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)))   # 5.0 : module
```

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

```
import cmath

print(cmath.phase(-1 + 0j)) # 3.14159265359
print(cmath.polar(3 + 4j)) # (5.0, 0.9272952180016122)
print(cmath.rect(1., cmath.pi/4)) # (0.707106781187+0.707106781187j)
```

2.6 Variables et affectation

2.6.1 Les variables

Dès que l'on possède des *types de données*, on a besoin des *variables* pour stocker les données.

En réalité, Python n'offre pas la notion de variable, mais plutôt celle de *référence d'objet*. Tant que l'objet n'est pas modifiable (comme les entiers, les flottants, etc.), il n'y a pas de différence notable. On verra que la situation change dans le cas des objets modifiables...


Définition

 Une variable est un **identifiant** associé à une valeur.

Informatiquement, c'est une **référence d'objet** situé à une adresse mémoire.

2.6.2 L'affectation

Définition

 On **affecte** une variable par une valeur en utilisant le signe `=` (qui *n'a rien à voir* avec l'égalité en math!). Dans une affectation, le membre de gauche **reçoit** le membre de droite ce qui nécessite d'évaluer la valeur correspondant au membre de droite avant de l'affecter au membre de gauche.

```
a = 2 # prononcez : a "reçoit" 2
b = 7.2 * math.log(math.e / 45.12) - 2*math.pi
c = b ** a
```

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

```
a = a + 1 # 3 (incréméntation)
a = a - 1 # 2 (décréméntation)
```

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) :

```
>>> a = 2
>>> x = (a = 3) + 2
SyntaxError: invalid syntax
```

La **comparaison** a une valeur utilisable dans une expression mais n'a pas d'effet (l'automate interne représentant l'évolution du programme n'est pas modifié) :

```
>>> x = (a == 3) + 2
>>> x
2
```

2.6.4 Les variantes de l'affectation

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

```
# affectation simple
v = 4

# affectation augmentée
v += 2      # idem à : v = v + 2 si v est déjà référencé

# affectation de droite à gauche
c = d = 8   # cibles multiples

# affectations parallèles d'une séquence
e, f = 2.7, 5.1      # tuple
g, h, i = ['G', 'H', 'I'] # liste
x, y = coordonneesSouris() # retour multiple d'une fonction
```

2.6.5 Les affectations (explications graphiques)

Dans les schémas de la figure 2.2, les cercles représentent les identificateurs alors que les rectangles représentent les données.

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 :

2.7 Les chaînes de caractères

2.7.1 Les chaînes de caractères : présentation

Définition

i 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.

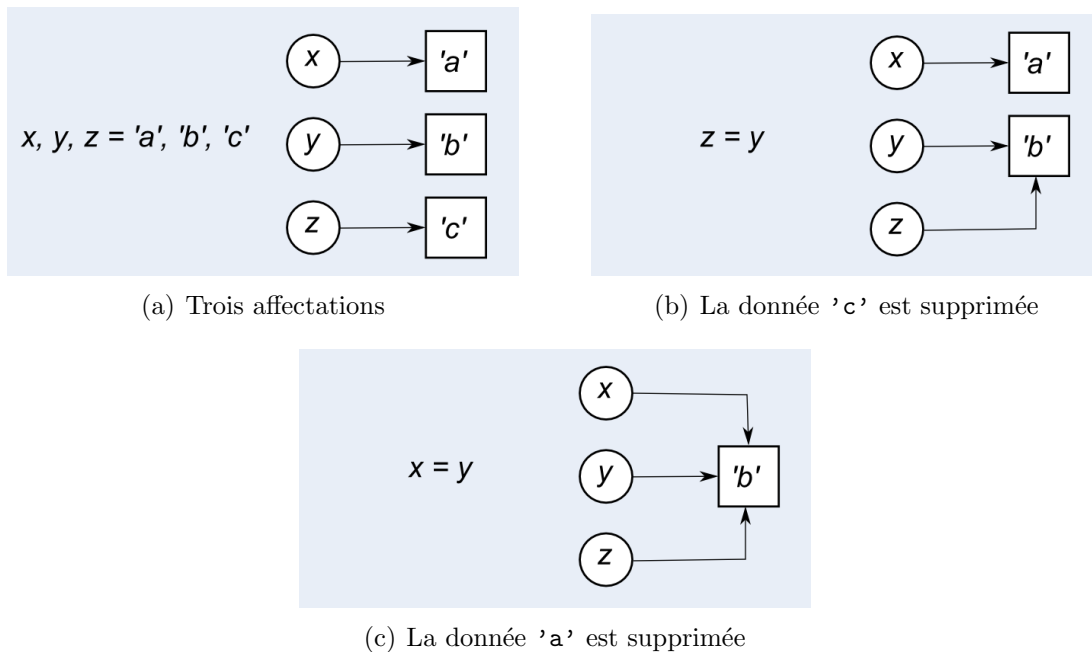


FIGURE 2.2 – L'affectation illustrée.

Trois syntaxes de chaînes 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"
syntaxe2 = r"Deuxième forme sans retour à la ligne \n"
syntaxe3 = """
    Troisième forme multi-ligne
    """
guillemets = "L'eau vive"
apostrophes = 'Forme "avec des apostrophes"'

```

2.7.2 Les chaînes de caractères : opérations

- Longueur :

```

s = "abcde"
len(s) # 5

```

- Concaténation :

```

s1 = "abc"
s2 = "defg"
s3 = s1 + s2 # 'abcdefg'

```

- Répétition :

```

s4 = "Fi! "
s5 = s4 * 3 # 'Fi! Fi! Fi! '
print(s5)

```

2.7.3 Les chaînes de caractères : fonctions *vs* méthodes

On peut agir sur une chaîne (et plus généralement sur une séquence) en utilisant des fonctions (notion procédurale) ou des méthodes (notion objet).

- Pour appliquer une fonction, on utilise l'opérateur `()` appliqué à la fonction :

```
ch1 = "abc"
long = len(ch1) # 3
```

- On applique une méthode à un objet en utilisant la **notation pointée** entre la donnée/variable à laquelle on applique la méthode, et le nom de la méthode suivi de l'opérateur `()` appliqué à la méthode :

```
ch2 = "abracadabra"
ch3 = ch2.upper() # "ABRACADABRA"
```

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

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

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

- `isupper()` et `islower()` : retournent `True` si `ch` ne contient respectivement que des majuscules/minuscules :

```
print("chAise basSe".isupper()) # False
```

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

```
print("Chaise Basse".istitle()) # True
```

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

```
print("3 chaises basses".isalpha()) # False
print("54762".isdigit()) # True
```

- `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` :

```
print("abracadabra".startswith('ab')) # True
print("abracadabra".endswith('ara')) # False
```

2.7.5 Méthodes retournant une nouvelle chaîne

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

```
# s sera notre chaîne de test pour toutes les méthodes
s = "chAise basSe"

print(s.lower()) # chaise basse
print(s.upper()) # CHAISE BASSE
print(s.capitalize()) # Chaise basse
print(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) :

```
print(s.center(20, '-')) # ----cHAise basSe----
print(s.rjust(20, '@')) # @@@@@@cHAise basSe
```

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

```
print(s.zfill(20)) # 0000000cHAise basSe
```

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

```
print(s.strip('ce')) # HAise basS
```

- `find(sub[, start[, stop]])` : renvoie l'indice 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 :

```
print(s.find('se b')) # 4
```

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

```
print(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 :

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

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

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

2.7.6 Les chaînes de caractères : indiçage simple

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

```
s = "Rayon X" # len(s) ==> 7
print(s[0]) # R
print(s[2]) # y
print(s[-1]) # X
print(s[-3]) # n
```

2.7.7 Extraction de sous-chaînes

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

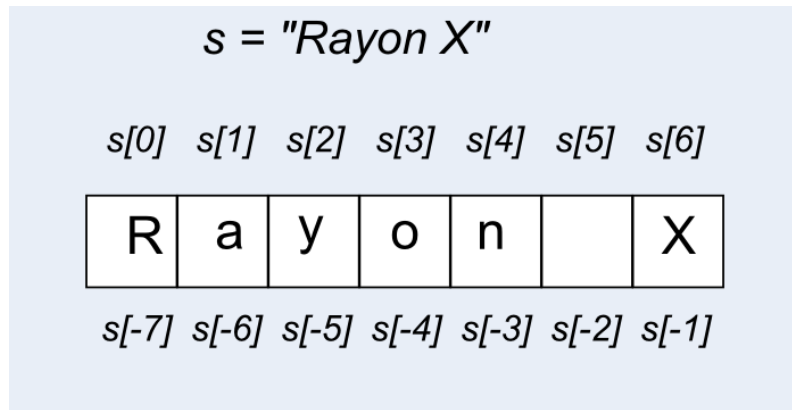


FIGURE 2.3 – L’indiciage d’une chaîne.

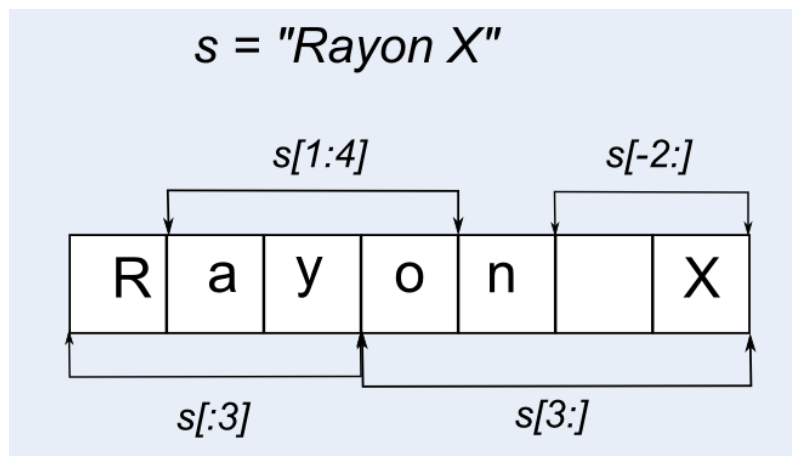


FIGURE 2.4 – Extraction de sous-chaînes.

```

s = "Rayon X" # len(s) ==> 7
s[1:4]      # 'ayo' (de l'indice 1 compris à 4 non compris)
s[-2:]     # ' X' (de l'indice -2 compris à la fin)
s[:3]      # 'Ray' (du début à 3 non compris)
s[3:]      # 'on X' (de l'indice 3 compris à la fin)
s[::2]     # 'RynX' (du début à la fin, de 2 en 2)

```

2.8 Les données binaires

Les types binaires

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

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`.

Exemples de données binaires et de méthodes :

```
# données binaires
b_mot = b"Animal"           # chaîne préfixée par b : type byte
print(b_mot)               # b'Animal'
for b in b_mot:
    print(b, end=" ")      # 65 110 105 109 97 108 (cf. table ASCII)
print()
bMot = bytearray(b_mot)    # retourne un nouveau tableau de bytes...
bMot.pop()                 # ...qui possède les méthodes usuelles
print()
print(bMot, "\n")         # bytearray(b'Anima')
data = b"5 Hills \x35\x20\x48\x69\x6C\x6C\x73"
print(data.upper())        # b'5 HILLS 5 HILLS'
print(data.replace(b"ill", b"at")) # b'5 Hats 5 Hats'
```

Bien différencier les codes, glyphes, caractères et octets ! (Cf. Fig. 2.5)

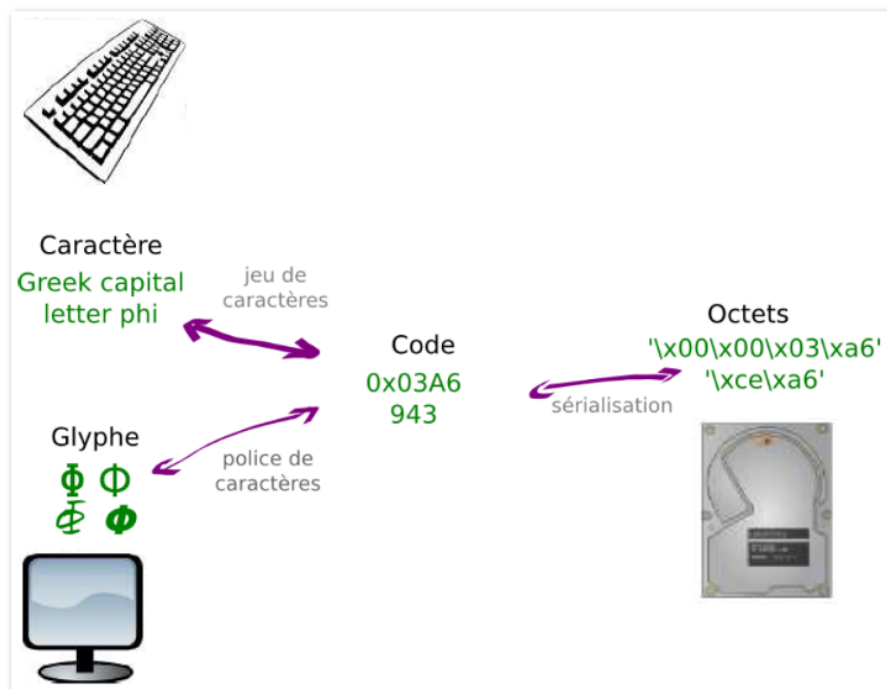


FIGURE 2.5 – Codes, glyphes, caractères et octets.

2.9 Les entrées-sorties

L'utilisateur a besoin d'interagir avec le programme (cf. Fig. 2.6). En mode « console » (on verra 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.

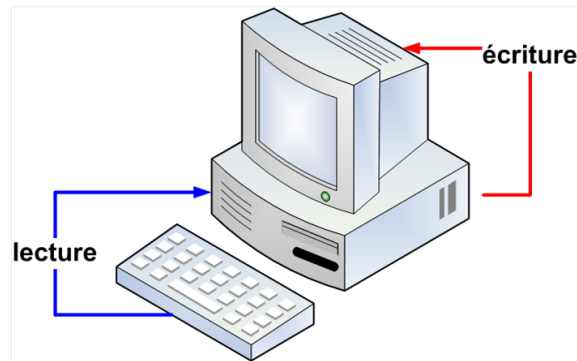


FIGURE 2.6 – Les entrées-sorties.

2.9.1 Les entrées

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

La fonction standard `input()` effectue toujours une saisie en *mode texte* (la saisie est une chaîne) dont on peut ensuite changer le type (on dit aussi *transtyper*) :

```
nb_etudiant = input("Entrez le nombre d'étudiants : ")
print(type(nb_etudiant)) # <class 'str'> (c'est une chaîne)

f1 = input("\nEntrez un flottant : ")
f1 = float(f1)          # transtypage en flottant
# ou plus brièvement :
f2 = float(input("Entrez un autre flottant : "))
print(type(f2))        # <class 'float'>
```

2.9.2 Les sorties

En mode « calculatrice », Python *lit-évalue-affiche*, mais la fonction `print()` reste indispensable aux affichages dans les scripts :

```
import sys

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()                    # affiche une nouvelle ligne
# pour afficher un espace à la place de la nouvelle ligne:
print(a, end=" ")          # 2 (et ne va pas à la ligne)
print("\nErreur fatale !", file=sys.stderr) # dans un fichier
print("On a <", 2**32, "> cas !", sep="###")
# On a <###4294967296###> cas !
```

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 :

Séquence	Signification
\saut_ligne	saut de ligne ignoré
\\	affiche un antislash
\'	apostrophe
\"	guillemet
\a	sonnerie (<i>bip</i>)
\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
\Uhhhhhhh	caractère sous forme de code Unicode 32 bits
\ooo	caractère sous forme de code octal
\xhh	caractère sous forme de code hexadécimal

Exemples :

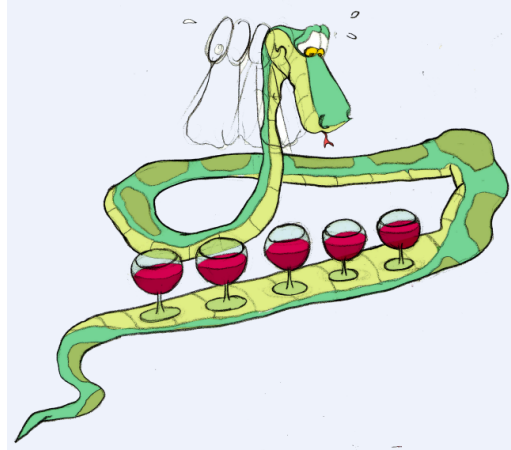
```

D:\Pythiut3\CL2\Src\2_067.py - Sc1
File Edit Search View Tools Options Language Buffers Help
1_2_067.py
1 print("\N{pound sign} \u00A3 \U000000A3") # £ £ £
2
3 print("d \144 \x64") # d d d
4 # séquence d'échappement inactive dans les chaînes brutes :
5 print(r"d \144 \x64") # d \144 \x64
6

```

FIGURE 2.7 – Utilisation des séquences d'échappement

Le contrôle du flux d'instructions



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

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

Cette exécution en séquence peut être modifiée pour *choisir* ou *répéter* des portions de code. C'est l'objet principal de ce chapitre.

3.1 Les instructions composées

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**.
-

Attention

⚠ Toutes les instructions au même niveau d'indentation appartiennent au même bloc (cf. Fig. 3.1).

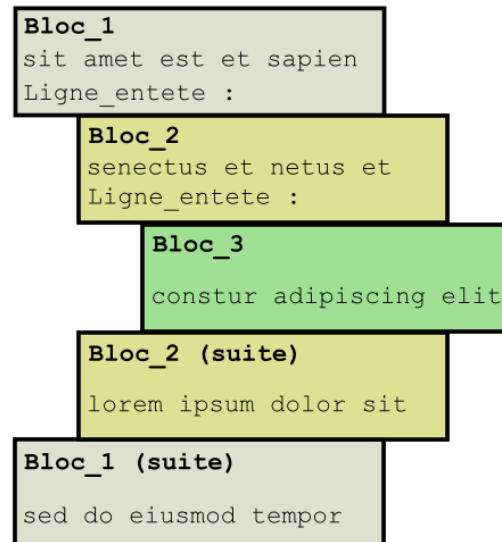


FIGURE 3.1 – Les instructions composées.

Exemple :

```
somme = 0.0
nb_valeurs = 0

for v in valeurs:
    nb_valeurs = nb_valeurs + 1
    somme = somme + valeurs

moyenne = somme / nb_valeurs
```

On a souvent besoin d'imbriquer les instructions composées :

```
if a == 0:
    if b != 0:
        print("\nx = {:.2f}".format(-c/b))
    else:
        print("\nPas de solution.")
else:
    delta = b**2 - 4*a*c
    if delta > 0.0:
        rac_delta = sqrt(delta)
        print("\nx1 = {:.2f} \t x2 = {:.2f}"
              .format((-b-rac_delta)/(2*a), (-b+rac_delta)/(2*a)))
    elif delta < 0.0:
        print("\nPas de racine réelle.")
    else:
        print("\nx = {:.2f}".format(-b/(2*a)))
```

3.2 Choisir

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

Contrôler une alternative :

```
if x < 0:
    print("x est négatif")
```

```
elif x % 2:
    print("x est positif et impair")
else:
    print("x n'est pas négatif et est pair")
```

Test d'une valeur booléenne :

```
if x: # mieux que (if x is True:) ou que (if x == True:)
    pass
```

3.2.2 Syntaxe compacte d'une alternative

Pour trouver, par exemple, le minimum de deux nombres, on peut utiliser l'opérateur ternaire (repris du C) :

```
x, y = 4, 3

# Ecriture classique :
if x < y:
    plus_petit = x
else:
    plus_petit = y

# Utilisation de l'opérateur ternaire :
plus_petit = x if x < y else y
print("Plus petit : ", plus_petit) # 3
```

3.3 Boucler

3.3.1 Boucler : while

Répéter une portion de code :

```
x, cpt = 257, 0
print("L'approximation de log2 de", x, "est", end=" ")
while x > 1:
    x //= 2      # division avec troncature
    cpt += 1    # incrémentation
print(cpt, "\n") # 8
```

Utilisation classique : la *saisie filtrée* d'une valeur numérique (on doit *préciser le type* car `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* (tout conteneur que l'on peut parcourir élément par élément, dans l'ordre ou non, suivant son type) :

```

for lettre in "ciao":
    print(lettre, end=" ") # c i a o

for x in [2, 'a', 3.14]:
    print(x, end=" ") # 2 a 3.14

for i in range(5):
    print(i, end=" ") # 0 1 2 3 4

```

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): # 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
    if x == 5:
        break
    print(x, end=" ")

print("\nBoucle interrompue pour x =", x)

# affiche :
# 1 2 3 4
# Boucle interrompue pour x = 5

```

3.4.2 Court-circuiter une boucle : continue

Passes 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): # 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
    if x == 5:
        continue
    print(x, end=" ")

print("\nLa boucle a sauté la valeur 5")

# affiche :
# 1 2 3 4 6 7 8 9 10
# La boucle a sauté la valeur 5

```

3.4.3 Syntaxe complète des boucles

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

for - else

Un exemple avec le parcours d'une liste :

```
une_sequence = [2, 5, 9, 7, 11]

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

for i in une_sequence:
    if i == cible:
        sauve = i
        break      # voici l'interruption !
else:
    print(cible, "n'est pas dans", une_sequence)
    sauve = None

# sauve vaut donc cible ou None :
print("On obtient sauve =", sauve)
```

3.4.4 Exceptions

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


Le mécanisme des **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 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 une 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 sont des instances de sous-classe de la classe `Exception`.

La hiérarchie des sous-classes offre une vingtaine d'exceptions standard.

L'instruction `raise` permet de lever *volontairement* une exception :

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

Syntaxe complète d'une exception :

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

Les 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 Les séquences

4.1.1 Qu'est-ce qu'une séquence ?

Définition

i Une séquence est un conteneur **ordonné** d'éléments **indiqués par des entiers**.

Python dispose de trois types prédéfinis de séquences :

- les chaînes (vues précédemment) ;
- les listes ;
- les tuples¹.

4.2 Les listes

4.2.1 Définition, syntaxe et exemples

Définition

i 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.

```
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

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

```
truc, machin = [], [0.0] * 3
print(truc)             # [] (liste vide)
print(machin)           # [0.0, 0.0, 0.0]

l1 = list(range(4))
print("l1 =", l1)       # l1 = [0, 1, 2, 3]
l2 = list(range(4, 8))
print("l2 =", l2)       # l2 = [4, 5, 6, 7]
l3 = list(range(2, 9, 2))
print("l3 =", l3)       # l3 = [2, 4, 6, 8]

print(2 in l1, 8 in l2, 6 in l3) # True False True

for i in range(len(l3)):
    print(i, l3[i], sep="-", end=" ") # 0-2 1-4 2-6 3-8
```

4.2.3 Méthodes

Quelques méthodes de modification des listes :

```
nombres = [17, 38, 10, 25, 72]
nombres.sort()
print(nombres)          # [10, 17, 25, 38, 72]
nombres.append(12)
nombres.reverse()
nombres.remove(38)
print(nombres)          # [12, 72, 25, 17, 10]
print(nombres.index(17)) # 3
nombres[0] = 11
nombres[1:3] = [14, 17, 2]
print(nombres.pop())    # 10
print(nombres)          # [11, 14, 17, 2, 17]
print(nombres.count(17)) # 2
nombres.extend([1, 2, 3])
print(nombres)          # [11, 14, 17, 2, 17, 1, 2, 3]
```

4.2.4 Manipulation des « tranche »

Syntaxe



Si on veut supprimer, remplacer ou insérer *plusieurs* éléments d'une liste, il faut

indiquer une tranche 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
print(mots)        # ['jambon', 'sel', 'beurre']
mots[1:3] = ['salade']
print(mots)        # ['jambon', 'salade']
mots[1:] = ['mayonnaise', 'poulet', 'tomate']
print(mots)        # ['jambon', 'mayonnaise', 'poulet', 'tomate']
mots[2:2] = ['miel']      # insertion en 3è position
print(mots)        # ['jambon', 'mayonnaise', 'miel', 'poulet', 'tomate']
```

4.3 Les listes en intension

Une **liste en intension** est une expression qui permet de générer une liste de manière très compacte. Cette notation reprend la définition mathématique d'un **ensemble en intension** :

$$\{x^2 | x \in [2, 10]\} \Rightarrow [x**2 \text{ for } x \text{ in range}(2, 11)]$$

Définition

 Une liste en intension est équivalente à une boucle `for` qui construirait la même liste en utilisant la méthode `append()`.

Les listes en intension 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)
```

Des utilisations très *pythoniques* :

```

valeurs_s = ["12", "78", "671"]
# conversion d'une liste de chaînes en liste d'entier
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]]

```

4.4 Les tuples

Définition



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])
```

- Les tuples s'utilisent comme les listes mais leur parcours est plus rapide ;
- Ils sont utiles pour définir des constantes.

Attention



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

4.5 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 :

- crée en mémoire un objet du type *ad hoc* (membre de droite) ;
- stocke la donnée dans l'objet créé ;
- crée un nom de variable (membre de gauche) ;
- associe ce nom de variable à l'objet contenant la valeur.

Une conséquence de ce mécanisme est que, si un objet modifiable est affecté, tout changement sur un objet modifiera l'autre :

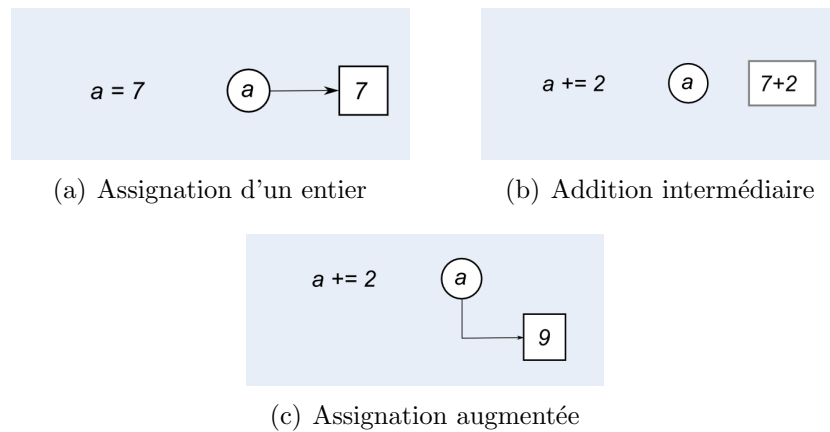


FIGURE 4.1 – Assignment augmentée d'un objet non modifiable.

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

phrase[4] = "casse"

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

Si l'on désire réaliser une *vraie* copie¹ d'un objet, on doit utiliser le module `copy` :

```
import copy

a = [1, 2, 3]
b = a # une référence
b.append(4)
print(a) # [1, 2, 3, 4]
c = copy.copy(a) # une copie de "surface"
c.append(5)
print(c) # [1, 2, 3, 4, 5]
print(a) # [1, 2, 3, 4]
```

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()`.

Complément graphique sur l'assignation

- Assignation augmentée d'un objet non modifiable (cas d'un entier : Fig. 4.1). On a représenté en gris clair l'addition intermédiaire :
- Assignation augmentée d'un objet modifiable (cas d'une liste : Fig. 4.2) : On a représenté en gris clair la création de la liste intermédiaire :

4.6 Les tableaux associatifs

4.6.1 Les types tableaux associatifs

Définition

i Un tableau associatif est un type de données permettant de stocker des couples

1. Dans le cas général. Pour copier une séquence simple, par exemple une liste `l`, on peut toujours écrire `l2 = l[:]`

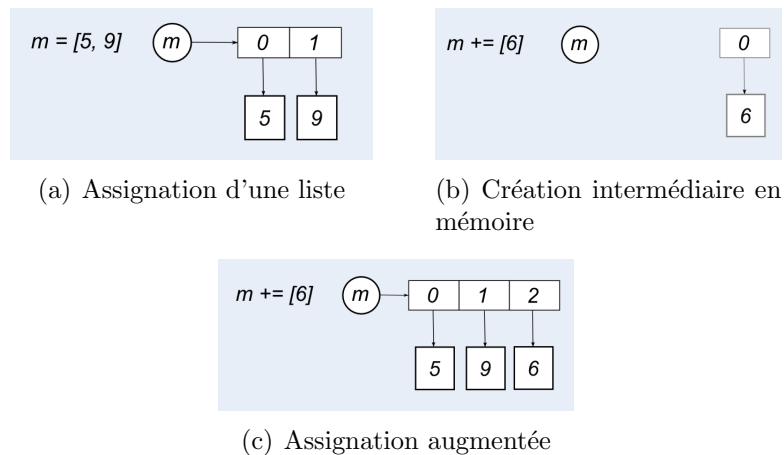


FIGURE 4.2 – Assignment augmentée d'un objet modifiable.

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`.

4.6.2 Les dictionnaires (`dict`)

Syntaxe

 Collection de couples **cle** : **valeur** entourée d'accolades.

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

Comme les listes, les dictionnaires sont modifiables, mais les couples enregistrés n'occupent pas un ordre immuable, leur emplacement est géré par un algorithme spécifique (Cf. les fonctions de hachage p. 110).

Une *clé* pourra être alphabétique, numérique...en fait tout type hachable. Les *valeurs* pourront être des valeurs numériques, des séquences, des dictionnaires, mais aussi des fonctions, des classes ou des instances.

Exemples de création

```
# insertion de clés/valeurs une à une
d1 = {} # dictionnaire vide
d1["nom"] = 3
d1["taille"] = 176
print(d1) # {'nom': 3, 'taille': 176}
# définition en extension
d2 = {"nom": 3, "taille": 176}
print(d2) # {'nom': 3, 'taille': 176}
# définition en intension
d3 = {x: x**2 for x in (2, 4, 6)}
```

```
print(d3) # {2: 4, 4: 16, 6: 36}
# utilisation de paramètres nommés
d4 = dict(nom=3, taille=176)
print(d4) # {'taille': 176, 'nom': 3}
# utilisation d'une liste de couples clés/valeurs
d5 = dict([("nom", 3), ("taille", 176)])
print(d5) # {'nom': 3, 'taille': 176}
```

Méthodes

Quelques méthodes applicables aux dictionnaires :

```
tel = {'jack': 4098, 'sape': 4139}
tel['guido'] = 4127

print(tel) # {'sape': 4139, 'jack': 4098, 'guido': 4127}
print(tel['jack']) # 4098
del tel['sape']
tel['irv'] = 4127
print(tel) # {'jack': 4098, 'irv': 4127, 'guido': 4127}

print(list(tel.keys())) # ['jack', 'irv', 'guido']
print(sorted(tel.keys())) # ['guido', 'irv', 'jack']
print(sorted(tel.values())) # [4098, 4127, 4127]

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

4.7 Les ensembles (set)

Définition

i Collection itérable non ordonnée d'éléments hachables distincts.

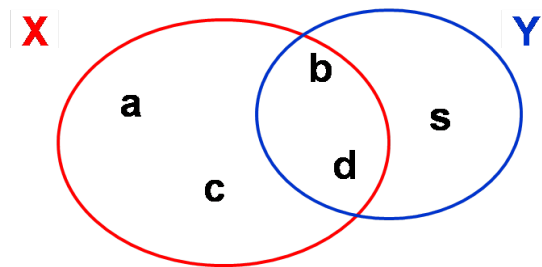


FIGURE 4.3 – Opérations sur les ensembles

```
X, Y = set('abcd'), set('sbds')
print("X =", X) # X = {'a', 'c', 'b', 'd'}
print("Y =", Y) # Y = {'s', 'b', 'd'} : un seul élément 's'

print('c' in X) # True
print('a' in Y) # False
print(X - Y) # {'a', 'c'}
print(Y - X) # {'s'}
print(X | Y) # {'a', 'c', 'b', 'd', 's'}
print(X & Y) # {'b', 'd'}
```

4.8 Les fichiers textuels

4.8.1 Les fichiers : introduction

On rappelle que l'ordinateur n'exécute que les programmes présents dans sa mémoire volatile (la RAM).

Mais, pour conserver durablement des informations, il faut utiliser une mémoire permanente comme par exemple le disque dur, la clé USB, le DVD,...

Comme la plupart des langages, Python utilise classiquement la notion de **fichier**. C'est un type pré-défini en Python, qui ne nécessite donc pas d'importer de module externe.

Nous nous limiterons aux fichiers *textuels* (portables, lisible par un éditeur), mais signalons que les fichiers stockés en codage *binnaire* sont plus compacts et plus rapides à gérer.

4.8.2 Gestion des fichiers

Ouverture et fermeture des fichiers

Principaux *modes* d'ouverture des fichiers textuels :

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

Python utilise les fichiers en mode *texte* par défaut (noté **t**) (pour les fichiers *binnaires*, il faut préciser le mode **b**).

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

Une seule méthode de fermeture :

```
f1.close()
```

Écriture séquentielle

Méthodes d'écriture :

```
f = open("truc.txt", "w")
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()

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

Lecture séquentielle

Méthodes de lecture :

```
f = open("truc.txt", "r")
s = f.read()      # lit tout le fichier --> string
s = f.read(3)    # lit au plus n octets --> string
s = f.readline() # lit la ligne suivante --> string
s = f.readlines() # lit tout le fichier --> liste de strings
f.close()

# Affichage des lignes d'un fichier une à une
f = open("truc.txt") # mode "r" par défaut
for ligne in f:
    print(ligne[:-1]) # pour sauter le retour à la ligne
f.close()
```

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 clés et valeurs en bouclant sur une liste :

```
for i, v in enumerate(["tic", "tac", "toe"]):
    print(i, v, end=" ", sep="->") # 0->tic 1->tac 2->toe
```

Boucler sur deux séquences (ou plus) appariées :

```
question = ["name", "quest", "favorite color"]
answers = ["Lancelot", "the Holy Grail", "blue"]
for q, a in zip(question, answers):
    print("What is your {}? It is {}".format(q, a))
# What is your name? It is Lancelot.
# What is your quest? It is the Holy Grail.
# What is your favorite color? It is blue.
```

Boucler sur une séquence inversée (la séquence initiale est inchangée) :

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

Boucler sur une séquence triée à éléments uniques (la séquence initiale est inchangée) :

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

4.10 L'affichage formaté

La méthode `format()` permet de contrôler finement toutes sortes d'affichages.

Remplacements simples :

```
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'{} et du {} 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 {}".format(dict(name='Fred')))
# My name is Fred
d = dict(animal = 'éléphant', poids = 12000)
print("L'{} pèse {} kg\n".format(d))
# L'éléphant pèse 12000 kg
```

Remplacement avec attributs nommés :

```
import math
import sys

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

Conversions `str()` et `repr()` :

```
print("{}!s {}!r".format("Une chaîne"))
# Une chaîne 'Une chaîne'
```

Formatages numériques :

```
n = 100
pi = 3.1415926535897931

print("{} et {}".format(n, pi)) # 100, et 3.14159265359
print("{}, et {}".format(n, pi)) # 100, et 3.14159265359
print("{} {1} et {}".format(n, pi)) # 100, 3.14159265359 et 100

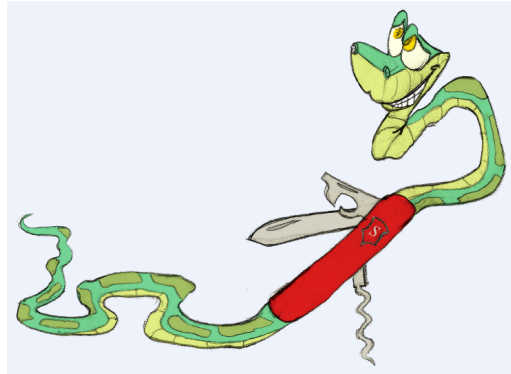
print("{:.4e}".format(pi)) # 3.1416e+00
print("{:g}".format(pi)) # 3.14159

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


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.....]
long = 12
print("{}".format(s[:long])) # [The sword of]
m = 123456789
print("{:0=12}".format(m))   # 000123456789
print("{:#=12}".format(m))   # ###123456789
```


Fonctions et espaces de noms



Remarque

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

Elles offrent différents avantages :

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

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

Permet la réutilisation : mécanisme de l'import ;

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

Ces avantages sont illustrés sur la figure 5.1.

5.1 Définition et syntaxe

Définition

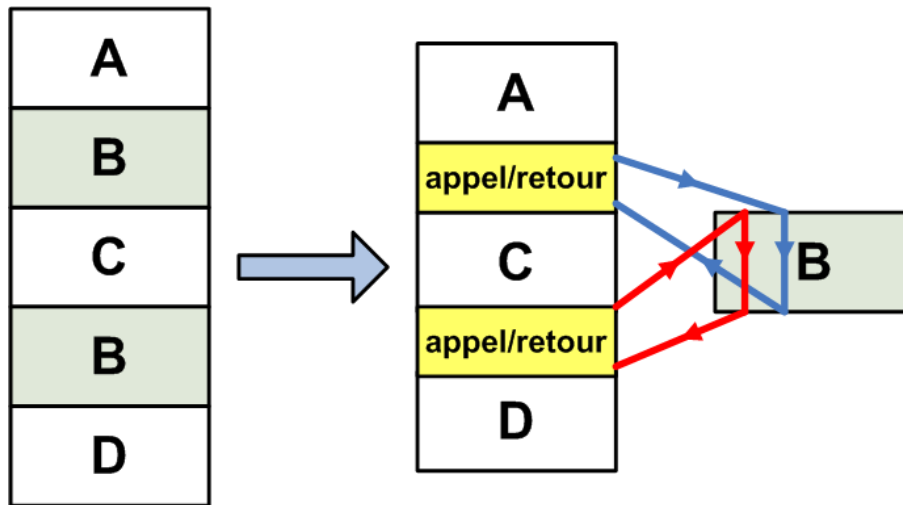
i Groupe d'instructions regroupé sous un *nom* (*définition*) et s'exécutant à la demande (*appel*).

Syntaxe

✎ C'est une instruction composée :

```
def nomFonction(paramètres):
    """Documentation de la fonction."""
    <bloc_instructions>
```

Le bloc d'instructions est **obligatoire**. S'il est vide, on emploie l'instruction `pass`. La documentation (facultative) est *fortement* conseillée.



(a) Évite la duplication de code.

```

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.

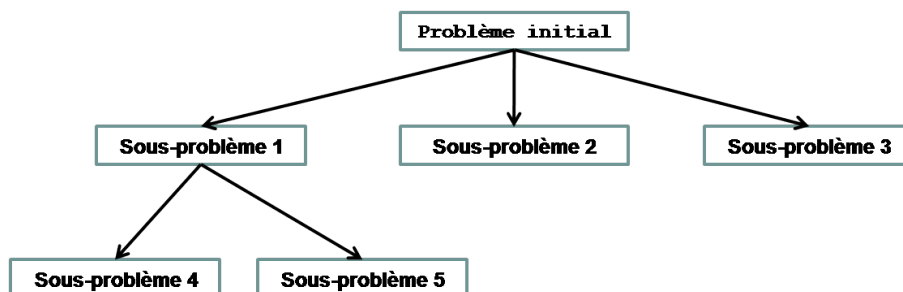
```
import util
```

```

...
p1 = util.proportion(uneChaine, "le")
...
p2 = util.proportion(uneAutreChaine, "des")

```

(c) L'import permet la réutilisation.



(d) Améliore la conception.

FIGURE 5.1 – Les avantages de l'utilisation des fonctions

5.2 Passage des arguments

5.2.1 Mécanisme général

Remarque

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

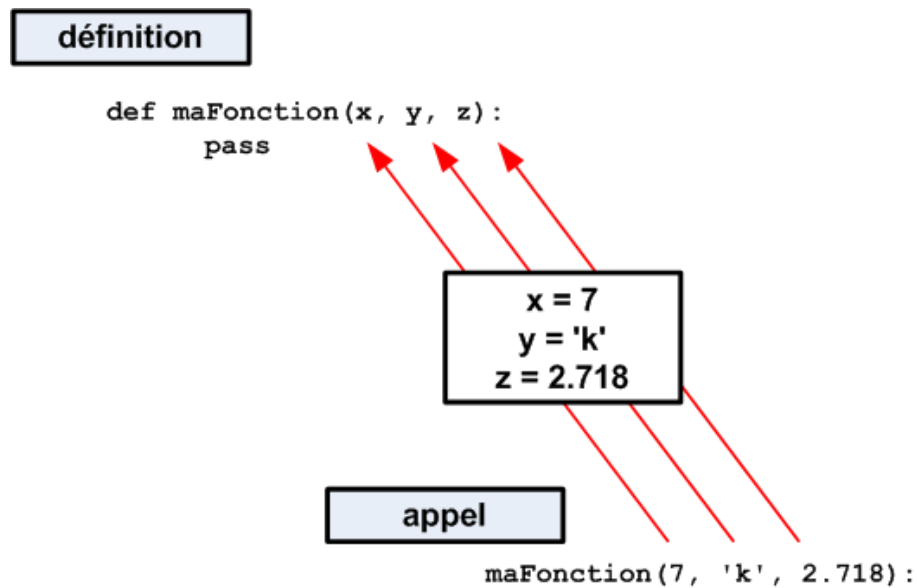


FIGURE 5.2 – Passage des arguments par affectation.

5.2.2 Un ou plusieurs paramètres, pas de retour

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

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

# exemple d'appel :
table(7, 2, 11)
# 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 9 x 7 = 63 10 x 7 = 70 11 x 7 = 77
```

5.2.3 Un ou plusieurs paramètres, utilisation du retour

Exemple avec utilisation d'un `return` unique :

```
from math import pi
```

```
def cube(x):
    return x**3

def volumeSphere(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))
```

5.2.4 Passage d'une fonction en paramètre

```
def tabuler(fonction, borneInf, borneSup, nbPas):
    """Affichage des valeurs de <fonction>.
    On doit avoir (borneInf < borneSup) et (nbPas > 0)"""
    h, x = (borneSup - borneInf) / float(nbPas), borneInf
    while x <= borneSup:
        y = fonction(x)
        print("f({:.2f}) = {:.2f}".format(x, y))
        x += h

def maFonction(x):
    return 2*x**3 + x - 5

tabuler(maFonction, -5, 5, 10)
# f(-5.00) = -260.00
# f(-4.00) = -137.00
# ...
# f(5.00) = 250.00
```

5.2.5 Paramètres avec valeur par défaut

On utilise de préférence des valeurs par défaut **non modifiables** car la modification d'un paramètre par un premier appel est visible les fois suivantes :

```
def initPort(speed=9600, parity="paire", data=8, stops=1):
    print("Init. à", speed, "bits/s", "parité :", parity)
    print(data, "bits de données", stops, "bits d'arrêt")

# Appels possibles :
initPort()
# Init. à 9600 bits/s parité : paire
# 8 bits de données 1 bits d'arrêt
```

```

initPort(parity="nulle")
# Init. à 9600 bits/s parité : nulle
# 8 bits de données 1 bits d'arrêt
initPort(2400, "paire", 7, 2)
# Init. à 2400 bits/s parité : paire
# 7 bits de données 2 bits d'arrêt

```

5.2.6 Nombre d'arguments arbitraire : passage d'un tuple

```

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

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

```

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

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

```

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

```

def unDict(**kargs):
    return kargs

# 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}

```

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

5.3 Espaces de noms

5.3.1 Portée des objets

Remarque

☞ Portée : les noms des objets sont créés lors de leur *première affectation*, mais ne sont visibles que dans certaines régions de la mémoire.

On distingue :

La portée globale : celle du module `__main__`. Un dictionnaire gère les objets globaux : l'instruction `globals()` fournit les couples `variable:valeur` ;

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

5.3.2 Résolution des noms : règle *LGI*

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

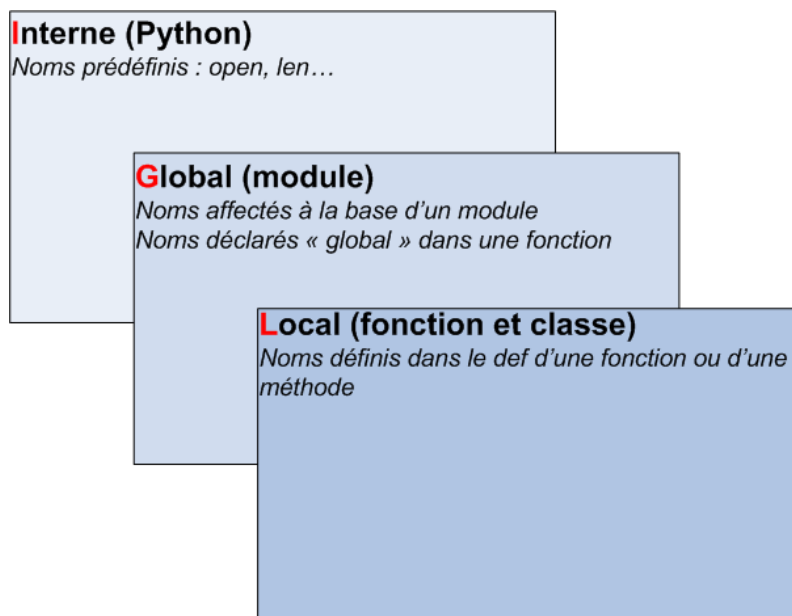


FIGURE 5.3 – Règle LGI

Exemples de portée

```
# 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 += 2
    z = x + y
    return z

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



```
# 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
```

```
# 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
```


Modules et packages



Un programme Python est généralement composé de plusieurs fichiers sources, appelés *modules*. Leur nom est suffixé `.py`.

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 et importer des modules dans un autre.

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

6.1 Modules

Définition

i Module : fichier *indépendant* permettant de scinder un programme en plusieurs scripts.

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 ;
- partition de l'espace de noms du système.

6.1.1 Import d'un module

Deux syntaxes possibles :

- la commande `import <nom_module>` importe la totalité des objets du module :

```
import tkinter
```

- la commande `from <nom_module> import obj1, obj2...` n'importe que les objets `obj1, obj2...` du module :

```
from math import pi, sin, log
```

Il est conseillé d'importer dans l'ordre :

- les modules de la bibliothèque standard ;
- les modules des bibliothèques tierces ;
- Les modules personnels.

6.1.2 Exemples

Notion d'« auto-test »

Un module `cube_m.py`. Remarquez l'utilisation de « l'auto-test » qui permet de tester le module seul :

```
def cube(y):
    """Calcule le cube du paramètre <y>."""
    return y**3

# Auto-test -----
if __name__ == "__main__": # False lors d'un import ==> ignoré
    help(cube)             # affiche le docstring de la fonction
    print("cube de 9 :", cube(9)) # cube de 9 : 729
```

Utilisation de ce module. On importe la fonction `cube()` incluse dans le fichier `cube_m.py` :

```
from cube_m import cube
for i in range(1, 4):
    print("cube de", i, "=", cube(i), end=" ")
# cube de 1 = 1 cube de 2 = 8 cube de 3 = 27
```

Une interface à gnuplot

L'application libre `gnuplot` permet d'afficher des courbes. La fonction suivante est une interface d'appel qui permet d'afficher des données issues de fichiers :

```
import os

def plotFic(courbes):
    dem = open("_.dem", "w") # fichier réutilisé à chaque tracé
    dem.write("set grid\n")
    plot_data = ["'%s' with %s" % (c[0], c[1]) for c in courbes]
    dem.write("plot " + ','.join(plot_data))
    dem.write('\npause -1 "\'Entrée\' pour continuer'\n')
    dem.write("reset")
    dem.close()
    os.system("wgnuplot _.dem")
```

L'auto-test suivant illustre son utilisation :

```
if __name__ == '__main__':
    f, g, h = open("d1.dat", "w"), open("d2.dat", "w"), open("d3.dat", "w")

    for i in range(201):
        x = 0.1*i - 5.0
        y = x**3 - 20*x**2
```

```

f.write("%g %g\n" %(x, y))
y = x**3 - 30*x**2
g.write("%g %g\n" %(x, y))
y = x**3 - 40*x**2
h.write("%g %g\n" %(x, y))
h.close(); g.close(); f.close()
plotFic([('d1.dat', 'points')])
plotFic([('d1.dat', 'lines'), ('d2.dat', 'points'),
        ('d3.dat', 'lines')])

```

6.2 Bibliothèque standard

6.2.1 La bibliothèque standard

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`... et la classe `Formatter` qui peut être spécialisée en sous-classes spécialisées 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 peut oublier le module `re` qui offre à Python la puissance des expressions régulières.

Exemple : le module `io.StringIO`

Ce module fournit des objets compatibles avec l'interface des objets fichiers.

Exemple de gestion ligne à ligne d'un fichier ou d'une chaîne avec la même fonction `scanner()` utilisant le même traitement :

```

def scanner(objet_fichier, gestionnaire_ligne):
    for ligne in objet_fichier:
        gestionnaire_ligne(ligne)

if __name__=='__main__':
    def premierMot(ligne): print(ligne.split()[0])

    fic = open("data.dat")
    scanner(fic, premierMot)

```

```
import io
chaine = io.StringIO("un\ndeux xxx\ntrois\n")
scanner(chaine, premierMot)
```

La gestion de la ligne de commande

La gestion est assurée par deux modules : `getopt`, le module historique hérité du C et `optparse`, un module récent beaucoup plus puissant :

```
from optparse import OptionParser
parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
                  help="write report to FILE", metavar="FILE")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose", default=True,
                  help="don't print status messages to stdout")

(options, args) = parser.parse_args()
```

Les lignes de commande :

```
python 6_025.py -h
```

ou

```
python 6_025.py --help
```

produisent la même documentation :

```
Usage: 6_025.py [options]

Options:
  -h, --help            show this help message and exit
  -f FILE, --file=FILE write report to FILE
  -q, --quiet           don't print status messages to stdout
```

Bibliothèques mathématiques et types numériques

En standard, Python propose les modules `fraction` et `decimal` :

```
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.

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'indice 0 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 :

```
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]
```

Il est bien sûr possible d'avoir des tuples nommés emboîtés.

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 domaines pourraient être explorés :

- accès au système ;
- utilitaires fichiers ;
- programmation réseau ;
- persistance ;
- les fichiers XML ;
- la compression ;
- ...

6.3 Bibliothèques tierces

6.3.1 Une grande diversité

Outre les 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 (*The Python Package Index*) recense des milliers de modules et de packages !

6.3.2 Un exemple : la bibliothèque Unum

Elle permet de calculer en tenant compte des unités du système S.I.

Voici un exemple de session interactive :

```

-- Welcome in Unum Calculator (ver 04.00) --
>>> d = 1609 * M
>>> t = 11.7 * S
>>> v = d/t
>>> v
137.521367521 [m/s]
>>> a = v/t
>>> a

```



```
11.753963036 [m/s2]  
>>>
```

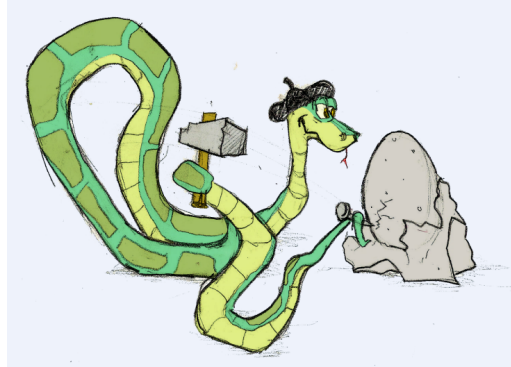
6.4 Packages

Définition

i 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.

En résumé, un package est simplement un répertoire qui contient des modules et un fichier `__init__.py` décrivant l'arborescence du package.

La programmation Orientée Objet



La *Programmation Orientée Objet* :

- la *POO* permet de mieux modéliser la réalité en concevant des ensembles 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 Insuffisance de l'approche procédurale

Un exemple

On veut représenter un cercle, ce qui nécessite au minimum trois informations, les coordonnées du centre et le rayon :

```
cercle = (11, 60, 8)
```

Mais comment interpréter ces trois données ?

```
cercle = (x, y, rayon)
# ou bien
cercle = (rayon, x, y)
```

Pour résoudre ce problème et améliorer la lisibilité, on peut utiliser des tuples nommés :

```
from collection import namedtuple

Cercle = namedtuple("Cercle", "x y rayon")
cercle = Cercle(11, 60, 8)
# exemple d'utilisation :
distance = distance_origine(cercle.x, cercle.y)
```

Par contre, il reste le problème des données invalides, ici un rayon négatif :

```
cercle = Cercle(11, 60, -8)
```

Si les cercles doivent changer de caractéristiques, il faut opter pour un type modifiable, liste ou dictionnaire ce qui ne règle toujours pas le problème des données invalides...

On a donc besoin d'un mécanisme pour empaqueter les données nécessaires pour représenter un cercle *et* pour empaqueter les méthodes applicables à ce nouveau type de données (la *classe*), de telle sorte que seules les opérations valides soient utilisables.

7.2 Terminologie

Le vocabulaire de la POO

Une **classe** est donc équivalente à un **nouveau type de données**. On connaît déjà par exemple `int` ou `str`.

Un **objet** ou une **instance** est un exemplaire particulier d'une classe. Par exemple "truc" est une instance de la classe `str`.

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 comme `upper()`.

On pourrait définir un objet comme une *capsule*, à savoir un « paquet » contenant des attributs et des méthodes : `objet = [attributs + méthodes]`

Beaucoup de classes offrent des caractéristiques supplémentaires comme par exemple la concaténation des chaînes en utilisant simplement l'opérateur `+`. Ceci est obtenu grâce aux **méthodes spéciales**. Par exemple l'opérateur `+` est utilisable car on a redéfini la méthode `__add().__`.

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 :

- `.imag` et `.real`, ses attributs ;
- beaucoup de méthodes, comme `conjugate()` ;
- des méthodes spéciales : `+`, `-`, `/`...

Les attributs sont normalement implémentés comme des **variables d'instance**, particulières à chaque instance d'objet.

Le mécanisme de **property** permet un accès contrôlé aux données, ce qui permet de les valider et de les sécuriser.

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 **supper 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 Python s'adaptera dynamiquement, dès l'affectation. 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.

Enfin Python supporte également le *duck typing* : « s'il marche comme un canard et cancale comme un canard, alors c'est un canard! ». Ce qui signifie que Python ne s'intéresse qu'au *comportement* des objets. Par exemple un objet fichier peut être créé par `open()` ou par une instance de `io.StringIO`. Les deux approches offrent la même API (interface de programmation), c'est-à-dire les mêmes méthodes.

7.3 Classes et instantiation d'objets

7.3.1 L'instruction class

Syntaxe

 Instruction composée : en-tête (avec *docstring*) + corps indenté :

```
class C:
    """Documentation de la classe."""
    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`.

7.3.2 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 (i.e. création, production depuis l'*usine*) en appelant le nom de sa classe :

```
a = C()          # a est un objet de la classe C
print(dir(a))   # affiche les attributs de l'objet a
print(a.x)      # affiche 23. x est un attribut de classe
a.x = 12        # modifie l'attribut d'instance (attention...)
print(C.x)      # 23, l'attribut de classe est inchangé
a.y = 44        # nouvel attribut d'instance

b = C()          # b est un autre objet de la classe C
print(b.x)      # 23. b connaît son attribut de classe, mais...
print(b.y)      # AttributeError: C instance has no attribute 'y'
```

7.3.3 Retour sur les espaces de noms

Tout comme les fonctions, les classes possèdent leurs espaces de noms :

- Chaque classe possède son propre espace de noms. Les variables qui en font partie sont appelées *attributs de classe*.
- Chaque objet instance (créé à partir d'une classe) obtient son propre espace de noms. Les variables qui en font partie sont appelées *attributs d'instance*.
- Les classes peuvent utiliser (mais pas modifier) les variables définies au niveau principal.

- Les instances peuvent utiliser (mais pas modifier) les variables définies au niveau de la classe et les variables définies au niveau principal.

Les espaces de noms sont implémentés par des *dictionnaires* pour les modules, les classes et les instances.

- **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).


L'exemple suivant affiche le dictionnaire lié à la classe `C` puis la liste des attributs liés à une instance de `C` :

```
class C:
    x = 20

print(C.__dict__) # {'__dict__': <attribute '__dict__' of 'C' objects>, 'x': 20,
                  '__module__': '__main__', '__weakref__': <attribute '__weakref__' of 'C'
                  objects>, '__doc__': None}
a = C()
print(dir(a))    # ['__class__', '__delattr__', '__dict__', '__doc__', '
                  __getattr__', '__hash__', '__init__', '__module__', '__new__', '
                  __reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__str__', '
                  __weakref__', 'x']
```

7.4 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 = 23          # x et y : attributs de classe
    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

ob = C()          # instantiation de l'objet ob
ob.affiche()     # 28 42 (à l'appel, ob affecte self)
```

7.5 Méthodes spéciales

7.5.1 Les méthodes spéciales

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

Elles servent :

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

7.5.2 L'initialisateur

Lors de l'instanciation d'un objet, la méthode `__init__` est automatiquement invoquée. Elle permet d'effectuer toutes les initialisations nécessaires :

```
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
print(une_instance.x) # 42
```

C'est une *procédure* automatiquement invoquée lors de l'instanciation : elle ne contient *jamais* l'instruction `return`.

7.5.3 Surcharge des opérateurs

La *surcharge* permet à un opérateur de posséder un sens différent suivant le type de leurs opérands. 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 :

Nom	Méthode spéciale	Utilisation
opposé	<code>__neg__</code>	<code>-obj1</code>
addition	<code>__add__</code>	<code>obj1 + obj2</code>
soustraction	<code>__sub__</code>	<code>obj1 - obj2</code>
multiplication	<code>__mul__</code>	<code>obj1 * obj2</code>
division	<code>__div__</code>	<code>obj1 / obj2</code>

7.5.4 Exemple de surcharge

```
class Vecteur2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, autre): # addition vectorielle
        return Vecteur2D(self.x + autre.x, self.y + autre.y)
```

```

def __str__(self):          # affichage d'un Vecteur2D
    return "Vecteur({:g}, {:g})" % (self.x, self.y)

v1 = Vecteur2D(1.2, 2.3)
v2 = Vecteur2D(3.4, 4.5)

print(v1 + v2)            # Vecteur(4.6, 6.8)

```

7.6 Héritage et polymorphisme

7.6.1 Héritage et polymorphisme

Définition

i L'*héritage* est le mécanisme qui permet de se servir d'une classe préexistante pour en créer une nouvelle qui possédera des fonctionnalités différentes ou supplémentaires.

Définition

i Le *polymorphisme* est la faculté pour une méthode portant le même nom mais appartenant à des classes distinctes héritées d'effectuer un travail différent. Cette propriété est acquise par la technique de la surcharge.

7.6.2 Exemple d'héritage et de polymorphisme

Dans l'exemple suivant, la classe `Carre` hérite de la classe `Rectangle`, et la méthode `__init__` est polymorphe :

```

class Rectangle:
    def __init__(self, longueur=30, largeur=15):
        self.L, self.l, self.nom = longueur, largeur, "rectangle"

class Carre(Rectangle):
    def __init__(self, cote=10):
        Rectangle.__init__(self, cote, cote)
        self.nom = "carré"

r = Rectangle()
print(r.nom)    # 'rectangle'
c = Carre()
print(c.nom)    # 'carré'

```

7.7 Retour sur l'exemple initial

7.7.1 La classe Cercle : conception

Nous allons tout d'abord concevoir une classe `Point` héritant de la classe mère `object`. Puis nous pourrons l'utiliser comme classe de base de la classe `Cercle`. Dans les schémas UML¹ ci-dessous, les attributs en italiques sont hérités, ceux en casse normale sont nouveaux et ceux en gras sont redéfinis (surchargés).

1. *Unified Modeling Language* : notation graphique de conception objet.

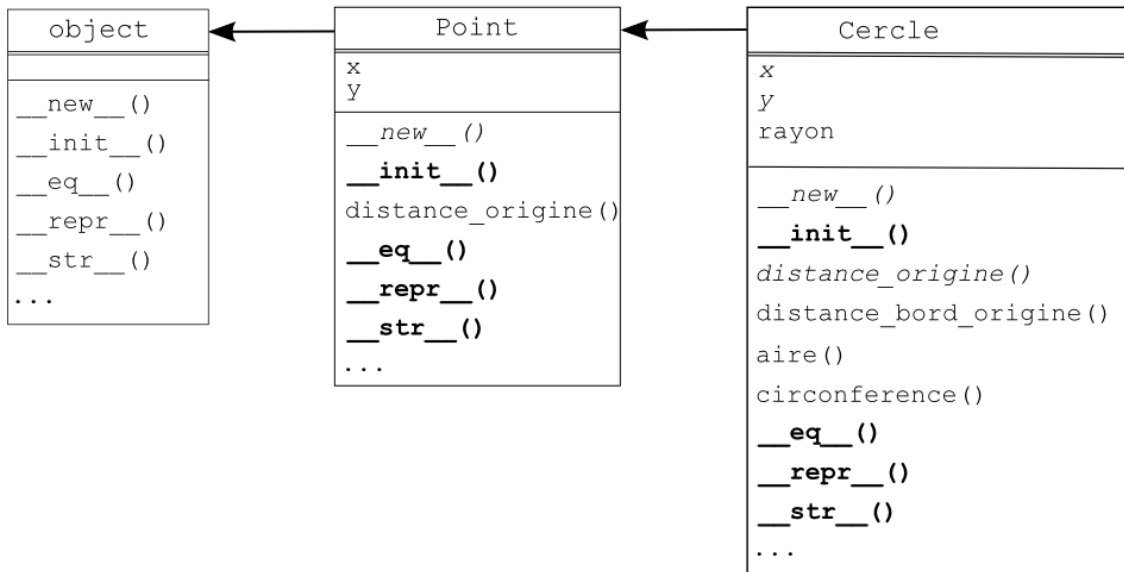


FIGURE 7.1 – Conception UML de la classe Cercle.

7.7.2 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 "{0.x!s}, {0.y!s}".format(self)
  
```

L'utilisation du décorateur `property` permet un accès en *lecture seule* au résultat de la méthode `distance_origine()` considérée alors comme un simple attribut (car il n'y a pas de parenthèse) :

```

if __name__ == "__main__":
    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

```

7.8 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 :

- la **composition** qui repose sur la relation **a-un** ou sur la relation **utilise-un** ;
- la **dérivation** qui repose sur la relation **est-un**.

Bien sûr, ces deux conceptions peuvent cohabiter, et c'est souvent le cas !

7.8.1 Composition

Définition

i La composition est la collaboration de plusieurs classes distinctes via une *association* (utilise-un) ou une *agrégation* (a-un).

La classe composite bénéficie de l'ajout de fonctionnalités d'autres classes qui n'ont rien en commun.

L'implémentation Python utilisée est généralement l'instanciation de classes dans le constructeur de la classe composite.

Exemple

```
class Point:
    def __init__(self, x, y):
        self.px, self.py = x, y

class Segment:
    """Classe composite utilisant la classe distincte Point."""
    def __init__(self, x1, y1, x2, y2):
        self.orig = Point(x1, y1) # Segment "a-un" Point origine,
        self.extrem = Point(x2, y2) # et "a-un" Point extrémité

    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)]
```

7.8.2 Dérivation

Définition

i La dérivation décrit la création de sous-classes par spécialisation.

On utilise dans ce cas le mécanisme de l'*héritage*.

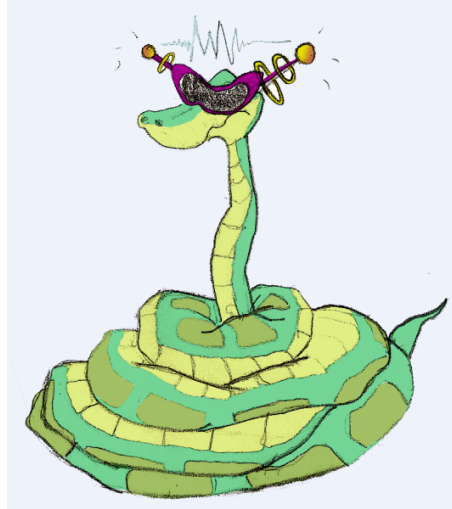
L'implémentation Python utilisée est généralement l'appel dans le constructeur de la classe dérivée du constructeur de la classe parente, soit nommément, soit grâce à l'instruction `super`.

Exemple

```
class Rectangle:
    def __init__(self, longueur=30, largeur=15):
        self.L, self.l, self.nom = longueur, largeur, "rectangle"

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
```


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.

8.1 Techniques procédurales

8.1.1 Améliorer la documentation

La fonction utilitaire `printApi()`¹ filtre les méthodes disponibles de `element`, et affiche les *docstrings* associés sous une forme plus lisible que `help()` :

```
def printApi(element):
    methods = [el for el in dir(element) if not el.startswith('_')]
    for meth in methods:
        print(getattr(element, meth).__doc__)

if __name__ == "__main__":
    printApi([])

"""
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*
```

1. Cf. [B6] p. 132

```
L.sort(cmp=None, key=None, reverse=False) -- stable sort *IN PLACE*;
cmp(x, y) -> -1, 0, 1
"""
```

8.1.2 Faire des menus avec un dictionnaire

On peut utiliser le couple `clé : valeur` d'un dictionnaire pour implémenter un menu, de la façon suivante :

- on donne à `cle` le nom d'un élément du menu ;
- la `valeur` correspondante est une référence : l'appel à une procédure sans argument.

L'exemple proposé est la programmation d'une queue *FIFO*, une structure de données illustrée par la file d'attente à un guichet : le premier arrivé est le premier servi.

Source du module de fonctions :

```
"""Module de gestion d'une queue FIFO."""
queue = [] # initialisation

def enQueue():
    queue.append(int(input("Entrez un entier : ")))

def deQueue():
    if len(queue) == 0:
        print("\nImpossible : la queue est vide !")
    else:
        print("\nÉlément '%d' supprimé" % queue.pop(0))

def afficheQueue():
    print("\nqueue :", queue)
```

Source du menu de gestion :

```
"""Implémentation d'une queue FIFO avec une liste.

Un menu (utilisant un dictionnaire)
appelle des procédures sans argument.
"""

# import
from queue_FIFO_menu_m import enQueue, deQueue, afficheQueue

# programme principal -----
afficheQueue()

CMDs = {'a':enQueue, 'v':afficheQueue, 's':deQueue}

menu = """
(A)jouter
(V)oir
(S)upprimer
(Q)uitter

Votre choix ? """
```

```

while True:
    while True:
        try:
            choix = input(menu).strip()[0].lower()
        except:
            choix = 'q'

        if choix not in 'avsq':
            print("Option invalide ! Réessayez")
        else:
            break

    if choix == 'q':
        print("\nAu revoir !")
        break
    CMDs[choix]()

```

8.1.3 Les fonctions récursives

Définition

 Une fonction récursive peut 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.

Bien que cette méthode soit souvent plus difficile à comprendre et à coder que la méthode classique dite *itérative*, elle est, dans certains cas, l'application la plus directe de sa définition mathématique.

Voici l'exemple classique de définition par récurrence de la fonction factorielle¹ :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n(n-1)! & \text{si } n \geq 1 \end{cases}$$

Remarque

 Son code est *très exactement* calqué sur sa définition mathématique :

```

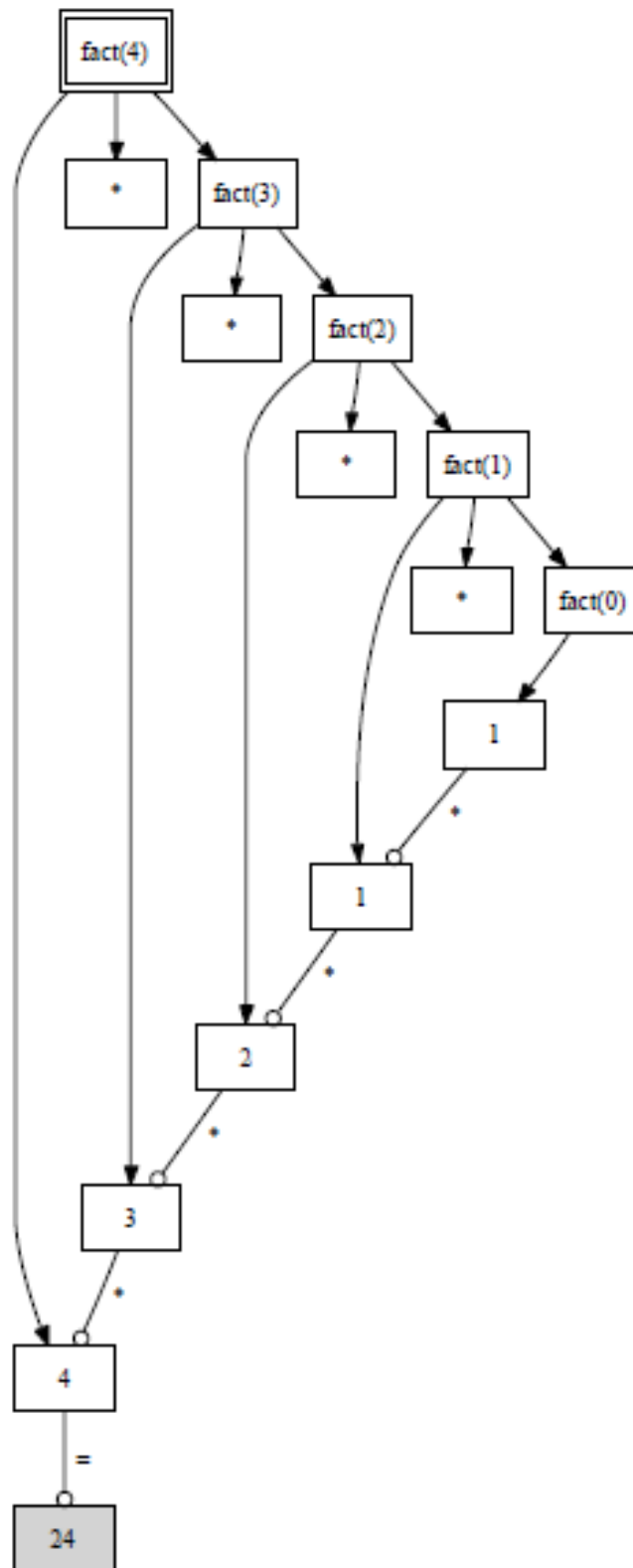
def factorielle(n):
    if n == 0:      # cas de base : condition terminale
        return 1
    else           # cas récursif
        return n*factorielle(n-1)

```

Dans cette définition, la valeur de $n!$ n'est pas connue tant que l'on n'a pas atteint la condition terminale (ici $n == 0$). Le programme *empile* les appels récursifs jusqu'à atteindre la condition terminale puis *dépile* les valeurs.

Ce mécanisme est illustré par la figure 8.1.

1. Par opposition à sa définition *itérative* : $n! = n(n-1)(n-2)\dots \times 2 \times 1$.

FIGURE 8.1 – Empilage/dépilage de $4!$

8.1.4 Les générateurs et les expressions génératrices

Les générateurs

Les générateurs fournissent un moyen de générer des *exécutions paresseuses*, ce qui signifie qu'elles ne calculent que les valeurs réellement demandées. Ceci peut s'avérer beaucoup plus efficace (en terme de mémoire) que le calcul, par exemple, d'une énorme liste en une seule fois.

Voici un exemple de générateur qui fournit autant de valeurs que demandées :

```
def quarters(next_quarter=0.0):
    while True:
        yield next_quarter
        next_quarter += 0.25

if __name__ == "__main__":
    result = []
    for x in quarters():
        result.append(x)
        if x == 1.0:
            break

    print("Liste résultante : ", result)
    # Liste résultante : [0.0, 0.25, 0.5, 0.75, 1.0]
```

Il est aussi possible de passer une initialisation au générateur :

```
# import
import sys


def quarters(next_quarter=0.0):
    while True:
        received = (yield next_quarter)
        if received is None:
            next_quarter += 0.25
        else:
            next_quarter += received

if __name__ == "__main__":
    result = []
    generator = quarters()
    while len(result) < 5:
        x = next(generator)
        if abs(x - 0.5) < sys.float_info.epsilon:
            x = generator.send(1.0) # réinitialise le générarteur
        result.append(x)

    print("Liste résultante : ", result)
    # Liste résultante : [0.0, 0.25, 1.5, 1.75, 2.0]
```

Les expressions génératrices

Syntaxe

 Une expression génératrice possède une syntaxe presque identique à celle des listes en intension ; la différence est qu'une expression génératrice est entourée de parenthèses.

Utilisation

Les expressions génératrices sont aux générateurs ce que les listes en intension sont aux fonctions.

8.1.5 Les fonctions incluses

La syntaxe de définition des fonctions en Python permet tout à fait d'*emboîter* leur définition. Distinguons deux cas d'emploi :

- Idiomme 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

# Programme principal -----
## création de deux fabriques distinctes
p = creer_plus(23)
q = creer_plus(42)
## utilisation
print("p(100) =", p(100))
print("q(100) =", q(100))
```

- Fonction fabrique renvoyant une classe :

```
# classes
class CasNormal(object):

    def uneMethode(self):
        print("normal")

class CasSpecial(object):

    def uneMethode(self):
        print("spécial")

# fonction
def casQuiConvient(estNormal=True):
    """Fonction fabrique renvoyant une classe."""
    return CasNormal() if estNormal else CasSpecial()

# Programme principal -----
une_instance = casQuiConvient()
une_instance.uneMethode() # normal

autre_instance = casQuiConvient(False)
autre_instance.uneMethode() # spécial
```

8.1.6 Les décorateurs

Remarque

☞ On utilise un décorateur lorsqu'on a besoin d'effectuer un **prétraitement** lors de l'appel d'une fonction.

Soit le prétraitement suivant :

```
def pretraitement(fonction):
    fonction.__doc__ += "(fonction décorée)."
    return fonction

def traitement():
    """ma fonction """
    print("traitement")

traitement = pretraitement(traitement)
print(traitement.__doc__) # ma fonction (fonction décorée).
```

Nous obtenons le même résultat en utilisant un décorateur :

```
def pretraitement(fonction):
    fonction.__doc__ += "(fonction décorée)."
    return fonction

@pretraitement
def traitement():
    """ma fonction """
    print("traitement")

print(traitement.__doc__) # ma fonction (fonction décorée).
```

Enfin il est possible d'enchaîner les décorateurs (à l'image de la composition des fonctions en mathématique) :

```
@f1 @f2 @f3
def fonction():
    pass
# Notation équivalente à : fonction = f1(f2(f3(fonction)))
```

8.2 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. Toutes ces classes dérivent d'une unique classe de base, ancêtre de tous les autres : la classe `object`.

8.2.1 `__slots__` et `__dict__`

Examinons le code suivant :

```
class Point:
    __slots__ = ("x", "y")
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
```

Quand une classe est créée *sans* utiliser l'instruction `__slot__`, ce que nous avons fait jusqu'à maintenant, Python crée de manière transparente un dictionnaire privé appelé `__dict__` pour chaque instance de la classe, et ce dictionnaire contient les attributs de l'instance. Voici pourquoi il est possible d'ajouter ou de retirer des attributs d'un objet.

Mais si l'on se contente d'objets sur lesquels nous accédons aux attributs sans en ajouter ou en ôter, nous pouvons créer des classes sans dictionnaire privé, ce qui économisera de la mémoire à chaque instanciation. C'est ce qui est réalisé dans l'exemple ci-dessus en définissant un attribut de classe `__slot__` dont la valeur est un tuple formé des noms des attributs.

8.2.2 Functor

En Python un objet fonction ou *functor* est une référence à tout objet « callable »¹ : fonction, fonction lambda, méthode, classe. La fonction prédéfinie `callable()` permet de tester cette propriété :

```
>>> def maFonction():
    print('Ceci est "appelable"')

>>> callable(maFonction)
True
>>> chaine = 'Ceci est "appelable"'
>>> callable(chaine)
False
```

Il est possible de transformer les instances d'une classe en functor si la méthode spéciale `__call__()` est définie dans la la classe :

```
>>> class A:
    def __call__(self, un , deux):
        return un + deux

>>> a = A()
>>> callable(a)
True
>>> a(1, 6)
7
```

8.2.3 Les gestionnaires de contexte

Les gestionnaires de contexte simplifient nos codes en assurant que certaines opérations sont effectuées avant et après un bloc d'instruction donné.

Syntaxe



On utilise l'instruction `with`

```
with <expression> as <variable>:
    <instructions>
```

Une utilisation classique est d'assurer la fermeture d'un fichier :

1. *callable* en anglais.


```
with open("hello.txt") as f:
    for line in f:
        print line
```

8.3 Techniques fonctionnelles

8.3.1 Directive lambda

Issue de langages fonctionnels (comme Lisp), la directive `lambda` permet de définir un objet *fonction anonyme* dont le retour est limité à **une expression**.

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"
```

Dans la présentation des interfaces graphiques, on verra que les *callbacks* sont souvent codés ainsi.

```
f = lambda x=1, y=1, z=0: 3*x + 2*y - z
print(f())           # 5

def make_increment(n):
    return lambda x: x + n

f2, f6 = make_increment(2), make_increment(6)
print(f2(42), f6(42)) # 44 48

lc = [lambda x: x**2, lambda x: x**3, lambda x: x**4]
for f in lc:
    print(f(2), end=" ") # 4 8 16
```

8.3.2 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 et rejette le changement d'état et la mutation des données. Elle souligne l'application des fonctions, contrairement au modèle de programmation impérative qui met en avant les changements d'état¹.

Le paradigme fonctionnel n'utilise pas de machine d'états pour décrire un programme, mais un emboîtement de fonctions que l'on peut voir comme des « boîtes noires » que l'on peut imbriquer les unes dans les autres. Chaque boîte possédant plusieurs paramètres en entrée mais une seule sortie, elle ne peut sortir qu'une seule valeur possible pour chaque n-uplet de valeurs présentées en entrée. Ainsi, les fonctions n'introduisent pas d'effets de bord. Un programme est donc une application, au sens mathématique, qui ne donne qu'un seul résultat pour chaque ensemble de valeurs en entrée².

1. Cf. Wikipedia

2. Idem

La programmation fonctionnelle 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 :

```
>>> def renvoiTitres(element):
    return element.title()

>>> map(renvoiTitres, ['fiat lux', "lord of the fly"])
['Fiat Lux', 'Lord Of The Fly']
>>>
>>> [element.title() for element in ['fiat lux', "lord of the fly"]]
['Fiat Lux', 'Lord Of The Fly']
```

On remarque que `map()` peut être remplacée par une liste en intension.

La fonction `filter()` :

`filter()` construit et renvoie un itérateur sur une liste qui contient tous les éléments de la séquence initiale répondant au critère :

fonction(element) == True :

```
>>> filter(lambda x: x > 0, [1, -2, 3, -4])
[1, 3]
>>>
>>> [x for x in [1, -2, 3, -4] if x > 0]
[1, 3]
```

On remarque que `filter()` peut être remplacée par une liste en intension avec un test.

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
```

On remarque que `reduce()` peut être remplacée par une des fonctions suivantes : `all()`, `any()`, `max()`, `min()` ou `sum()`.

8.3.3 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.

Exemple simple :

```
>>> from functools import partial
>>> baseTwo = partial(int, base=2)
>>> baseTwo('10010')
18
```

Les PFA sont très 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) :

```
from functools import partial
import tkinter as tk

root = tk.Tk()
# instantiation partielle de classe :
MonBouton = partial(tk.Button, root, fg='purple', bg='green')
MonBouton(text="Bouton 1").pack()
MonBouton(text="Bouton 2").pack()
MonBouton(text="QUITTER", bg='red', fg='black',
          command=root.quit).pack(fill=tk.X, expand=True)
root.title("PFA !")
root.mainloop()
```

Ce résultat est illustré par la figure 8.2.

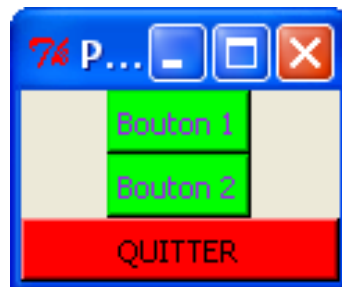
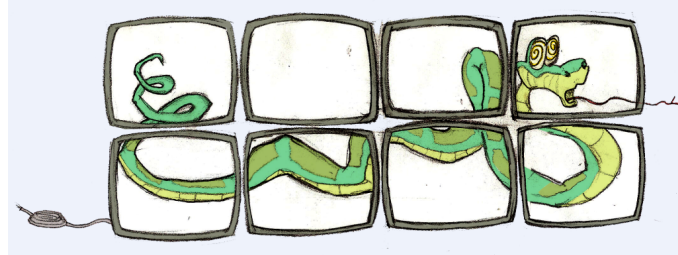


FIGURE 8.2 – Application partielle d'un widget

La programmation « OO » graphique



Très utilisée 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 très employée, car elle est installée de base dans toute les distributions Python.

`tkinter` facilite la construction d'interfaces graphiques simples. Après avoir importé la bibliothèque, la méthode consiste à créer, configurer et positionner les widgets utilisés, à coder les fonctions/méthodes associées aux widgets, puis d'entrer dans la boucle principale de gestion des événements.

9.1 Programmes pilotés par des événements

En programmation graphique objet, on remplace le déroulement *séquentiel* du script par une **boucle d'événements** (cf. Fig. 9.1)

9.2 La bibliothèque tkinter

Un exemple simple (cf. Fig. 9.2)

```
import tkinter

widget = tkinter.Label(None, text='Bonjour monde graphique !')
widget.pack()      # positionnement du label
widget.mainloop() # lancement de la boucle d'événements
```

9.2.1 Les widgets de tkinter

Définition

i 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

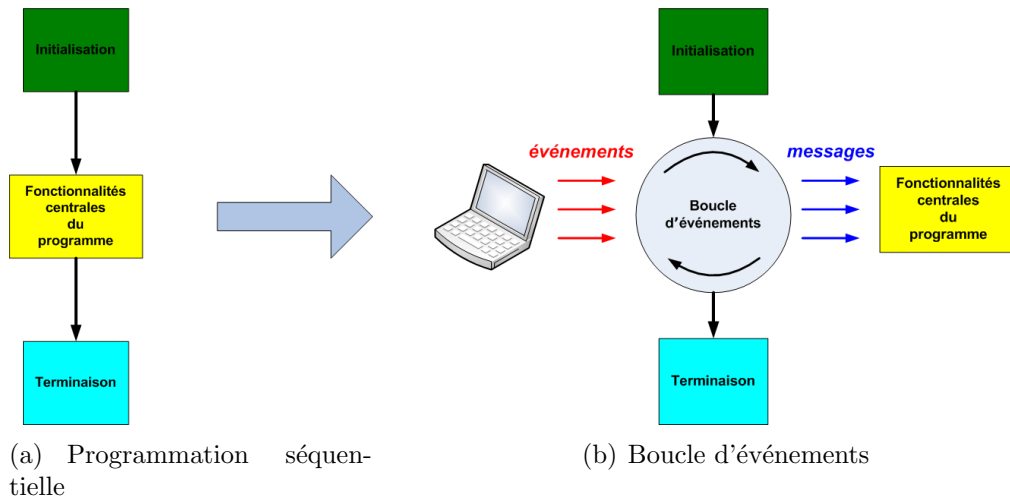


FIGURE 9.1 – Deux styles de programmation.



FIGURE 9.2 – Un exemple simple

Label zone de message

Button bouton 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

Scale glissière à plusieurs positions

PhotoImage sert à placer des images sur des widgets

BitmapImage sert à placer des *bitmaps* sur des widgets

Menu associé à un **Menubutton**

Menubutton bouton ouvrant un menu d'options

Scrollbar ascenseur

Listbox liste de noms à sélectionner

Text édition de texte multi-lignes

Canvas zone de dessins graphiques ou de photos

OptionMenu liste déroulante

ScrolledText widget **Text** avec ascenseur

PanedWindow interface à onglets

LabelFrame contenant avec un cadre et un titre

Spinbox un widget de sélection multiple

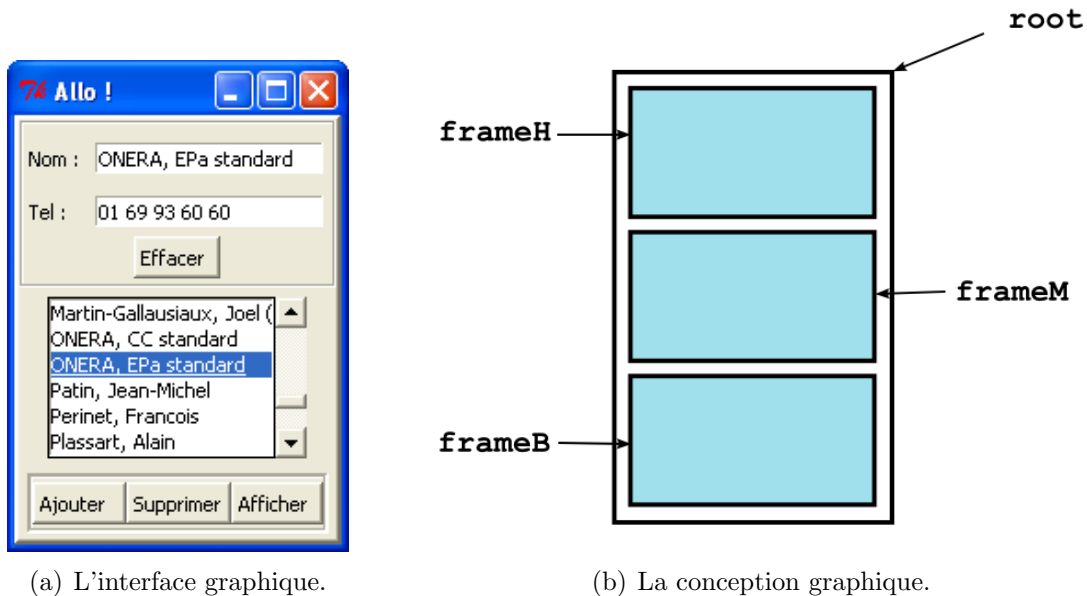


FIGURE 9.3 – tkPhone.

9.2.2 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 w dans un widget conteneur exactement selon ce que demande w . C'est un placement absolu (usage peu fréquent).

9.3 Deux exemples

9.3.1 tkPhone, un exemple sans menu

Il s'agit de créer un script de gestion d'un carnet téléphonique. L'aspect de l'application est illustré Fig. 9.3

Conception graphique

La conception graphique va nous aider à choisir les bons widgets. Tout d'abord, il est naturel de distinguer trois zones :

1. la zone supérieure, dédiée à l'affichage ;
2. la zone médiane est une liste alphabétique ordonnée ;
3. la 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 toute trois incluses dans une instance de `Tk()` (cf. conception Fig. 9.3).

Le code de l'interface graphique

Afin de ne pas mélanger les problèmes, on se propose la méthodologie suivante :

Remarque

☞ Séparer le codage de l'interface graphique et celui des *callbacks*.

Voici dans un premier temps le code de l'interface graphique. L'initialisateur crée un 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 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 :

```
import tkinter as tk

# class
class Allo_IHM(object):
    """IHM de l'application 'répertoire téléphonique'."""
    def __init__(self, fic):
        """Initialisateur/lanceur de la fenêtre de base"""
        self.phoneList = []
        self.fic = fic
        f = open(fic)
        try:
            for line in f:
                self.phoneList.append(line[: -1].split('*'))
        except: # création du fichier de répertoire
            f = open(self.fic, "w")
        finally:
            f.close()
        self.phoneList.sort()
        self.root = tk.Tk()
        self.root.title("Allo !")
        self.root.config(relief=tk.RAISED, bd=3)
        self.makeWidgets()
        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__':
    app = Allo_IHM('phones.txt') # instancie l'IHM

```

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.

```

# imports
import tkinter as tk
from tkPhone_IHM import Allo_IHM

```

```

# classes
class Allo(Allo_IHM):
    """Répertoire téléphonique."""
    def __init__(self, fic='phones.txt'):
        "Constructeur de l'IHM."
        Allo_IHM.__init__(self, 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, tk.END)
        for i in self.phoneList:
            self.select.insert(tk.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, tk.END)
        for i in self.phoneList:
            self.select.insert(tk.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, tk.END)
        self.phoneEnt.delete(0, tk.END)

# programme principal -----
app = Allo() # instancie l'application

```

9.3.2 IDLE, un exemple avec menu

Toutes les distributions Python comporte l'application IDLE, l'interpréteur/éditeur écrit en Python¹. Cette application se présente sous l'aspect d'une interface graphique complète (cf Fig. 9.4), avec menu.

C'est un source Python dont le code est disponible² et constitue à lui seul un cours complet à `tkinter`.

1. Dans certaines distributions linux, IDLE est un package particulier.

2. Mais il est trop volumineux pour être reproduit dans ces notes...

```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> def somme(a, b):
>>>     return a+b
>>> somme(21, 53)
74
>>> |
Ln: 8 Col: 4
```

(a) L'interpréteur d'IDLE

```
idle.pyw - C:\Python31\Lib\idlelib\idle.pyw
File Edit Format Run Options Windows Help
try:
    import idlelib.PyShell
except ImportError:
    # IDLE is not installed, but maybe PyShell is on sys.path:
    try:
        from . import PyShell
    except ImportError:
        raise
    else:
        import os
        idledir = os.path.dirname(os.path.abspath(PyShell.__file__))
        if idledir != os.getcwd():
            # We're not in the IDLE directory, help the subprocess find run.py
            pypath = os.environ.get('PYTHONPATH', '')
            if pypath:
                os.environ['PYTHONPATH'] = pypath + ':' + idledir
            else:
                os.environ['PYTHONPATH'] = idledir
        PyShell.main()
else:
    idlelib.PyShell.main()
Ln: 1 Col: 0
```

(b) L'éditeur d'IDLE

FIGURE 9.4 – IDLE.

Notion de développement agile



Le développement « agile » est un style de développement logiciel itératif, davantage centré sur les personnes que sur les méthodes, et qui met l'accent sur la satisfaction du client à travers l'intégration continue d'un logiciel entièrement fonctionnel.

Le « Manifeste Agile » est un texte rédigé en 2001 par 17 experts reconnus pour leurs apports respectifs au développement d'applications informatiques sous la forme de plusieurs méthodes dont les plus connues sont l'*eXtreme Programming* et *Scrum*.

Cette philosophie couvre l'ensemble du cycle de développement du projet, mais nous nous limiterons ici aux problèmes de la documentation et des tests.

10.1 Les tests

Dès lors qu'un programme dépasse le stade du petit script, le problème des erreurs et donc des tests se pose inévitablement.

Définition

i 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.

Méthode

Dans la philosophie du « développement agile », les tests sont écrits **en même temps** que le code, voire juste avant. On parle de DDT, Développement Dirigé par les Tests (ou TDD, *Test Driven Development*).

10.1.1 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.

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`

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.

On peut citer les projets :

- **Mechanize** : fournit des objets Python sans IHM qui bouchonnent un navigateur Web ;
- **Selenium** : tests fonctionnels pour les applications Web dans un véritable navigateur ;
- **guitest** : teste les applications GTK ;
- **FunkLoad** offre un système de benchmark et de reporting étendu.

10.1.2 Le développement dirigé par les tests

Le module Python `unittest` fournit l'outil `PyUnit`, outils 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`.

`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 unitaire¹ :

```
"""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
```

```
"""Module de test du module de calculs."""

# imports
import unittest
import os
import sys
```

1. Cf. [B6] p. 132

```

dirName = os.path.dirname(__file__)
if dirName == '':
    dirName = '.'
dirName = os.path.realpath(dirName)
upDir = os.path.split(dirName)[0]
if upDir not in sys.path:
    sys.path.append(upDir)

from calculs import moyenne, division

# classes
class CalculTest(unittest.TestCase):

    def test_moyenne(self):
        self.assertEqual(moyenne(1, 2, 3), 2)
        self.assertEqual(moyenne(2, 4, 6), 4)

    def test_division(self):
        self.assertEqual(division(10, 5), 2)
        self.assertRaises(ZeroDivisionError, division, 10, 0)

def test_suite():
    tests = [unittest.makeSuite(CalculTest)]
    return unittest.TestSuite(tests)

if __name__ == '__main__':
    unittest.main()

```

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.

10.2 La documentation

Durant la vie d'un projet, on distingue plusieurs types de documentation :

- les documents de spécification (*upstream documentation*) ;
- les documents techniques attachés au code (*mainstream documentation*) ;
- les manuels d'utilisation et autres documents de haut niveau (*downstream documentation*).

Les documents *mainstream* évoluent au rythme du code et doivent 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 :

le **reStructuredText** (ou **reST**) : un format enrichi ;

les **doctests** : compatibles avec le format **reST**. Ils permettent de combiner les textes applicatifs avec les tests.


```
La section 2 est un peu vantarde : la section 1 est *très bien*.
```

```
-----
```

```
Un tableau de trois images au format "png"
```

```
::::::::::::::::::::::::::::::::::::::::::::
```

```
=====  =====  =====
```

```
Image 1 Image 2 Image 3
```

```
=====  =====  =====
```

```
|shamr| |elysT| |helen|
```

```
=====  =====  =====
```

```
.. |shamr| image:: shamr.png
```

```
.. |elysT| image:: elysT.png
```

```
.. |helen| image:: helen.png
```

L'utilitaire `rst2html`, appliqué à ce fichier, produit le fichier de même nom mais avec l'extension `.html` (cf. Fig. 10.1).

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.

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*.

Test d'un tableau d'images en format "png"




Image 1	Image 2	Image 3
		

FIGURE 10.1 – exemple de sortie au format HTML.

10.2.2 Le module doctest

Le principe du *literate programming* de Donald KNUTH, le créateur de L^AT_EX, a été repris en Python pour documenter les API via les chaînes de documentation. Des programmes comme Epydoc peuvent alors les extraire des modules pour composer une documentation.

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 trois exemples.

Pour chacun, nous donnerons d'une part le source muni de sa chaîne de documentation, et dans lequel le module standard `doctest` permet d'extraire puis de lancer ces sessions pour vérifier qu'elles fonctionnent, et d'autre part une capture d'écran de l'exécution.

Premier exemple : doctest1.py

```

"""Module d'essai de doctest."""

# import
import doctest

# fonctions
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 a rien vu !")
    input()
    print("{:-^40}".format(" Mode détaillé "))
    doctest.testmod(verbose=True)

```

L'exécution de ce fichier est illustré Fig. 10.2.

Deuxième exemple : doctest2.py

```

"""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 :

```

```

----- Mode silencieux -----
Si tout va bien, on 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.

```

FIGURE 10.2 – Exécution du script doctest1.py.

```

>>> texte = "Est-ce que tu as regardé la télé hier soir ? Il y avait un thème
    sur les ramasseurs d'escargots en Laponie, ils en bavent..."
>>> accentEtrange(texte)
Est-ceu queu tu as rReugarRrdé la télé hieurRr soirRr ? Il y avait un théma
    surRr leus rRramasseurRrs d'euscarRrgots eun Laponieu, ils eun baveunt...

Cette technique permet d'internationaliser 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 est illustré Fig. 10.3.

Troisième exemple : exemple.py

```

"""
This is the "example" module.

The example module supplies one function, factorial(). For example,

>>> factorial(5)
120
"""

def factorial(n):

```



```

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 rReugarRrdé 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.

```

FIGURE 10.3 – Exécution du script doctest2.py.

```

"""Return the factorial of n, an exact integer >= 0.

If the result is small enough to fit in an int, return an int.
Else return a long.

>>> [factorial(n) for n in range(6)]
[1, 1, 2, 6, 24, 120]
>>> factorial(30)
26525285981219105863630848000000
>>> factorial(-1)
Traceback (most recent call last):
...
ValueError: n must be >= 0

Factorials of floats are OK, but the float must be an exact integer:
>>> factorial(30.1)
Traceback (most recent call last):
...
ValueError: n must be exact integer
>>> factorial(30.0)
26525285981219105863630848000000

It must also not be ridiculously large:
>>> factorial(1e100)
Traceback (most recent call last):
...
OverflowError: n too large
"""

import math
if not n >= 0:
    raise ValueError("n must be >= 0")
if math.floor(n) != n:
    raise ValueError("n must be exact integer")
if n+1 == n: # catch a value like 1e300
    raise OverflowError("n too large")
result = 1
factor = 2
while factor <= n:

```

```

        result *= factor
        factor += 1
    return result

def _test():
    import doctest
    doctest.testmod(verbose=True)

if __name__ == "__main__":
    _test()
    print("OK")

```

L'exécution de ce fichier est illustré Fig. 10.4.

10.2.3 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 : doctest2.py

Source du module¹.

```

Le module ‘‘accent’’
=====

Test de la fonction ‘‘accentEtrange’’
-----

Ce module fournit une fonction ‘‘accentEtrange’’ qui permet d’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 thème
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 thème
surRr
leus rRramasseurRrs 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.

"""Module d’essai de doctest2."""

import doctest
doctest.testfile("doctest2.txt", verbose=True)

```

1. Cf. [B6] p. 132

```

^ Trying:
    factorial(5)
Expecting:
    120
ok
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
Trying:
    factorial(30)
Expecting:
    265252859812191058636308480000000
ok
Trying:
    factorial(-1)
Expecting:
    Traceback (most recent call last):
      ...
    ValueError: n must be >= 0
ok
Trying:
    factorial(30.1)
Expecting:
    Traceback (most recent call last):
      ...
    ValueError: n must be exact integer
ok
Trying:
    factorial(30.0)
Expecting:
    265252859812191058636308480000000
ok

    factorial(1e100)
Expecting:
    Traceback (most recent call last):
      ...
    OverflowError: n too large
ok
1 items had no tests:
    __main__.test
2 items passed all tests:
   1 tests in __main__
   6 tests in __main__.factorial
7 tests in 3 items.
7 passed and 0 failed.
Test passed.
OK

```

FIGURE 10.4 – Exécution du script `exemple.py`.

La documentation de ce fichier est illustré Fig. 10.5.

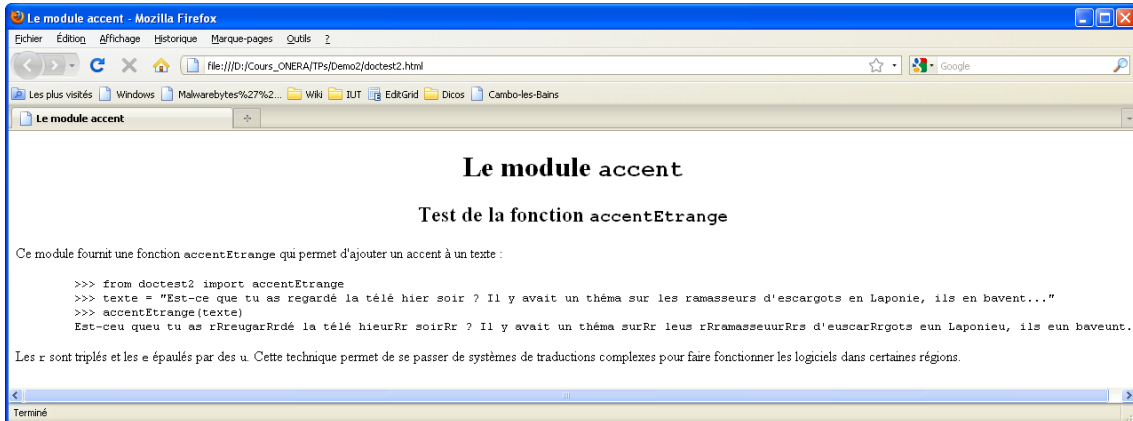


FIGURE 10.5 – Documentation du script doctest2.py.

Exemple de calcul des factorielles

Source du module¹.

```
The ‘‘example’’ module
=====

Using ‘‘factorial’’
-----

This is an example text file in reStructuredText format. First import
‘‘factorial’’ from the ‘‘example’’ module:

>>> from example import factorial

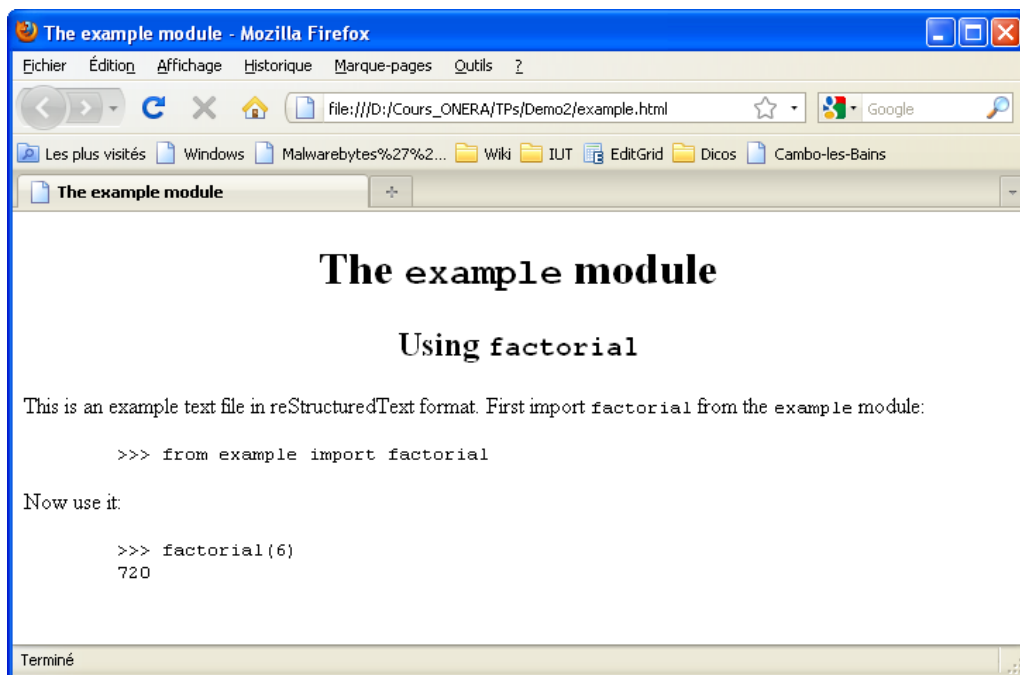
Now use it:

>>> factorial(6)
720

import doctest
doctest.testfile("example.txt", verbose=True)
```

La documentation de ce fichier est illustré Fig. 10.6.

1. Cf. [B6] p. 132

FIGURE 10.6 – Documentation du script `example.py`.

Interlude

Le Zen de Python ¹

Préfère :

*la beauté à la laideur,
l'explicite à l'implicite,
le simple au complexe
et 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 nommage ! Sacrée bonne idée ! Faisons plus de trucs
comme ça.*

1. Tim PETERS (PEP n° 20), traduction Cécile TREVIAN et Bob CORDEAU.
Retour chap.1 historique, p. 4

Le Graal de Python¹ !



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'entretuer 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!)

1. Cf. [B5] p. 132

LANCELOT:

[rabat l'écran de son laptop]

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!

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.

En effet, la grande diversité des langues humaines et de leur représentation nécessite un codage adapté.

La première idée est de construire une table qui associe les symboles à représenter à un nombre (généralement codé sur un octet) :

Symbole \longleftrightarrow Nombre

La plus célèbre est la table ASCII¹ (cf. Fig.B.1), codée sur 7 bits (soit 128 codes), mais bien d'autres tables ont été créées (EBCDIC, ISO-8852-1...).

n°	char	n°	char	n°	char	n°	char
32		56	B	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	Δ

FIGURE B.1 – Table ASCII.

La table Unicode

En Python 3, les chaînes de caractères (le type `str()`) sont des chaînes Unicode, norme dans laquelle les identifiants numériques de leurs caractères sont uniques et universels.

1. American Standard Code for Information Interchange

Comme il s'agit de différencier plusieurs centaines de milliers de caractères (on compte environ 6000 langues dans le monde) ils ne pourront évidemment pas être encodés 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 (cf. Fig.B.2).

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0100	Ā	ā	Ă	ă	Ą	ą	Ć	ć	Ĉ	ĉ	Č	č	Ď	ď		
0110	Đ	đ	Ě	ě	É	é	Ě	ě	Ĝ	ĝ	Ğ	ğ				
0120	Ġ	ġ	Ģ	ģ	Ĥ	ĥ	Ħ	ħ	Ĩ	ĩ	Ĳ	ĳ	Ĵ	ĵ	Ķ	ķ
0130	Ĳ	ĳ	Ĵ	ĵ	Ķ	ķ	ĸ	Ĺ	ĺ	Ł	ł	Ł	ł	Ł	ł	Ł
0140	Ł	ł	Ń	ń	Ņ	ņ	Ň	ň	Ŋ	ŋ	Ō	ō	Ŏ	ö		
0150	Ŏ	ö	Œ	œ	Ŕ	ŕ	Ŗ	ŗ	Ř	ř	Ś	ś	Ŝ	ŝ	Ş	ş
0160	Ŝ	ŝ	Ţ	ţ	Ť	ť	Ŧ	ŧ	Ū	ū	Ū	ū	Ŭ	ŭ	Ŭ	ŭ
0170	Ū	ū	Ŭ	ŭ	Ŷ	ŷ	Ÿ	ÿ	Ž	ž	Ž	ž	Ž	ž	Ů	ů

FIGURE B.2 – Extrait de la table Unicode.

Encodage

Après avoir collecté tous les symboles et y avoir associé un nombre, il reste à leur trouver une *représentation binaire*.

Pour l'ASCII un seul octet suffisait mais pour représenter les millions de possibilités de l'Unicode, plusieurs octets par caractère sont nécessaires.

Comme la plupart des textes n'utilisent que la table ASCII ce qui correspond justement à la partie basse de la table Unicode, l'encodage le plus économique est l'**UTF-8**.

Pour les codes 0 à 127, l'UTF-8 utilise l'octet de la table ASCII. Pour les caractères spéciaux (codes 128 à 2047), l'UTF-8 utilise 2 octets. Pour les caractères spéciaux encore moins courants (codes 2048 à 65535), l'UTF-8 utilise 3 octets, et ainsi de suite¹.

Symbole \longleftrightarrow **Nombre** \longleftrightarrow **Bits**

Exemple de l'encodage UTF-8 du caractère Unicode « é » :

é \longleftrightarrow **233** \longleftrightarrow C3 A9

1. Retour chap. 2 identifiants, p. 10

Applications pratiques

Les entrées/sorties

Il est important de pouvoir préciser sous quelle forme exacte les données sont attendues par nos programmes, que ces données soient fournies par l'intermédiaire de frappes au clavier ou par importation depuis une source quelconque. De même, nous devons pouvoir choisir le format des données que nous exportons vers n'importe quel dispositif périphérique, qu'il s'agisse d'une imprimante, d'un disque dur, d'un écran...

Pour toutes ces entrées ou sorties de chaînes de caractères, nous devons donc toujours considérer qu'il s'agit *concrètement* de séquences d'octets, et utiliser divers mécanismes pour convertir ces séquences d'octets en chaînes de caractères, ou vice-versa.

Python met désormais à votre disposition le nouveau type de données `bytes`, spécifiquement conçu pour traiter les séquences (ou chaînes) d'octets. Les caractères peuvent bien entendu être *encodés* en octets, et les octets *décodés* en caractères (en particulier avec l'encodage UTF-8 que nous venons de voir) :

```
>>> chaine = "Une çédille\n"
>>> of = open("test.txt", "w") # une chaine de caractères, type str()
>>> of.write("chaine")
12
>>> of.close()
>>> of = open("test.txt", "rb") # une chaine d'octets, type bytes()
>>> octets = of.read()
>>> type(chaine)
<class 'str'>
>>> print(chaine)
Une çédille

>>> len(chaine)
12
>>> type(octets)
<class 'bytes'>
>>> print(octets)
b'Un \xe7\xe9dille\r\b'
>>> len(octets)
13
```

Cas des scripts Python

Puisque les scripts Python que l'on produit avec un éditeur sont eux-mêmes des textes, ils risquent d'être encodés suivant différentes normes. Afin que Python puisse les interpréter correctement, il est utile de préciser le jeu de caractères (obligatoirement en 1^{re} ou 2^e ligne des sources).

Il faut au préalable connaître le jeu de caractères utilisé par votre système (cf. Fig.B.3).

```
import sys
print(sys.stdout.encoding) # cp1252 (Windows XP/SciTE)
```

Alors on le précise dans les scripts :

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

Ou bien (ce qui est le défaut pour la version 3 de Python) :

```
# -*- coding: UTF-8 -*-
```

Windows-1252 (CP1252)																
	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VI	FF	CR	SO	SI
1x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2x	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL
8x	€		,	f	„	…	†	‡	^	%o	Š	<	Œ		Ž	
9x		'	'	"	"	•	-	—	~	™	š	>	œ		ž	ÿ
Ax	NBSP	ı	¢	£	¤	¥	¦	§	¨	©	ª	«	¬		®	¯
Bx	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Dx	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
Ex	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
Fx	ø	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

FIGURE B.3 – Le jeu de caractères cp1252.

Pour éviter les erreurs Unicode

Pour éviter les erreurs Unicode, la technique consiste à indiquer l'encodage au moment des entrées-sorties. Par exemple :

```
>>> chaine = "Une çédille\n"
>>> of = open("test.txt", "w", encoding="Latin-1")
>>> of.write("chaine")
12
>>> of.close()
>>> of = open("test.txt", "r", encoding="Latin-1")
>>> ch_lue = of.read()
>>> of.close()
>>> ch_lue
'Une çédille\n'
```

Les fonctions logiques

La logique de Boole

Au 19^e siècle, le logicien et mathématicien George BOOLE restructura complètement la logique en un système formel. Aujourd'hui, l'algèbre de BOOLE trouve de nombreuses applications en informatique et dans la conception des circuits électroniques.

C'est une logique à deux valeurs. Soit $\mathcal{B} = \{0, 1\}$, l'ensemble de définition, sur lequel on définit les opérateurs NON, ET et OU.

Les valeurs des variables sont parfois notées **False** et **True**. Les opérateurs sont parfois notés respectivement \bar{a} , $a.b$ et $a + b$.

Les tables de vérité

Table de vérité des opérateurs booléens de base :

Opérateur unaire NON

a	$NON(a)$
0	1
1	0

Opérateurs binaires OU et ET

a	b	$a OU b$	$a ET b$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

Table de vérité des opérateurs composés¹ :

Opérateurs ou exclusif, équivalence et implication

a	b	$a XOR b$	$a \iff b$	$a \implies b$
0	0	0	1	1
0	1	1	0	1
1	0	1	0	0
1	1	0	1	1

1. Retour chap. 2 expressions booléennes, p. 12

Les bases arithmétiques

Définition

Définition

i 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.

Conversion

Définition

i Les changements de base : un nombre dans une base n donnée s'écrit sous la forme d'addition des puissances successives de cette base¹.

Exemples

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

1. Retour chap.2 type int, p. 11

Les fonctions de hachage

Principe

C'est une application f qui prend en entrée des fichiers de longueur différente, les condense, et fournit en sortie une séquence binaire de longueur fixe (cf. Fig.E.1) :

$$f : \{0, 1\}^* \rightarrow \{0, 1\}^k$$

où f applique l'ensemble des séquences binaires : $\{0, 1\}^* = \{\emptyset, 0, 1, 10, 11, 100, 101, \dots\}$ sur l'ensemble des séquences binaires de k bits : $\{0, 1\}^k = \{0\dots00, 0\dots01, 0\dots10, \dots, 1\dots11\}$.

Cette application doit permettre d'*identifier* les fichiers en entrée : $f(x) = f(y)$ sera vrai si et seulement si $x = y$.

Par contre, à cause de la longueur finie de k , on ne peut pas *reconstituer* les fichiers : il existe deux valeurs x et y différentes (fichiers distincts), telles que $f(x) = f(y)$ (même séquence binaire). On parle alors de *collision*.

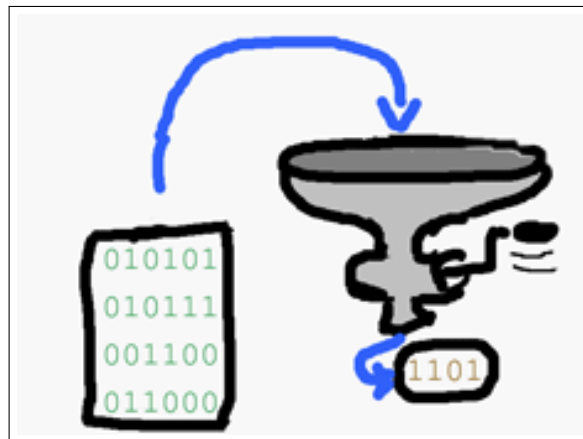


FIGURE E.1 – Le principe du hachage

Réalisation pratique

Il faut résoudre deux problèmes :

- diminuer le nombre de collisions en choisissant une séquence binaire assez longue. Avec, par exemple, $k = 512$, on obtient 2^{512} soit environ 10^{154} cellules disponibles.
- gérer les collisions restantes par un algorithmes approprié.

Application aux dictionnaires

On dispose d'un espace mémoire S pour stocker m données. On peut accéder à chaque donnée par une notation associative où l'information entre crochet s'appelle la *clé* : $S[0], S[1], \dots, S[m - 1]$.

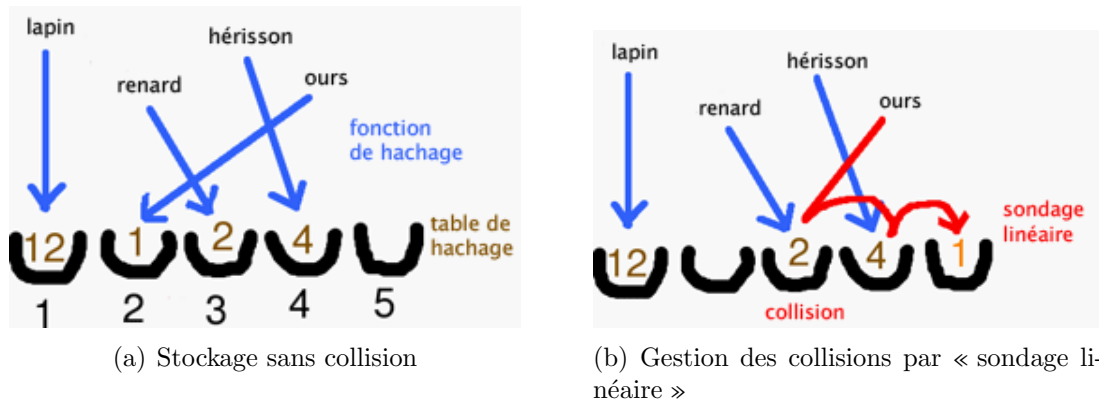


FIGURE E.2 – Hachage des clés d’un dictionnaire.

Par exemple avec des clés de caractères, on utilisera une fonction de hachage ainsi définie¹ :

$$f : \{a, b, c, \dots, z\}^* \rightarrow \{0, 1, 2, \dots, m - 1\}$$

Soit à stocker les informations suivantes dans un dictionnaire (cf. Fig.E.2) :

lapin	12
renard	2
hérisson	4
ours	1

1. Retour chap.4 type dict, p. 34

Exercices corrigés

Énoncés des exercices

Remarque

☞ Les exercices suivants sont fournis à titre d'exemples et de modèles. Ils sont soit simples, soit moins simples (notés ▷ dans la marge) soit plus difficiles (notés ▷▷).

1. Écrire un programme qui, à partir de la saisie d'un rayon et d'une hauteur, calcule le volume d'un cône droit.
2. Une boucle while : entrez un prix HT (entrez 0 pour terminer) et affichez sa valeur TTC.
3. Une autre boucle while : calculez la somme d'une suite de nombres positifs ou nuls. Comptez combien il y avait de données et combien étaient supérieures à 100.
Un nombre inférieur ou égal à 0 indique la fin de la suite.
4. L'utilisateur donne un entier positif n et le programme affiche PAIR s'il est divisible par 2 et IMPAIR sinon.
5. L'utilisateur donne un entier positif et le programme annonce combien de fois de suite cet entier est divisible par 2.

- ▷ 6. L'utilisateur donne un entier supérieur à 1 et le programme affiche, s'il y en a, tous ses diviseurs propres *sans répétition* ainsi que leur nombre. S'il n'y en a pas, il indique qu'il est premier. Par exemple :

```
Entrez un entier strictement positif : 12
Diviseurs propres sans répétition de 12 : 2 3 4 6 (soit 4 diviseurs propres
)
```

```
Entrez un entier strictement positif : 13 Diviseurs propres sans
répétition de 13 : aucun ! Il est premier
```

7. Écrire un programme qui estime la valeur de la constante mathématique e en utilisant la formule :

$$e = \sum_{i=0}^n \frac{1}{i!}$$

Pour cela, définissez la fonction factorielle et, dans votre programme principal, saisissez l'ordre n et affichez l'approximation correspondante de e .

8. Un gardien de phare va aux toilettes cinq fois par jour or les WC sont au rez-de-chaussée. . .

Écrire une procédure (donc sans retour) `hauteurparcourue` qui reçoit deux paramètres le nombre de marches du phare et la hauteur de chaque marche (en cm), et qui affiche :

Pour x marches de y cm, il parcourt $z.zz$ m par semaine.

On n'oubliera pas :

- qu'une semaine comporte 7 jours ;
- qu'une fois en bas, le gardien doit remonter ;
- que le résultat est à exprimer en m.

9. Un permis de chasse à points remplace désormais le permis de chasse traditionnel. Chaque chasseur possède au départ un capital de 100 points. S'il tue une poule il perd 1 point, 3 points pour 1 chien, 5 points pour une vache et 10 points pour un ami. Le permis coûte 200 euros.

Écrire une fonction `amende` qui reçoit le nombre de victimes du chasseur et qui renvoie la somme due.

Utilisez cette fonction dans un programme principal qui saisit le nombre de victimes et qui affiche la somme que le chasseur doit déboursier.

10. Je suis ligoté sur les rails en gare d'Arras. Écrire un programme qui affiche un tableau me permettant de connaître l'heure à laquelle je serai déchiqueté par le train parti de la gare du Nord à 9h (il y a 170 km entre la gare du Nord et Arras).

Le tableau prédira les différentes heures possibles pour toutes les vitesses de 100 km/h à 300 km/h, par pas de 10 km/h, les résultats étant arrondis à la minute inférieure.

- Écrire une procédure `tchacatchac` qui reçoit la vitesse du train et qui affiche l'heure du drame ;
- écrire le programme principal qui affiche le tableau demandé.

11. Un programme principal saisit une chaîne d'ADN valide et une séquence d'ADN valide (« valide » signifie qu'elles ne sont pas vides et sont formées exclusivement d'une combinaison arbitraire de "a", "t", "g" ou "c").

Écrire une fonction `valide` qui renvoie vrai si la saisie est valide, faux sinon.

Écrire une fonction `saisie` qui effectue une saisie valide et renvoie la valeur saisie sous forme d'une chaîne de caractères.

Écrire une fonction `proportion` qui reçoit deux arguments, la chaîne et la séquence et qui retourne la proportion de séquence dans la chaîne (c'est-à-dire son nombre d'occurrences).

Le programme principal appelle la fonction `saisie` pour la chaîne et pour la séquence et affiche le résultat.

Exemple d'affichage :

Il y a 13.33 % de "ca" dans votre chaîne.

12. Il s'agit d'écrire, d'une part, un programme principal, et d'autre part, une fonction utilisée dans le programme principal.

La fonction `listAleaInt(n, a, b)` retourne une liste de n entiers aléatoires dans $[a .. b]$ en utilisant la fonction `randint(a, b)` du module `random`.

Dans le programme principal :

- construire la liste en appelant la fonction `listAleaInt()` ;
- calculer l'indice de la case qui contient le minimum ;
- échangez le premier élément du tableau avec son minimum.

13. Comme précédemment, il s'agit d'écrire, d'une part, un programme principal, et d'autre part, une fonction utilisée dans le programme principal.

La fonction `listAleaFloat(n)` retourne une liste de n flottants aléatoires en utilisant la fonction `random()` du module `random`.

Dans le programme principal :

- Saisir un entier n dans l'intervalle : $[2 \dots 100]$;
- construire la liste en appelant la fonction `listAleaFloat()` ;
- afficher l'*amplitude* du tableau (écart entre sa plus grande et sa plus petite valeur) ;
- afficher la *moyenne* du tableau.

14. Fonction renvoyant plusieurs valeurs sous forme d'un *tuple*.

Écrire une fonction `minMaxMoy` qui reçoit une liste d'entiers et qui renvoie le minimum, le maximum et la moyenne de cette liste. Le programme principal appellera cette fonction avec la liste : $[10, 18, 14, 20, 12, 16]$.

15. Saisir un entier entre 1 et 3999 (pourquoi cette limitation ?). L'afficher en nombre romain.

▷ 16. Améliorer le script précédent en utilisant la fonction `zip()`.

▷ 17. Un tableau contient n entiers ($2 < n < 100$) aléatoires tous compris entre 0 et 500. Vérifier qu'ils sont tous différents.

18. L'utilisateur donne un entier n entre 2 et 12, le programme donne le nombre de façons de faire n en lançant deux dés.

19. Même problème que le précédent mais avec n entre 3 et 18 et trois dés.

▷▷ 20. Généralisation des deux questions précédentes. L'utilisateur saisit deux entrées, d'une part le nombre de dés, nb (que l'on limitera pratiquement à 10), et d'autre part la somme, s , comprise entre nb et $6 \cdot nb$. Le programme calcule et affiche le nombre de façons de faire s avec les nb dés.

▷▷ 21. Même problème que le précédent mais codé récursivement.

22. Nombres parfaits et nombres chanceux.

- On appelle *nombre premier* tout entier naturel supérieur à 1 qui possède exactement deux diviseurs, lui-même et l'unité.
- On appelle *diviseur propre* de n , un diviseur quelconque de n , n exclu.
- Un entier naturel est dit *parfait* s'il est égal à la somme de tous ses diviseurs propres.
- Les nombres a tels que : $(a + n + n^2)$ est premier pour tout n tel que $0 \leq n < (a - 1)$, sont appelés *nombres chanceux*.

Écrire un module (`parfait_chanceux_m.py`) définissant quatre fonctions : `somDiv`, `estParfait`, `estPremier`, `estChanceux` et un auto-test :

- la fonction `somDiv` retourne la somme des diviseurs propres de son argument ;
- les trois autres fonctions vérifient la propriété donnée par leur définition et retourne un booléen. Plus précisément, si par exemple la fonction `estPremier` vérifie que son argument est premier, elle retourne `True`, sinon elle retourne `False`.

La partie de test doit comporter quatre appels à la fonction `verif` permettant de tester `somDiv(12)`, `estParfait(6)`, `estPremier(31)` et `estChanceux(11)`.

Puis écrire le programme principal (`parfait_chanceux.py`) qui comporte :

- l'initialisation de deux listes : `parfaits` et `chanceux` ;
- une boucle de parcours de l'intervalle `[2, 1000]` incluant les tests nécessaires pour remplir ces listes ;
- enfin l'affichage de ces listes.

Solutions des exercices

```
# -*- coding: Latin-1 -*-
"""Volume d'un cône."""

# imports
from math import pi

# programme principal -----
rayon = float(input("Rayon du cône (m) : "))
hauteur = float(input("Hauteur du cône (m) : "))

volume = (pi*rayon*rayon*hauteur)/3.0
print("Volume du cône =", volume, "m3")
```

```
# -*- coding: Latin-1 -*-
"""Calcul d'un prix TTC."""

# programme principal -----
prixHT = float(input("Prix HT (0 pour terminer)? "))
while prixHT > 0:
    print("Prix TTC : {:.2f}\n".format(prixHT * 1.196))
    prixHT = float(input("Prix HT (0 pour terminer)? "))

print("Au revoir !")
```

```
# -*- coding: Latin-1 -*-
"""Somme d'entiers et nombre d'entiers supérieur à 100."""

# programme principal -----
somme, nombreTotal, nombreGrands = 0, 0, 0

x = int(input("x (0 pour terminer) ? "))
while x > 0:
    somme += x
    nombreTotal += 1
    if x > 100:
        nombreGrands += 1
    x = int(input("x (0 pour terminer) ? "))

print("\nSomme :", somme)
print(nombreTotal, "valeur(s) en tout, dont", nombreGrands, "supérieure(s) à 100")
```

```
# -*- coding: Latin-1 -*-
"""Parité."""
```

```
# programme principal -----
n = int(input("Entrez un entier strictement positif : "))
while n < 1:
    n = int(input("Entrez un entier STRICTEMENT POSITIF, s.v.p. : "))

if n%2 == 0:
    print(n, "est pair.")
else:
    print(n, "est impair.")
```

```
# -*- coding: Latin-1 -*-
"""Nombre de fois qu'un entier est divisible par 2."""

# programme principal -----
n = int(input("Entrez un entier strictement positif : "))
while n < 1:
    n = int(input("Entrez un entier STRICTEMENT POSITIF, s.v.p. : "))
save = n

cpt = 0
while n%2 == 0:
    n /= 2
    cpt += 1

print(save, "est", cpt, "fois divisible par 2.")
```

```
# -*- coding: Latin-1 -*-
"""Diviseurs propres d'un entier."""

# programme principal -----
n = int(input("Entrez un entier strictement positif : "))
while n < 1:
    n = int(input("Entrez un entier STRICTEMENT POSITIF, s.v.p. : "))

i = 2      # plus petit diviseur possible de n
cpt = 0    # initialise le compteur de divisions
p = n/2    # calculé une fois dans la boucle

print("Diviseurs propres sans répétition de ", n, ":", end=' ')
while i <= p:
    if n%i == 0:
        cpt += 1
        print(i, end=' ')
    i += 1

if not cpt:
    print("aucun ! Il est premier.")
else:
    print("(soit", cpt, "diviseurs propres)")
```

```
# -*- coding: Latin-1 -*-
"""Approximation de 'e'."""

# fonction
def fact(n):
    r = 1
    for i in range(1, n+1):
        r *= i
```

```

    return r

# programme principal -----
n = int(input("n ? "))
exp = 0.0
for i in range(n):
    exp = exp + 1.0/fact(i)

print("Approximation de 'e' : {:.3f}".format(exp))

```

```

# -*- coding: Latin-1 -*-
"""Gardien de phare."""

# fonction
def hauteurParcourue(nb, h):
    print("Pour {:d} marches de {:d} cm, il parcourt {:.2f} m par semaine.".
          format(nb, h, nb*h*2*5*7/100.0))

# programme principal -----
nbMarches = int(input("Combien de marches ? "))
hauteurMarche = int(input("Hauteur d'une marche (cm) ? "))

hauteurParcourue(nbMarches, hauteurMarche)

```

```

# -*- coding: Latin-1 -*-
"""Permis de chasse."""

# fonction
def permisSup(p, c, v, a):
    pointsPerdus = p + 3*c + 5*v + 10*a
    nbrePermis = pointsPerdus/100.0
    return 200*nbrePermis

# programme principal -----
poules = int(input("Combien de poules ? "))
chiens = int(input("Combien de chiens ? "))
vaches = int(input("Combien de vaches ? "))
amis = int(input("Combien d'amis ? "))

payer = permisSup(poules, chiens, vaches, amis)

print("\nA payer :", end=' ')
if payer == 0:
    print("rien à payer")
else:
    print(payer, "euros")

```

```

# -*- coding: Latin-1 -*-
"""Histoire de train."""

# fonction
def tchacatchac(v):
    """Affiche l'heure du drame."""
    heure = 9 + int(170/v)
    minute = (60 * 170 / v) % 60
    print("A", v, "km/h, je me fais déchiqueter à", heure, "h", minute, "min.")

# programme principal -----

```

```

i = 100
while i <= 300:
    tchacatchac(i)
    i += 10

```

```

# -*- coding: Latin-1 -*-
"""Proportion d'une séquence dans une chaîne d'ADN."""

# fonctions
def valide(seq):
    """Retourne vrai si la séquence est valide, faux sinon."""
    ret = any(seq)
    for c in seq:
        ret = ret and c in "atgc"
    return ret

def proportion(a, s):
    """Retourne la proportion de la séquence <s> dans la chaîne <a>."""
    return 100*a.count(s)/len(a)

def saisie(ch):
    s = input("{:s} : ".format(ch))
    while not valide(s):
        print("'{:s}' ne peut contenir que les chaînons 'a', 't', 'g' et 'c' et"
              " ne doit pas être vide".format(ch))
        s = input("{:s} : ".format(ch))
    return s

# programme principal -----
adn = saisie("chaîne")
seq = saisie("séquence")

print('Il y a {:.2f} % de "{:s}" dans votre chaîne.'
      .format(proportion(adn, seq), seq))

```

```

# -*- coding: Latin-1 -*-
"""Echanges."""

# imports
from random import seed, randint

# fonction
def listAleaInt(n, a, b):
    """Retourne une liste de <n> entiers aléatoires dans [<a> .. <b>]."""
    return [randint(a, b) for i in range(n)]

# programme principal -----
seed() # initialise le générateur de nombres aléatoires
t = listAleaInt(100, 2, 125) # construction de la liste

# calcul de l'indice du minimum de la liste
iMin = t.index(min(t))

print("Avant échange :")
print("\tt[0] =", t[0], "\tt[iMin] =", t[iMin])
t[0], t[iMin] = t[iMin], t[0] # échange
print("Après échange :")
print("\tt[0] =", t[0], "\tt[iMin] =", t[iMin])

```



```

# -*- coding: Latin-1 -*-
"""Amplitude et moyenne d'une liste de flottants."""

# imports
from random import seed, random

# fonctions
def listAleaFloat(n):
    "Retourne une liste de <n> flottants aléatoires"
    return [random() for i in range(n)]

# programme principal -----
n = int(input("Entrez un entier [2 .. 100] : "))
while not(n >= 2 and n <= 100):
    n = int(input("Entrez un entier [2 .. 100], s.v.p. : "))

seed() # initialise le générateur de nombres aléatoires
t = listAleaFloat(n) # construction de la liste

print("Amplitude : {:.2f}".format(max(t) - min(t)))
print("Moyenne : {:.2f}".format(sum(t)/n))

```

```

# -*- coding: Latin-1 -*-
"""Min, max et moyenne d'une liste d'entiers."""

# fonction
def minMaxMoy(liste):
    """Renvoie le min, le max et la moyenne de la liste."""
    min, max, som = liste[0], liste[0], float(liste[0])
    for i in liste[1:]:
        if i < min:
            min = i
        if i > max:
            max = i
        som += i
    return (min, max, som/len(liste))

# programme principal -----
lp = [10, 18, 14, 20, 12, 16]

print("liste =", lp)
l = minMaxMoy(lp)
print("min : {0[0]}, max : {0[1]}, moy : {0[2]}".format(l))

```

```

# -*- coding: Latin-1 -*-
"""Nombres romains (version 1)."""

# programme principal -----
n = int(input('Entrez un entier [1 .. 4000[ : ')
while not(n >= 1 and n < 4000):
    n = int(input('Entrez un entier [1 .. 4000[, s.v.p. : ')

s = "" # Chaîne résultante

while n >= 1000:
    s += "M"
    n -= 1000

if n >= 900:

```

```

s += "CM"
n -= 900

if n >= 500:
    s += "D"
    n -= 500

if n >= 400:
    s += "CD"
    n -= 400

while n >= 100:
    s += "C"
    n -= 100

if n >= 90:
    s += "XC"
    n -= 90

if n >= 50:
    s += "L"
    n -= 50

if n >= 40:
    s += "XL"
    n -= 40

while n >= 10:
    s += "X"
    n -= 10

if n >= 9:
    s += "IX"
    n -= 9

if n >= 5:
    s += "V"
    n -= 5

if n >= 4:
    s += "IV"
    n -= 4

while n >= 1:
    s += "I"
    n -= 1

print("En romain :", s)

```

```

# -*- coding: Latin-1 -*-
"""Nombres romains (version 2)."""

# globales
code = zip(
    [1000,900 ,500,400 ,100,90 ,50 ,40 ,10 ,9 ,5 ,4 ,1],
    ["M" , "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"]
)

```

```

# fonction
def decToRoman(num):
    res = []
    for d, r in code:
        while num >= d:
            res.append(r)
            num -= d
    return ''.join(res)

# programme principal -----
for i in range(1, 4000):
    print(i, decToRoman(i))

```

```

# -*- coding: latin-1 -*-
"""Liste d'entiers différents."""

# imports
from random import seed, randint

# fonction
def listAleaInt(n, a, b):
    """Retourne une liste de <n> entiers aléatoires entre <a> et <b>."""
    return [randint(a, b) for i in range(n)]

# programme principal -----
n = int(input("Entrez un entier [1 .. 100] : "))
while not(n >= 1 and n <= 100):
    n = int(input("Entrez un entier [1 .. 100], s.v.p. : "))

# construction de la liste
seed() # initialise le générateur de nombres aléatoires
t = listAleaInt(n, 0, 500)

# Sont-ils différents ?
tousDiff = True
i = 0
while tousDiff and i < (n-1):
    j = i + 1
    while tousDiff and j < n:
        if t[i] == t[j]:
            tousDiff = False
        else:
            j += 1
    i += 1

print("\n", t, end=' ')
if tousDiff:
    print(": tous les éléments sont distincts.")
else:
    print(": au moins une valeur est répétée.")

```

```

# -*- coding: Latin-1 -*-
"""Jeu de dés (1)."""

# programme principal -----
n = int(input("Entrez un entier [2 .. 12] : "))
while not(n >= 2 and n <= 12):
    n = int(input("Entrez un entier [2 .. 12], s.v.p. : "))

```

```

s = 0
for i in range(1, 7):
    for j in range(1, 7):
        if i+j == n:
            s += 1

print("Il y a {:d} façon(s) de faire {:d} avec deux dés.".format(s, n))

```

```

# -*- coding: Latin-1 -*-
"""Jeu de dés (2)."""

# programme principal -----
n = int(input("Entrez un entier [3 .. 18] : "))
while not(n >= 3 and n <= 18):
    n = int(input("Entrez un entier [3 .. 18], s.v.p. : "))

s = 0
for i in range(1, 7):
    for j in range(1, 7):
        for k in range(1, 7):
            if i+j+k == n:
                s += 1

print("Il y a {:d} façon(s) de faire {:d} avec trois dés.".format(s, n))

```

```

# -*- coding: Latin-1 -*-
"""Jeu de dés (3)."""

# globale
MAX = 8

# programme principal -----
nbd = int(input("Nombre de dés [2 .. {:d}] : ".format(MAX)))
while not(nbd >= 2 and nbd <= MAX):
    nbd = int(input("Nombre de dés [2 .. {:d}], s.v.p. : ".format(MAX)))

s = int(input("Entrez un entier [{:d} .. {:d}] : ".format(nbd, 6*nbd)))
while not(s >= nbd and s <= 6*nbd):
    s = int(input("Entrez un entier [{:d} .. {:d}], s.v.p. : ".format(nbd, 6*nbd)
    ))

if s == nbd or s == 6*nbd:
    cpt = 1 # 1 seule solution
else:
    I = [1]*nbd # initialise une liste de <nbd> dés
    cpt, j = 0, 0
    while j < nbd:
        som = sum([I[k] for k in range(nbd)])

        if som == s:
            cpt += 1 # compteur de bonnes solutions
        if som == 6*nbd:
            break

        j = 0
        if I[j] < 6:
            I[j] += 1
        else:
            while I[j] == 6:

```

```

        I[j] = 1
        j += 1
        I[j] += 1

print("Il y a {:d} façons de faire {:d} avec {:d} dés.".format(cpt, s, nbd))

```

```

# -*- coding: Latin-1 -*-
"""Jeu de dés (récuratif)."""

# globale
MAX = 8

# fonction
def calcul(d, n):
    """Calcul récuratif du nombre de façons de faire <n> avec <d> dés."""
    resultat, debut = 0, 1
    if (d == 1) or (n == d) or (n == 6*d): # conditions terminales
        return 1
    else:
        # sinon appels récuratifs
        if n > 6*(d-1): # optimisation importante
            debut = n - 6*(d-1)

        for i in range(debut, 7):
            if n == i:
                break
            resultat += calcul(d-1, n-i)
    return resultat

# programme principal -----
d = int(input("Nombre de dés [2 .. {:d}] : ".format(MAX)))
while not(d >= 2 and d <= MAX):
    d = int(input("Nombre de dés [2 .. {:d}], s.v.p. : ".format(MAX)))

n = int(input("Entrez un entier [{:d} .. {:d}] : ".format(d, 6*d)))
while not(n >= d and n <= 6*d):
    n = int(input("Entrez un entier [{:d} .. {:d}], s.v.p. : ".format(d, 6*d)))

print("Il y a {:d} façon(s) de faire {:d} avec {:d} dés.".format(calcul(d, n), n,
d))

```

```

# -*- coding: Latin-1 -*-
"""module d'exemple de polymorphisme."""

# classes
class Rectangle:
    """classe des rectangles."""
    def __init__(self, longueur=30, largeur=15):
        """Constructeur avec valeurs par défaut."""
        self.lon = longueur
        self.lar = largeur
        self.nom = "rectangle"

    def surface(self):
        """Calcule la surface d'un rectangle."""
        return self.lon*self.lar

    def __str__(self):
        """Affichage des caractéristiques d'un rectangle."""
        return "\nLe '{:s}' de côtés {:s} et {:s} a une surface de {:s}"

```

```
        .format(self.nom, self.lon, self.lar, self.surface())

class Carre(Rectangle):
    """classe des carrés (hérite de Rectangle)."""
    def __init__(self, cote=10):
        """Constructeur avec valeur par défaut"""
        Rectangle.__init__(self, cote, cote)
        self.nom = "carré" # surcharge d'attribut d'instance

# Auto-test -----
if __name__ == '__main__':
    r = Rectangle(12, 8)
    print r

    c = Carre()
    print c
```



Licence associée à ce document

Vous êtes libres :

- de reproduire, distribuer et communiquer cette création au public
- de modifier cette création

Selon les conditions suivantes :

- **Paternité** – Vous devez citer le nom de l’auteur original de la manière indiquée par l’auteur de l’œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d’une manière qui suggérerait qu’ils vous soutiennent ou approuvent votre utilisation de l’œuvre).
- **Pas d’Utilisation Commerciale** – Vous n’avez pas le droit d’utiliser cette création à des fins commerciales.
- **Partage des Conditions Initiales à l’Identique** – Si vous modifiez, transformez ou adaptez cette création, vous n’avez le droit de distribuer la création qui en résulte que sous un contrat identique à celui-ci.



FIGURE G.1 – Paternité – Pas d’Utilisation Commerciale – Partage Des Conditions Initiales À L’Identique

Creative Commons n’est pas un cabinet d’avocats et ne fournit pas de services de conseil juridique. La distribution de la présente version de ce contrat ne crée aucune relation juridique entre les parties au contrat présenté ci-après et Creative Commons. Creative Commons fournit cette offre de contrat-type en l’état, à seule fin d’information. Creative Commons ne saurait être tenu responsable des éventuels préjudices résultant du contenu ou de l’utilisation de ce contrat.

Contract

L’Œuvre (telle que définie ci-dessous) est mise à disposition selon les termes du présent contrat appelé Contrat Public Creative Commons (dénommé ici « CPCC » ou « Contrat »). L’Œuvre est protégée par le droit de la propriété littéraire et artistique (droit d’auteur, droits voisins, droits des producteurs de bases de données) ou toute autre loi applicable. Toute utilisation de l’Œuvre autrement qu’explicitement autorisée selon ce Contrat ou le droit applicable est interdite.

L’exercice sur l’Œuvre de tout droit proposé par le présent contrat vaut acceptation de celui-ci. Selon les termes et les obligations du présent contrat, la partie Offrante propose

à la partie Acceptante l'exercice de certains droits présentés ci-après, et l'Acceptant en approuve les termes et conditions d'utilisation.

1. Définitions

- (a) « Œuvre » : œuvre de l'esprit protégeable par le droit de la propriété littéraire et artistique ou toute loi applicable et qui est mise à disposition selon les termes du présent Contrat.
- (b) « Œuvre dite Collective » : une œuvre dans laquelle l'Œuvre, dans sa forme intégrale et non modifiée, est assemblée en un ensemble collectif avec d'autres contributions qui constituent en elles-mêmes des œuvres séparées et indépendantes. Constituent notamment des œuvres dites Collectives les publications périodiques, les anthologies ou les encyclopédies. Aux termes de la présente autorisation, une œuvre qui constitue une œuvre dite Collective ne sera pas considérée comme une œuvre dite Dérivée (telle que définie ci-après).
- (c) « Œuvre dite Dérivée » : une œuvre créée soit à partir de l'Œuvre seule, soit à partir de l'Œuvre et d'autres œuvres préexistantes. Constituent notamment des œuvres dites Dérivées les traductions, les arrangements musicaux, les adaptations théâtrales, littéraires ou cinématographiques, les enregistrements sonores, les reproductions par un art ou un procédé quelconque, les résumés, ou toute autre forme sous laquelle l'Œuvre puisse être remaniée, modifiée, transformée ou adaptée, à l'exception d'une œuvre qui constitue une œuvre dite Collective. Une œuvre dite Collective ne sera pas considérée comme une œuvre dite Dérivée aux termes du présent Contrat. Dans le cas où l'Œuvre serait une composition musicale ou un enregistrement sonore, la synchronisation de l'œuvre avec une image animée sera considérée comme une œuvre dite Dérivée pour les propos de ce Contrat.
- (d) « Auteur original » : la ou les personnes physiques qui ont créé l'Œuvre.
- (e) « Offrant » : la ou les personne(s) physique(s) ou morale(s) qui proposent la mise à disposition de l'Œuvre selon les termes du présent Contrat.
- (f) « Acceptant » : la personne physique ou morale qui accepte le présent contrat et exerce des droits sans en avoir violé les termes au préalable ou qui a reçu l'autorisation expresse de l'Offrant d'exercer des droits dans le cadre du présent contrat malgré une précédente violation de ce contrat.
- (g) « Options du Contrat » : les attributs génériques du Contrat tels qu'ils ont été choisis par l'Offrant et indiqués dans le titre de ce Contrat : Paternité - Pas d'Utilisation Commerciale - Partage Des Conditions Initiales A l'Identique.

2. **Exceptions aux droits exclusifs.** Aucune disposition de ce contrat n'a pour intention de réduire, limiter ou restreindre les prérogatives issues des exceptions aux droits, de l'épuisement des droits ou d'autres limitations aux droits exclusifs des ayants droit selon le droit de la propriété littéraire et artistique ou les autres lois applicables.

3. **Autorisation.** Soumis aux termes et conditions définis dans cette autorisation, et ceci pendant toute la durée de protection de l'Œuvre par le droit de la propriété littéraire et artistique ou le droit applicable, l'Offrant accorde à l'Acceptant l'autorisation mondiale d'exercer à titre gratuit et non exclusif les droits suivants :

- (a) reproduire l'Œuvre, incorporer l'Œuvre dans une ou plusieurs œuvres dites Collectives et reproduire l'Œuvre telle qu'incorporée dans lesdites œuvres dites Collectives ;
- (b) créer et reproduire des œuvres dites Dérivées ;

- (c) distribuer des exemplaires ou enregistrements, présenter, représenter ou communiquer l'Œuvre au public par tout procédé technique, y compris incorporée dans des œuvres Collectives ;
- (d) distribuer des exemplaires ou phonogrammes, présenter, représenter ou communiquer au public des œuvres dites Dérivées par tout procédé technique ;
- (e) lorsque l'Œuvre est une base de données, extraire et réutiliser des parties substantielles de l'Œuvre.

Les droits mentionnés ci-dessus peuvent être exercés sur tous les supports, médias, procédés techniques et formats. Les droits ci-dessus incluent le droit d'effectuer les modifications nécessaires techniquement à l'exercice des droits dans d'autres formats et procédés techniques. L'exercice de tous les droits qui ne sont pas expressément autorisés par l'Offrant ou dont il n'aurait pas la gestion demeure réservé, notamment les mécanismes de gestion collective obligatoire applicables décrits à l'article 4(e).

4. **Restrictions.** L'autorisation accordée par l'article 3 est expressément assujettie et limitée par le respect des restrictions suivantes :

- (a) L'Acceptant peut reproduire, distribuer, représenter ou communiquer au public l'Œuvre y compris par voie numérique uniquement selon les termes de ce Contrat. L'Acceptant doit inclure une copie ou l'adresse Internet (Identifiant Uniforme de Ressource) du présent Contrat à toute reproduction ou enregistrement de l'Œuvre que l'Acceptant distribue, représente ou communique au public y compris par voie numérique. L'Acceptant ne peut pas offrir ou imposer de conditions d'utilisation de l'Œuvre qui altèrent ou restreignent les termes du présent Contrat ou l'exercice des droits qui y sont accordés au bénéficiaire. L'Acceptant ne peut pas céder de droits sur l'Œuvre. L'Acceptant doit conserver intactes toutes les informations qui renvoient à ce Contrat et à l'exonération de responsabilité. L'Acceptant ne peut pas reproduire, distribuer, représenter ou communiquer au public l'Œuvre, y compris par voie numérique, en utilisant une mesure technique de contrôle d'accès ou de contrôle d'utilisation qui serait contradictoire avec les termes de cet Accord contractuel. Les mentions ci-dessus s'appliquent à l'Œuvre telle qu'incorporée dans une œuvre dite Collective, mais, en dehors de l'Œuvre en elle-même, ne soumettent pas l'œuvre dite Collective, aux termes du présent Contrat. Si l'Acceptant crée une œuvre dite Collective, à la demande de tout Offrant, il devra, dans la mesure du possible, retirer de l'œuvre dite Collective toute référence au dit Offrant, comme demandé. Si l'Acceptant crée une œuvre dite Collective, à la demande de tout Auteur, il devra, dans la mesure du possible, retirer de l'œuvre dite Collective toute référence au dit Auteur, comme demandé. Si l'Acceptant crée une œuvre dite Dérivée, à la demande de tout Offrant, il devra, dans la mesure du possible, retirer de l'œuvre dite Dérivée toute référence au dit Offrant, comme demandé. Si l'Acceptant crée une œuvre dite Dérivée, à la demande de tout Auteur, il devra, dans la mesure du possible, retirer de l'œuvre dite Dérivée toute référence au dit Auteur, comme demandé.
- (b) L'Acceptant peut reproduire, distribuer, représenter ou communiquer au public une œuvre dite Dérivée y compris par voie numérique uniquement sous les termes de ce Contrat, ou d'une version ultérieure de ce Contrat comprenant les mêmes Options du Contrat que le présent Contrat, ou un Contrat Creative Commons iCommons comprenant les mêmes Options du Contrat que le présent Contrat (par exemple Paternité - Pas d'Utilisation Commerciale - Partage Des Conditions Initiales A l'Identique 2.0 Japon). L'Acceptant doit inclure une copie ou l'adresse Internet (Identifiant Uniforme de Ressource) du

présent Contrat, ou d'un autre Contrat tel que décrit à la phrase précédente, à toute reproduction ou enregistrement de l'œuvre dite Dérivée que l'Acceptant distribue, représente ou communique au public y compris par voie numérique. L'Acceptant ne peut pas offrir ou imposer de conditions d'utilisation sur l'œuvre dite Dérivée qui altèrent ou restreignent les termes du présent Contrat ou l'exercice des droits qui y sont accordés au bénéficiaire, et doit conserver intactes toutes les informations qui renvoient à ce Contrat et à l'avertissement sur les garanties. L'Acceptant ne peut pas reproduire, distribuer, représenter ou communiquer au public y compris par voie numérique l'œuvre dite Dérivée en utilisant une mesure technique de contrôle d'accès ou de contrôle d'utilisation qui serait contradictoire avec les termes de cet Accord contractuel. Les mentions cidessus s'appliquent à l'œuvre dite Dérivée telle qu'incorporée dans une œuvre dite Collective, mais, en dehors de l'œuvre dite Dérivée en elle-même, ne soumettent pas l'œuvre Collective, aux termes du présent Contrat.

- (c) L'Acceptant ne peut exercer aucun des droits conférés par l'article 3 avec l'intention ou l'objectif d'obtenir un profit commercial ou une compensation financière personnelle. L'échange de l'Œuvre avec d'autres œuvres protégées par le droit de la propriété littéraire et artistique par le partage électronique de fichiers, ou par tout autre moyen, n'est pas considéré comme un échange avec l'intention ou l'objectif d'un profit commercial ou d'une compensation financière personnelle, dans la mesure où aucun paiement ou compensation financière n'intervient en relation avec l'échange d'œuvres protégées.
- (d) Si l'Acceptant reproduit, distribue, représente ou communique au public, y compris par voie numérique, l'Œuvre ou toute œuvre dite Dérivée ou toute œuvre dite Collective, il doit conserver intactes toutes les informations sur le régime des droits et en attribuer la paternité à l'Auteur Original, de manière raisonnable au regard au médium ou au moyen utilisé. Il doit communiquer le nom de l'Auteur Original ou son éventuel pseudonyme s'il est indiqué ; le titre de l'Œuvre Originale s'il est indiqué ; dans la mesure du possible, l'adresse Internet ou Identifiant Uniforme de Ressource (URI), s'il existe, spécifié par l'Offrant comme associé à l'Œuvre, à moins que cette adresse ne renvoie pas aux informations légales (paternité et conditions d'utilisation de l'Œuvre). Dans le cas d'une œuvre dite Dérivée, il doit indiquer les éléments identifiant l'utilisation l'Œuvre dans l'œuvre dite Dérivée par exemple « Traduction anglaise de l'Œuvre par l'Auteur Original » ou « Scénario basé sur l'Œuvre par l'Auteur Original ». Ces obligations d'attribution de paternité doivent être exécutées de manière raisonnable. Cependant, dans le cas d'une œuvre dite Dérivée ou d'une œuvre dite Collective, ces informations doivent, au minimum, apparaître à la place et de manière aussi visible que celles à laquelle apparaissent les informations de même nature.
- (e) Dans le cas où une utilisation de l'Œuvre serait soumise à un régime légal de gestion collective obligatoire, l'Offrant se réserve le droit exclusif de collecter ces redevances par l'intermédiaire de la société de perception et de répartition des droits compétente. Sont notamment concernés la radiodiffusion et la communication dans un lieu public de phonogrammes publiés à des fins de commerce, certains cas de retransmission par câble et satellite, la copie privée d'œuvres fixées sur phonogrammes ou vidéogrammes, la reproduction par reprographie.

5. Garantie et exonération de responsabilité

- (a) En mettant l'Œuvre à la disposition du public selon les termes de ce Contrat,

l'Offrant déclare de bonne foi qu'à sa connaissance et dans les limites d'une enquête raisonnable :

- i. L'Offrant a obtenu tous les droits sur l'Œuvre nécessaires pour pouvoir autoriser l'exercice des droits accordés par le présent Contrat, et permettre la jouissance paisible et l'exercice licite de ces droits, ceci sans que l'Acceptant n'ait aucune obligation de verser de rémunération ou tout autre paiement ou droits, dans la limite des mécanismes de gestion collective obligatoire applicables décrits à l'article 4(e) ;
- ii. L'Œuvre n'est constitutive ni d'une violation des droits de tiers, notamment du droit de la propriété littéraire et artistique, du droit des marques, du droit de l'information, du droit civil ou de tout autre droit, ni de diffamation, de violation de la vie privée ou de tout autre préjudice délictuel à l'égard de toute tierce partie.

- (b) À l'exception des situations expressément mentionnées dans le présent Contrat ou dans un autre accord écrit, ou exigées par la loi applicable, l'Œuvre est mise à disposition en l'état sans garantie d'aucune sorte, qu'elle soit expresse ou tacite, y compris à l'égard du contenu ou de l'exactitude de l'Œuvre.

6. **Limitation de responsabilité.** À l'exception des garanties d'ordre public imposées par la loi applicable et des réparations imposées par le régime de la responsabilité vis-à-vis d'un tiers en raison de la violation des garanties prévues par l'article 5 du présent contrat, l'Offrant ne sera en aucun cas tenu responsable vis-à-vis de l'Acceptant, sur la base d'aucune théorie légale ni en raison d'aucun préjudice direct, indirect, matériel ou moral, résultant de l'exécution du présent Contrat ou de l'utilisation de l'Œuvre, y compris dans l'hypothèse où l'Offrant avait connaissance de la possible existence d'un tel préjudice.

7. Résiliation

- (a) Tout manquement aux termes du contrat par l'Acceptant entraîne la résiliation automatique du Contrat et la fin des droits qui en découlent. Cependant, le contrat conserve ses effets envers les personnes physiques ou morales qui ont reçu de la part de l'Acceptant, en exécution du présent contrat, la mise à disposition d'œuvres dites Dérivées, ou d'œuvres dites Collectives, ceci tant qu'elles respectent pleinement leurs obligations. Les sections 1, 2, 5, 6 et 7 du contrat continuent à s'appliquer après la résiliation de celui-ci.
- (b) Dans les limites indiquées ci-dessus, le présent Contrat s'applique pendant toute la durée de protection de l'Œuvre selon le droit applicable. Néanmoins, l'Offrant se réserve à tout moment le droit d'exploiter l'Œuvre sous des conditions contractuelles différentes, ou d'en cesser la diffusion ; cependant, le recours à cette option ne doit pas conduire à retirer les effets du présent Contrat (ou de tout contrat qui a été ou doit être accordé selon les termes de ce Contrat), et ce Contrat continuera à s'appliquer dans tous ses effets jusqu'à ce que sa résiliation intervienne dans les conditions décrites ci-dessus.

8. Divers

- (a) À chaque reproduction ou communication au public par voie numérique de l'Œuvre ou d'une œuvre dite Collective par l'Acceptant, l'Offrant propose au bénéficiaire une offre de mise à disposition de l'Œuvre dans des termes et conditions identiques à ceux accordés à la partie Acceptante dans le présent Contrat.
- (b) À chaque reproduction ou communication au public par voie numérique d'une œuvre dite Dérivée par l'Acceptant, l'Offrant propose au bénéficiaire une offre

de mise à disposition du bénéficiaire de l'Œuvre originale dans des termes et conditions identiques à ceux accordés à la partie Acceptante dans le présent Contrat.

- (c) La nullité ou l'inapplicabilité d'une quelconque disposition de ce Contrat au regard de la loi applicable n'affecte pas celle des autres dispositions qui resteront pleinement valides et applicables. Sans action additionnelle par les parties à cet accord, lesdites dispositions devront être interprétées dans la mesure minimum nécessaire à leur validité et leur applicabilité.
- (d) Aucune limite, renonciation ou modification des termes ou dispositions du présent Contrat ne pourra être acceptée sans le consentement écrit et signé de la partie compétente.
- (e) Ce Contrat constitue le seul accord entre les parties à propos de l'Œuvre mise ici à disposition. Il n'existe aucun élément annexe, accord supplémentaire ou mandat portant sur cette œuvre en dehors des éléments mentionnés ici. L'Offrant ne sera tenu par aucune disposition supplémentaire qui pourrait apparaître dans une quelconque communication en provenance de l'Acceptant. Ce Contrat ne peut être modifié sans l'accord mutuel écrit de l'Offrant et de l'Acceptant.
- (f) Le droit applicable est le droit français.

Creative Commons n'est pas partie à ce Contrat et n'offre aucune forme de garantie relative à l'Œuvre. Creative Commons décline toute responsabilité à l'égard de l'Acceptant ou de toute autre partie, quel que soit le fondement légal de cette responsabilité et quel que soit le préjudice subi, direct, indirect, matériel ou moral, qui surviendrait en rapport avec le présent Contrat. Cependant, si Creative Commons s'est expressément identifié comme Offrant pour mettre une œuvre à disposition selon les termes de ce Contrat, Creative Commons jouira de tous les droits et obligations d'un Offrant. À l'exception des fins limitées à informer le public que l'Œuvre est mise à disposition sous CPCC, aucune des parties n'utilisera la marque « Creative Commons » ou toute autre indication ou logo afférent sans le consentement préalable écrit de Creative Commons. Toute utilisation autorisée devra être effectuée en conformité avec les lignes directrices de Creative Commons à jour au moment de l'utilisation, telles qu'elles sont disponibles sur son site Internet ou sur simple demande.

Si vous souhaitez en savoir davantage concernant Creative commons, veuillez consulter le site web : creativecommons.org/licenses/by-nc-sa/2.0/fr/

Ressources

Webographie

- Les sites généraux :
www.python.org
pypi.python.org/pypi
www.pythonxy.com/download.php
rgruet.free.fr
- Les EDI spécialisés :
www.wingware.com/downloads/wingide-101
eric-ide.python-projects.org/eric4-download.html
www.eclipse.org/downloads/
www.scintilla.org/SciTEDownload.html
- Les outils :
sourceforge.net/projects/gnuplot/files/
- Le lien des liens :
www.limsi.fr/Individu/pointal/python.html

Bibliographie

- [B1] SWINNEN, Gérard, *Apprendre à programmer avec Python 3*, Eyrolles, 2010.
- [B2] SUMMERFIELD, Mark, *Programming in Python 3*, Addison-Wesley, 2^e édition, 2009.
- [B3] MARTELLI, Alex, *Python en concentré*, O'Reilly, 2004.
- [B4] LUTZ, Mark et BAILLY, Yves, *Python précis et concis*, O'Reilly, 2^e édition, 2005.
- [B5] ZIADÉ, Tarek, *Programmation Python. Conception et optimisation*, Eyrolles, 2^e édition, 2009.
- [B6] ZIADÉ, Tarek, *Python : Petit guide à l'usage du développeur agile*, Dunod, 2007.
- [B7] ZIADÉ, Tarek, *Expert Python Programming*, Packt Publishing, 2008.
- [B8] YOUNKER, Jeff, *Foundations of Agile Python Development*, Apress, 2008.
- [B9] CHEVALIER Céline et collectif, *L^AT_EX pour l' impatient*, H & K, 3^e édition, 2009.
- [B10] CARELLA, David, *Règles typographiques et normes. Mise en pratique avec L^AT_EX*, Vuibert, 2006.

Colophon

Ce texte a été composé grâce au logiciel \LaTeX sous l'éditeur \TeXnicCenter . La distribution utilisée est \MiKTeX , version 2.8.

Le corps du texte est en police vectorielle **aeguill**, les exemples de code en police **Typewriter** et le programme **Inkscape** a permis de dessiner les illustrations.

