

PYTHON

INFO PRO

L'ESSENTIEL	TYPE D'OUVRAGE
SE FORMER	RETOURS D'EXPÉRIENCE

Tarek Ziadé

PYTHON

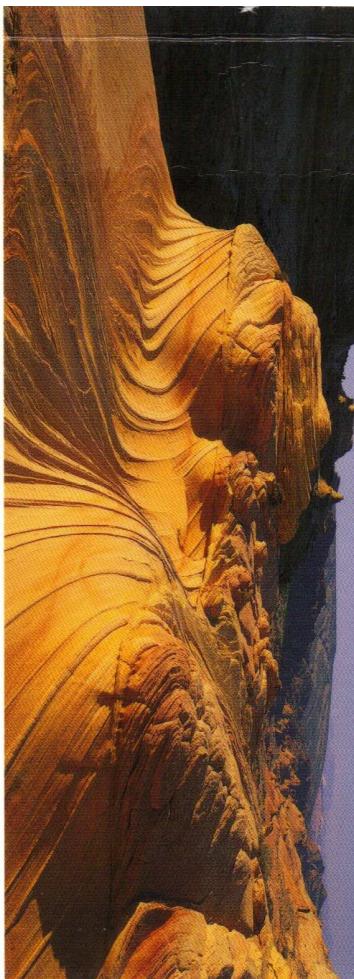
Petit guide à l'usage du développeur agile

Cet ouvrage s'adresse aux développeurs qui souhaitent découvrir et maîtriser Python, et aux chefs de projet délégués de faire évoluer leur équipe autour d'un projet Python.

Il rassemble toute une série de conseils concrets pour mener à bien des projets d'envergure et construire des applications de qualité avec le langage Python. Il explique les principes du **développement agile**, une philosophie de programmation basée sur la facilité de modifier une application, de la faire grandir sans en perdre le contrôle, et d'être réactif en toutes circonstances.

Le développeur et le chef de projet trouveront dans ce livre des informations utiles sur :

- l'environnement de développement (système d'exploitation, éditeur de code...);
- les principes d'architecture logicielle à respecter;
- la philosophie et la syntaxe de Python ;
- les bonnes pratiques qui permettent de tirer le meilleur parti de Python ;
- le développement dirigé par les tests ;
- le développement dirigé par la documentation ;
- l'art et la manière de mettre en place un environnement de projet agile.



Tarek Ziadé



DUNOD

6639587
ISBN 978-2-10-050883-9

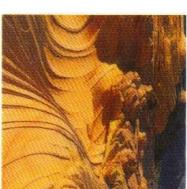
www.dunod.com



TAREK ZIADÉ
est architecte spécialisé dans l'Open Source. Il a contribué au projet Python majeurs comme Zope ou CPS et a lancé l'Association francophone Python dont il est l'actuel président.
Il est également auteur d'articles et intervient dans des conférences (EuroPython, PyCon, etc.)
tarek@ziade.org
<http://programmation-python.org>

MANAGEMENT DES
SYSTÈMES D'INFORMATIQUE
APPLICATIONS
EXPLOITATION
ET ADMINISTRATION
MÉTIERS
RÉSEAUX
& TÉLÉCOMS

INTÉGRATION
ÉTUDES, DÉVELOPPEMENT
RÉSEAUX
MANAGEMENT DES
APPLICATIONS
EXPLOITATION
ET ADMINISTRATION
MÉTIERS
RÉSEAUX
& TÉLÉCOMS



PYTHON

Table des matières

Petit guide à l'usage du développeur agile

Avant-propos 4

1 Introduction 7

2 Environnement de développement 9

2.1 Philosophie	9
2.2 Choisir un système d'exploitation	10
2.2.1 Système cible du projet	10
2.2.2 Importance des outils système	11
2.2.3 Culture d'entreprise	13
2.3 Choisir un éditeur de code Python	13
2.4 Ce qu'il faut retenir	15

3 Architecture logicielle 17

3.1 Philosophie	17
3.2 Les cinq lois de l'architecture logicielle	18
3.2.1 Modularité	18
3.2.2 Extensibilité	22
3.2.3 Simplicité	23
3.2.4 Paramétrabilité	23
3.2.5 Standardisation	25
3.3 Ce qu'il faut retenir	25

4 Développement 27

Tarek Ziadé
*Architecte spécialisé dans l'Open Source
et président de l'Association francophone Python*

4	4.1.1	L'art de programmer	27
4.2	Syntaxe de Python	32	
4.2.1	L'indentation	32	
4.2.2	Les variables	33	
4.2.3	Le tout objet	33	
4.2.4	Les types et objets notables	34	
4.2.5	Les structures conditionnelles	41	
4.2.6	pass, continue et break	42	
4.2.7	Les fonctions, classes, modules et paquets	43	
4.3	Ce qu'il faut retenir	49	
5	Bonnes pratiques	51	
5.1	Philosophie	51	
5.2	Du bon usage de l'objet	52	
5.2.1	Fonctionnement et intérêt des objets	52	
5.2.2	Design patterns communs	66	
5.3	Du bon usage de Python 2.5	74	
5.3.1	Code patterns communs	74	
5.3.2	Bonnes pratiques	91	
5.4	Outils d'assurance qualité	95	
5.4.1	PyLint	96	
5.4.2	Cheesecake	97	
5.5	Ce qu'il faut retenir	98	
6	Développement dirigé par les tests	99	
6.1	Philosophie	99	
6.1.1	Qu'est-ce qu'un test ?	100	
6.1.2	Quand intervient les tests dans le cycle de développement ?	100	
6.1.3	Quels sont les avantages des tests ?	101	
6.1.4	Tests unitaires	102	
6.1.5	Tests fonctionnels	103	
6.2	unittest et doctest	104	
6.2.1	unittest	104	
6.2.2	doctest	108	
6.3	Ce qu'il faut retenir	108	
7	Développement dirigé par la documentation	109	
7.1	Philosophie	109	
7.2	Les sept lois de l'écriture technique	110	
7.2.1	Écriture en deux étapes	111	
7.2.2	Style simple	113	
7.2.3	Lectorat cible	114	
7.2.4	Information ciblée	114	
7.2.5	Exemples réalistes	115	
7.2.6	Approche <i>light but sufficient</i>	116	
7.2.7	Documents structurés	117	
7.3	Outils d'écriture	117	
7.3.1	Le reStructuredText	118	
7.3.2	Les doctests	130	
7.4	Organisation de la documentation	136	
7.4.1	Documentation unitaire	136	
7.4.2	Documentation d'un paquet	140	
7.4.3	Documentation d'un projet	143	
7.5	Développement dirigé par la documentation	151	
7.5.1	Décomposition du cahier des charges	152	
7.5.2	Phase cyclique de construction	154	
7.6	Team writing	157	
7.6.1	Document reviewing	157	
7.6.2	Pair writing	158	
7.7	Ce qu'il faut retenir	158	
8	Gestion de projet	159	
8.1	Philosophie	159	
8.1.1	Visibilité continue	160	
8.1.2	Propriété collective	161	
8.1.3	Revue de code	162	
8.1.4	Gestion centralisée du code	164	
8.1.5	Cycle de développement itératif	166	
8.2	Mise en place technique	169	
8.2.1	Gestionnaire de source	169	
8.2.2	Gestionnaire de tickets	178	
8.2.3	Listes de diffusion	182	
8.2.4	Intégration continue	183	
8.2.5	Packaging et distribution	185	
8.3	Ce qu'il faut retenir	185	

Index	186
Bibliographie	188

Avant-propos

Mais qu'en est-il de nous autres, pauvres développeurs, qui subissons des vagues incessantes d'innovations, de modifications techniques ? Psoriasis, eczéma, jaunisse. Nous sommes devenus spécialistes des maladies liées au stress. Alors, plutôt que de consommer des amphétamines et de la cocaine, les développeurs qui survivent à la pression des projets sont ceux qui ont cultivé une certaine *agilité*².

Ce livre traite de l'agilité appliquée à Python. Il est le fruit de dix ans de pratique³.

La cible est :

- le développeur qui maîtrise un langage de programmation, ou qui commence à maîtriser Python ;
- le chef de projet qui cherche un ouvrage pour monter ou faire évoluer son équipe autour d'un projet Python.

¹Les collègues essaieront de trouver un stratagème pour la pousser dans les escaliers, pour qu'enfin cessent les pleurs et gémissements.

²Ce terme reviendra souvent.

³Et d'eczéma.

Le format est volontairement réduit et riche en liens sur Internet pour que le contenu dépende le moins possible de techniques chanoines : les principes sont exposés, et les détails qui ne conviennent pas au format papier laissés au bon soin des sites référencés. Les scripts complets présentés, ainsi qu'une liste d'outils tiers, ont été déportés dans cette optique sur mon site internet.

Le résultat est là sous vos yeux, et a pu être concocté grâce à la complicité de certaines personnes que je dois absolument remercier :

- Mon fils, Milo.
- Les membres de l'**AFFPy**, l'Association Francophone Python. Des gens formidables. La liste exhaustive est sur le site.
- Mes relecteurs techniques, fines fleurs de la programmation Python. Par ordre alphabétique : Olivier Grisel, David Larlet, Victor Stinner.
- Fred Le Guluche, qui a créé les petits bonhommes de tête de chapitre.
- Les éditions Dunod.

Si certaines erreurs vous ont sauté aux yeux, si ce livre vous a plu, ou a provoqué chez vous des spasmes ou tout autre phénomène, ou si tout simplement vous avez envie d'échanger, contactez-moi au plus vite ou venez me rendre visite sur le site.

Bonne lecture!

Tarek Ziade

email : tarek@ziade.org
site : <http://programming-python.org>

Chapitre 1

Introduction



*Les ordinateurs ne remplaceront jamais les livres.
Vous ne pouvez pas vous mettre sur une pile de disquettes pour atteindre le haut de votre armoire.*

Sam Ewing

Le mot *agile* a été associé pour la première fois à la conception d'applications par un consortium de spécialistes des méthodes, qui ont fondé pour l'occasion l'*Agile Manifesto* en 2001. Il exprime un ensemble de principes appliqués au développement informatique, qui ont pour objectif de créer des logiciels centrés sur les besoins réels des utilisateurs, et capables d'évoluer rapidement. Ce manifeste se trouve ici : <http://agilemanifesto.org> et exprime quelques principes.

L'agilité s'exprime à travers plusieurs méthodologies, la plus connue étant l'*eXtreme Programming* (XP). Mais dans la communauté des développeurs, le terme a été étendu à une philosophie de programmation basée sur la facilité de modifier une application, de la faire grandir sans en perdre le contrôle, et d'être réactif en toutes circonstances. On parle de *développeur agile* ou encore de *gestion de projet agile* sans vraiment se focaliser sur des méthodologies précises.

L'agilité en informatique consiste donc à réussir à maîtriser l'évo-

lution fiérotique des technologies, le grossissement boulinique des applications, et la nervosité endémique des clients.

Ce livre est un concentré de l'expérience vécue à appliquer les principes de l'agilité avec le langage Python, pour mener à bien des projets d'envergures et concevoir du code de qualité. Il présente, dans un format concentré et synthétique, l'approche agile. Il égrène des conseils concrets et directement applicables, pour construire des applications de qualité.

La suite est organisée en sept chapitres :

- **Environnement de développement**, qui traite du système d'exploitation, de l'éditeur de code et de toutes les automatisations qui facilitent le travail de développeur ;
- **Architecture logicielle**, qui présente une synthèse des principes à respecter lorsqu'une application est conçue ;
- **Développement**, qui condense l'essentiel du langage ;
- **Bonnes pratiques**, qui fournit des conseils pour produire du code de qualité ;
- **Développement dirigé par les tests**, qui explique le principe des tests et les outils disponibles pour pratiquer ;
- **Développement dirigé par la documentation**, qui étend le principe des tests à la conception de la documentation ;
- **Gestion de projet**, qui explique pourquoi et comment mettre en place un environnement de projet agile.

Chapitre 2

Environnement de développement



*Les hommes sont plus importants que les outils.
Si vous en doutez, mettez un bon outil dans les mains d'un mauvais travailleur.*

John Bernet

2.1 Philosophie

La mise en place d'un environnement adapté au développement d'applications est une étape importante. Même si l'essentiel de l'agilité ne figure pas dans les outils mais dans le comportement des développeurs, il est nécessaire de minimiser autant que possible le chemin entre la pensée et la réalisation des tâches d'un projet.

Les deux axes de réflexion pour se construire un environnement cohérent sont :

- **le système d'exploitation**, qui fournit l'aire de jeu pour l'ensemble des activités ;
- **l'éditeur de code**, avec lequel le programmeur agile développe une relation à la limite du fétichisme.

Le temps passé à affûter cet environnement n'est jamais du temps perdu, car c'est le confort du travail et la productivité qui sont en jeu.

En général, le développeur prend conscience de cet aspect lorsqu'il réalise qu'il va passer un tiers de sa vie devant un petit écran. Ce chapitre fournit des réflexions pour chacun de ces axes, qui peuvent être adaptées en fonction des contraintes liées au contexte du projet.

Il est organisé en deux sections :

- Choisir un système d'exploitation.
- Choisir un éditeur de code.

2.2 Choisir un système d'exploitation

En théorie, les trois systèmes d'exploitation les plus répandus (Mac OS X, Windows, Linux) permettent tous de développer dans de bonnes conditions. La portabilité de Python est une réalité et la plupart des projets communautaires sont composés de développeurs qui travaillent sur des systèmes divers. Mais certaines plateformes peuvent s'avérer plus productives en fonction :

- du système cible du projet ;
- de l'importance des outils système ;
- de la culture d'entreprise.

2.2.1 Système cible du projet

L'argument principal pour le choix d'une plateforme de développement est dicté par le système cible du projet. Par exemple, les applications serveurs destinées à fournir un service accessible par des clients lourds ou des navigateurs sont très fréquemment déployées sous Linux. A fortiori, une application desktop développée pour une administration sera souvent destinée à l'unique plateforme déployée sur les postes clients.

Travailler sur le système cible du projet permet de bénéficier d'un *dogfooding* partiel : le code qui est conçu et testé l'est dans un contexte réaliste. Le dogfooding est une pratique agile qui consiste à utiliser les logiciels que l'on conçoit. L'exercice permet avant tout de déceler les problèmes d'ergonomie ou de performances en prenant le rôle d'un utilisateur, mais dans le développement il permet tout simplement de

valider en continu que la base de code se comporte comme prévu sur le système cible.

Philippe conçoit des applications desktop pour Windows. Il n'a pas vraiment le choix et l'accepte car ses clients sont tous sur cette plateforme. Mais Philippe, comme beaucoup de développeurs Python, est un geek dans l'âme, et ne jure que par Linux. Les applications qu'il conçoit utilisent Tkinter pour l'interface et lui permettent donc de développer depuis son système préféré. Les tests unitaires et fonctionnels lui assurent une très grande qualité et c'est un argument qu'il met en avant auprès de ses clients.

Son application a été déployée la semaine dernière. Son client lui envoie un mail pour lui demander d'ajouter des raccourcis claviers. Un test unitaire et quelques tests fonctionnels après, Philippe envoie une nouvelle version. Mais les raccourcis ne fonctionnent pas sur le poste du client et un échange de mails s'ensuit pour tenter de déterminer le problème, que Philippe ne reproduit pas.

Un déplacement est finalement organisé et Philippe découvre sur place que les signaux générés par les touches ne fonctionnent pas exactement de la même manière sous Windows.

Cette pratique ne remplace en rien les *tests fonctionnels* mais permet d'anticiper plus de problèmes.

L'environnement d'exécution réel d'un projet doit être connu et maîtrisé, et utilisé au moins pour les tests.

2.2.2 Importance des outils système

Lorsqu'il ne tape pas du code, un développeur :

- effectue des recherches dans le code et les fichiers du projet ;
- soumet ses modifications à un gestionnaire de version centralisé ;
- lance des tests ;

- manipule des applications tierces, comme les bases de données.

Ces activités nécessitent des outils, qui sont directement intégrés en commandes shell dans le système ou fournis par des applications dédiées.

Disposer de toute l'armada des commandes pour ces tâches annexes dans une seule et même interface augmente drastiquement la productivité, puisqu'il n'y a plus besoin de lancer plusieurs applications ni de s'adapter à chaque fois à un fonctionnement particulier. Même si c'est l'affaire de quelques secondes, ces changements de rythme coûtent cher dans une journée de travail. Cette concentration des fonctionnalités n'est donc possible que par le biais du shell ou d'un Environnement de Développement Intégré (EDI). Il faut donc opter pour une des deux solutions.

L'intérêt du shell est qu'il est extensible à souhait et universel. En effet, il permet de piloter et de manipuler toutes les applications utilisées par le développeur, par des commandes natives ou par le biais de scripts. Le défaut de cette solution est qu'elle nécessite de disposer d'un shell digne de ce nom, et Windows est de ce fait hors course. Linux et Mac OS X sont les seuls systèmes disposant nativement d'un shell qui offre toute les fonctionnalités nécessaires.

La séquence de commandes ci-dessous montre comment un développeur sous Linux va successivement :

- rechercher dans toute l'arborescence du projet le mot *toto* ;
- lancer les tests unitaires d'un paquet Python ;
- poster sur un repository Mercurial les modifications locales ;
- faire une requête SQL dans une base postgres.

```
dabox:~ tarek$ grep -ri "toto" .
./pylintrc:bad-names=foo,bar,baz,toto,tutu,tata
dabox:~ tarek$ python tests.py
.....
```

Ran 7 tests in 0.450s

OK

```
dabox:~ tarek$ hg ci -m "modification du test #4"
dabox:~ tarek$ psql -c "select * from user"
current_user
```

postgres
(1 row)

Cette manipulation, qui a demandé moins d'une minute au développeur, aurait été beaucoup plus longue s'il avait fallu lancer toutes les applications dédiées.

Un EDI a lui aussi pour objectif de réunir un maximum d'outils en une seule et même interface, et peut-être quant à lui portable sous Windows et compenser la faiblesse du shell de ce système.

2.2.3 Culture d'entreprise

Une autre influence majeure pour le choix de la plateforme est la culture d'entreprise. Si tous les outils annexes imposés par la direction, comme le suivi de l'activité ou encore l'annuaire centralisé, ne peuvent fonctionner que sur une plateforme donnée, il peut s'avérer fastidieux de passer continuellement d'un système à l'autre. L'influence peut parfois même porter sur une distribution spécifique d'un système.

Quoi qu'il en soit, l'essentiel pour un développeur est de se sentir à l'aise avec son système. Imposer ou s'imposer un environnement que l'on ne maîtrise pas peut s'avérer contre-productif ; mais prendre le temps de maîtriser les bons outils est payant à moyen et long terme.

Il y a deux étapes très dures à vivre dans le parcours d'un informaticien :

- se mettre à gnu/emacs ou Vim ;
- être obligé de passer à un autre outil.

2.3 Choisir un éditeur de code Python

Il y a deux écoles pour le choix d'un éditeur de code :

- un outil spécialisé dans l'édition, qui complète le shell ;
- un EDI, qui intègre le maximum de commandes et d'outils complémentaires.

TAB. 2.1 – Les éditeurs de code Python

Nom	CC	II	RD	SE	GS	FREE
Stani's WingIDE	X ²	X	X	X	X	X
PyDev (Eclipse)	X	X	X	X	X	X
Komodo	X	X	X	X	X	X

Un simple éditeur nécessite la mise en place d'un environnement complémentaire, avec le shell comme pivot central pour toutes les opérations d'édition, de recherche, etc. L'EDI, quant à lui, est une sorte de centre de commandement et doit être configurable au maximum. Le choix de l'une ou l'autre de ces solutions est fonction de la maîtrise du shell et parfois des conventions mises en place autour du projet.

Cette section se concentre uniquement sur les fonctionnalités liées à l'édition du code et regroupe dans le tableau comparatif 2.1 une sélection des meilleurs éditeurs du marché. Les critères de comparaison sont les suivants :

- **CC : complétion de code** : la saisie des mots est complétée automatiquement;
- **II : indentation intelligente** : en fonction de la position, le passage à la ligne positionne le curseur au bon niveau;
- **RD : raccourci vers la définition** : un raccourci clavier permet d'atteindre la définition d'une fonction ou classe utilisée;
- **SE : soulignement des erreurs en temps réel** : les erreurs de syntaxe sont immédiatement soulignées;
- **GS : intégration du gestionnaire de source** : les modifications sont directement commitées¹;
- **FREE** : Open source et/ou gratuit.

Voici les sites des éditeurs présentés :

- **Stani's** : <http://pythondide.blogspot.com>;
- **WingIDE** : <http://www.wingware.com/wingide>;
- **PyDev** : <http://pydev.sourceforge.net>;
- **Komodo** : http://www.activestate.com/products/komodo_ide.

Les éditeurs généralistes comme *Vim* ou *gnu/emacs* n'ont pas été présentés mais proposent des modes Python qui fournissent l'intégralité des fonctionnalités du tableau. *PyDev* quant à lui, est le seul éditeur intégré à un véritable EDI : *Eclipse*. Ce dernier propose des dizaines de fonctionnalités annexes et se positionne comme le seul EDI complet pour Python. Enfin, certains éditeurs, considérés comme moins complets, n'ont pas été présentés (*Idle*, *Eric3*), même s'ils conviennent à certains développeurs.

2.4 Ce qu'il faut retenir

Chaque membre d'une équipe peut vouloir opter pour un environnement particulier, fruit de ses précédentes expériences. Si les méthodes doivent être communes, le développeur doit rester le seul maître de son outil. C'est pourquoi il est important pour les chefs de projets de ne pas imposer un système d'exploitation ou un outil d'édition de code particulier, mais simplement de faire des suggestions s'il lui semble que le développeur peut être plus productif. Car forcer à l'adoption d'un outil ne rendra jamais un programmeur non conciliant plus productif.

De plus, une équipe composée de développeurs qui travaillent dans des environnements différents est une richesse supplémentaire, aussi bien au niveau de la portabilité du logiciel développé, qu'au niveau des influences culturelles apportées.

Le prochain chapitre s'intéresse aux principes généraux de conception d'une application, en respectant une règle pour éviter de déborder sur un sujet beaucoup trop vaste : il n'est basé sur aucune méthodologie d'analyse existante et se contente de fournir des axes de réflexion aux architectes.

¹Voir section 8.1.4.

Chapitre 3

Architecture logicielle



Méfiez-vous de l'ingénieur informatique qui porte un tournevis.

Robert Paul

3.1 Philosophie

L'architecture logicielle est la définition des structures d'un programme, de leurs interactions et de la manière dont *le monde extérieur*¹ les perçoit et les utilise. La phase de conception de l'architecture d'un logiciel est garantie de sa pérennité : il est facile de créer un programme qui répond aux attentes des utilisateurs à un instant donné. Mais qu'en est-il lorsqu'une nouvelle fonctionnalité doit être ajoutée, ou un élément modifié ? Comment le programme résiste aux mois, voire aux années de correctifs, patches ou mises à niveau, aux changements d'échelle au niveau de la quantité de données manipulées, etc. ?

Quelques exemples typiques de problèmes issus d'un mauvais design :

¹ Des utilisateurs finaux ou d'autres programmes.

- le nom de la table en base de données ne peut pas être changé facilement : il faut revoir l'intégralité du code car ce nom y est utilisé un peu partout ;
- l'écran d'affichage des articles met plusieurs secondes à s'afficher, il avait été prévu pour gérer quelques centaines d'articles tout au plus, et doit faire face à présent à plusieurs milliers d'articles ;
- les échanges de données avec d'autres logiciels nécessitent de redévelopper des modules d'import-export : le format de fichier propriétaire du logiciel n'est pas facile à utiliser par les applications tierces.

Comme pour toutes les activités liées au développement logiciel, c'est avant tout l'expérience qui prévaut : il faut bien souvent avoir vécu les problèmes de plein fouet pour les éviter par la suite. Mais la majorité des écueils peuvent être évités en respectant une certaine hygiène dans la conception d'un logiciel. C'est ce que résume ce chapitre à travers les **cinq lois de l'architecture logicielle**, qui synthétisent les bonnes pratiques lors de la conception de la structure d'une application. Ce chapitre ne présente pas de méthode de conception précise car il en existe des dizaines, mais les principes communs qui doivent être suivis lors de la conception d'une application.

3.2 Les cinq lois de l'architecture logicielle

On pourrait résumer ces cinq lois en cinq mots :

- modularité ;
- extensibilité ;
- simplicité ;
- paramétrabilité ;
- standardisation.

3.2.1 Modularité

Une application est composée de lignes de code regroupées dans des *briques logicielles*² elles-même organisées en ensembles de plus grosse taille, et ainsi de suite, jusqu'à former l'ensemble complet (la

² On parlera dans ce chapitre de brique, de composant ou encore d'ensemble, pour décrire tout regroupement de code.

section 4.2.7 présente les niveaux de structuration possible en Python). Construire un ensemble modulaire consiste à organiser ces regroupements en suivant une logique de composition basée sur les interactions possibles de chaque partie du code. Chaque niveau de composition est formé en regroupant les éléments qui concernent le même champ fonctionnel. La meilleure technique est de considérer l'élément regroupant comme une application à part entière, indépendante du reste.

Les briques peuvent être organisées par couches :

- les **composants périphériques**, qui fournissent un accès aux éléments extérieurs ;
- les **composants fonctionnels** ;
- les **composants médiateurs**, qui lient l'ensemble.

Composants périphériques

D'un point de vue pratique, découper une application en briques se fait de l'extérieur vers l'intérieur : il s'agit de définir dans un premier temps les interactions avec les données, les utilisateurs et les éventuelles applications tierces, puis de rentrer au cœur du logiciel.

Dans la figure 3.1, cette ceinture de composants fournie à l'application un accès à une base de données, à un serveur LDAP, et à des Web Services. Par la suite, tout accès à l'une de ces ressources devra se faire exclusivement par le biais de ces briques, qui manipulent elles-mêmes des éléments de plus bas niveau comme des bibliothèques tierces.

Composants fonctionnels

Vient ensuite une deuxième couche de composants, qui forment l'application et qui fournissent des fonctionnalités concrètes (deuxième niveau dans la figure 3.1). Toujours dans un souci de modularité, ces composants doivent être pensés pour fonctionner de manière autonome et ne dépendre, lorsque c'est nécessaire, que de la ceinture extérieure. Cette autonomie confère à l'application plus de robustesse : les évolutions futures de chacun des composants auront un impact plus contrôlable sur l'application.

Composants médiateurs

Les composants doivent bien sûr communiquer entre eux pour livrer leur fonctionnalités à l'application. Pour ne pas créer d'interdépendances, ces espaces d'échanges doivent être isolés dans un ou plusieurs *composant médiateur*, qui contient le *code glue*. Ces médiateurs sont les éléments de plus haut niveau, qui orchestrent les opérations et qui constituent le cœur de l'application. Ils se basent parfois sur des systèmes d'abstraction pour manipuler les autres briques. Le plus courant est la *component architecture*, qui met en place un système d'annuaire où chaque brique peut venir s'inscrire comme fournisseur de fonctionnalité. Cette liaison lâche offre alors suffisamment de souplesse pour que les composants médiateurs ne manipulent pas directement les autres briques. Pour la plupart des applications³, isoler les échanges dans des composants orchestre sans avoir à utiliser un système d'abstraction est largement suffisant (noyau dans la figure 3.1).

Les composants médiateurs peuvent être vus comme des scripts de haut niveau, et sont les briques les plus souvent modifiées.

Évolution des composants

Chaque composant présente une *API*⁴ qui est utilisée par les autres composants. C'est l'interface publique, avec laquelle le code client manipule le composant. Cette API doit être suffisamment indépendante de l'implémentation et du fonctionnement interne du composant pour que les évolutions du code n'impactent pas le code utilisateur. Cette stabilité permet aux autres composants de construire du code au-dessus sans être troublés par d'éventuelles modifications. Les composants sont comparables dans ce principe à des boîtes noires. C'est, ce besoin de stabilité qui introduit pour chacun de ces composants la nécessité de gérer des versions : une fois un composant *released*, c'est-à-dire fourni aux autres composants comme brique fonctionnelle, toutes les modifications ultérieures sont regroupées dans des livraisons, qui passent le composant à un nouvel état stable. Chaque livraison introduit de nouvelles fonctionnalités, des correctifs de bugs et, parfois des changements de l'API, qui entraînent des modifications en conséquence dans les composants utilisateurs. Ces évolutions

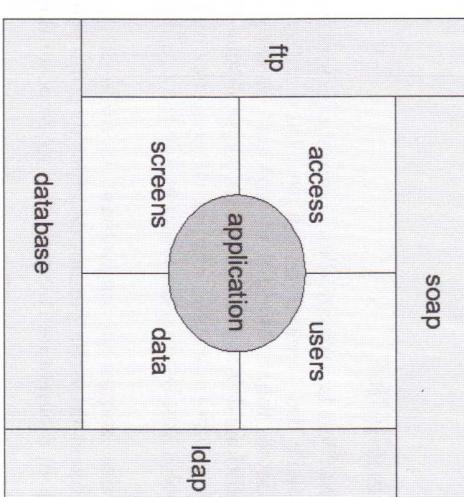


FIG. 3.1 – Composant médiateur

³Sauf les frameworks.

⁴Application Programming Interface

d'API se font en général de manière progressive, pour éviter les impacts sur les applications. La meilleure stratégie consiste à introduire les nouvelles API et à conserver les anciennes pendant un certain nombre de versions, tout en prévenant les utilisateurs de leur disparition prochaine.

Chaque composant, fonctionnel ou médiateur, maintient une liste des autres composants qu'il utilise. Cette liste est souvent sous la forme d'un fichier texte nommé *DEPENDENCIES.txt*, où chaque composant tiers est noté, avec le numéro de version utilisé. Les systèmes de déploiement évolués comme *setuptools*⁵ fournissent en outre la possibilité d'installer les composants tiers lorsqu'ils sont disponibles en ligne.

3.2.2 Extensibilité

*L'extensibilité*⁶ est la capacité d'un composant à fonctionner quelle que soit l'échelle des données manipulées. En d'autres termes, les performances de ses fonctionnalités ne doivent pas ou peu changer lorsque les conditions deviennent extrêmes. Ce problème est fréquent dans les systèmes informatiques, car les concepteurs imaginent des solutions adaptées aux besoins du moment sans se projeter sur leurs évolutions.

Concevoir des briques en surveillant la taille de tous les paramètres d'entrée et de sortie permet de réfléchir à des solutions plus solides. Il suffit au moment de la conception de se poser des questions comme :

- Comment va réagir le système avec 10 000 utilisateurs ?
- Est-ce que cette liste est capable d'afficher 100 000 articles ?
- Combien de temps va mettre cette fonction avec 3 gigas de données ?

Ces réflexions entraînent une modification des techniques de programmation. Les algorithmes deviennent plus robustes, leur complexité en durée et en espace mémoire mieux contrôlée pour se rapprocher d'une évolution linéaire. Des outils de tests des performances peuvent dans ce cas servir à calibrer le code. Le script basé sur la mesure des *Pystones* est un bon outil pour mesurer de manière empirique les performances du code. Il est présenté dans la section *scripts*

du site <http://programmation-python.org>. La simplicité d'un composant, que ce soit son utilisation ou son fonctionnement, facilite sa maintenance et son adoption. Mais trouver des solutions simples et efficaces est probablement la tâche la plus ardue du métier de développeur. Il s'agit de réussir à contrôler l'évolution de son code dans l'optique de conserver cette simplicité.

3.2.3 Simplicité

La simplicité d'un composant, que ce soit son utilisation ou son fonctionnement, facilite sa maintenance et son adoption. Mais trouver des solutions simples et efficaces est probablement la tâche la plus ardue du métier de développeur. Il s'agit de réussir à contrôler l'évolution de son code dans l'optique de conserver cette simplicité.

Contrôler l'évolution du code

À moins d'avoir un esprit torturé, un développeur conçoit intuitivement du code simple : la complexité est introduite par les évolutions du code. Chaque petite modification, aussi simple soit-elle, peut rendre un composant complexe. Quelques signes de cette évolution négative :

- la signature des méthodes ou fonctions devient de moins en moins précise, et les développeurs utilisent les *arguments mafiques* (voir section 4.2.7) ;
- le rôle des classes est de plus en plus large et leurs méthodes sont de plus en plus nombreuses ;
- la taille et la complexité des fonctions et méthodes s'allongent.

Pour lutter contre cette *dissolution structurale*, il convient pour chaque modification de vérifier que le code ajouté ou remplacé conserve la simplicité de l'ensemble.

Il ne faut jamais hésiter à créer de nouvelles classes, de nouveaux modules, ou composants, pour que le code conserve sa simplicité. *Small is beautiful!*

La section 5.3.2 fournit des conseils concrets pour maîtriser le code Python produit.

3.2.4 Paramétrabilité

La faculté d'un composant à s'adapter à plusieurs contextes d'exécution est sa *paramétrabilité*. Plus les composants utilisateurs doivent ajouter de code au-dessus et moins l'application est robuste. Si au

⁵<http://peak.telecommunity.com/DevCenter/setuptools>

⁶*Scalability* en anglais

contraire le composant fourni suffisamment d'options pour que le code appelant n'ait pas besoin d'adapter son fonctionnement, il reste souvent sur les évolutions du code nécessaire à son fonctionnement et sur les interactions.

La solution est de penser son code générique et parfois *génératif*.

Programmer générique

Programmer générique consiste à concevoir du code capable de fonctionner dans des contextes différents. Une classe en charge d'afficher une liste d'articles, par exemple, sera conçue pour être capable d'afficher n'importe quelle donnée puis paramétrée pour afficher les articles. Cette généralisation permet de réutiliser la fonctionnalité partout où une liste est utile. Le code est moins redondant, et à terme plus robuste, même si la démultiplication des combinaisons introduit dans un premier temps plus de bugs qu'une solution *one shot*.

Certains développeurs ont parfois tendance à confondre généréricité et évolutivité : ils conçoivent du code qui n'est paramétrable qu'en *dérivant* (voir section 5.2.1) les structures existantes. Ils se trompent de cible et d'objectif : un composant, lorsqu'il n'est pas abstrait (c'est-à-dire conçu pour être étendu), fournit des fonctionnalités. Ces fonctionnalités ne doivent pas être obtenues par une couche supplémentaire de code mais directement accessibles et paramétrables.

L'aspect dynamique de Python rend cette réflexion floue : le paramétrage peut lui-même être composé de scripts Python au lieu d'être sous la forme de fichiers de configuration au format XML ou INI⁷ par exemple. Mais ce code ne fait pas partie en soi du code des composants : il doit rester indépendant des structures et des stratégies de composition du code générique.

Programmer génératif

Une autre technique plus subtile de la programmation de briques réutilisables est la programmation générative, qui consiste à créer des composants abstraits qui génèrent des composants concrets. Ce type de programmation est basée dans les langages compilés par la génération de code source, mais peut être pratiquée à la volée en Python, par le biais des *méta-classes*(voir section 5.2.1).

3.2.5 Standardisation

La standardisation consiste à opter pour des formats d'échange de données, et des protocoles de communication communément adoptés par le domaine de l'application. À moins de travailler dans l'embarqué, il ne faut jamais inventer un nouveau protocole d'échange TCP ou un format de données propriétaire.

Opter pour des standards :

- facilite l'interopérabilité entre les composants et les applications tierces;
- facilite l'adoption de bibliothèques tierces qui partagent ce standard;
- permet de manipuler les données ou les briques logicielles avec des applications répandues et universelles, comme les tableurs.

Quelques exemples de standards :

- le XML, pour les données et la configuration ;
- les fichiers au format *INI*, *CSV* ;
- les WebServices et Soap, ou l'XML-RPC ;
- le FTP, le SSH, etc.

3.3 Ce qu'il faut retenir

Quelle que soit la méthodologie employée pour concevoir une application, le respect des cinq règles présentées garantit une croissance équilibrée de la base de code. Mis à part les aspects sur la métaprogrammation, ces lois s'appliquent dans tous les langages.

Le chapitre suivant s'intéresse au développement Python à proprement parler et introduit également, après un résumé de la syntaxe, des bonnes pratiques en termes de structuration des programmes (voir section 5.2).

En effet Python offre la possibilité au programmeur d'intervenir en amont de linstanciation de classes ou d'objet, et donc de faire varier les définitions de classe en fonction de paramètres.

Les techniques de métaprogrammation sont toutefois plus complexes à appréhender, et ne doivent pas être utilisées à outrance pour que le code reste simple.

⁷Format standard des fichiers de configuration sous Windows.

Chapitre 4

Développement



En théorie, il n'y a pas de différences entre théorie et pratique. En pratique, il y en a.

Chuck Reid

4.1 Philosophie

4.1.1 L'art de programmer

Dans le domaine de la programmation, les ouvrages aux titres ronflants, qui commencent par „*Maitriser...*” ou terminent par „...en 5 jours” ou encore „...pour l'impatient”¹ donnent l'illusion que cette activité ne nécessite que la maîtrise d'un ensemble de règles techniques. Le manuel de référence de Python² serait alors un mode d'emploi complet, permettant de concevoir toute sorte d'applications. Or, et heureusement pour nous autres développeurs, la réalité est tout autre. Développer des applications nécessite de faire appel à des réflexions poussées pour projeter dans le contexte cible le fonctionnement imaginé, et de manipuler des abstractions pour concevoir du code facile à

¹Toute ressemblance avec des titres existants n'est que fortuite.

²En ligne sur <http://python.org/doc>.

maintenir et réutilisable. Et pour mener à bien ces activités, les cinq qualités fondamentales sont :

- la créativité;
- la rigueur;
- l'esprit de synthèse;
- le goût;
- l'adaptabilité

Ces qualités ne s'expriment, et s'affinent, bien évidemment, qu'à travers l'expérience. Et la durée nécessaire pour la plupart d'entre nous, misérables mortels, pour maîtriser l'art de programmer, se rapproche plus des dix ans [3] [7] que des dix jours que certains ouvrages annoncent³.

La créativité

Beaucoup⁴ de musiciens ou encore de peintres retrouvent dans la programmation, lorsqu'ils s'y adonnent, les mêmes plaisirs intellectuels que dans leurs activités premières. Un logiciel est comparable à bien des égards à une œuvre d'art dans le processus de création⁵ que le développeur met en œuvre. Cette créativité s'exprime à travers des associations originales entre différents concepts dans le cadre de la résolution de problèmes. Ces associations originales deviennent des idées et cheminent vers des lignes de codes pour former les fonctionnalités souhaitées. Bien sûr, la plupart du temps, les idées qui traversent l'esprit d'un développeur ont déjà été vues, revues et corrigées par des milliers de développeurs dans le passé, et tout est question de recyclage. Les *codes patterns* et *design patterns* peuvent être vus en ce sens comme des recueils d'idées synthétiques [5] prêtes à l'emploi. Mais cet état d'esprit permet de faire évoluer les applications et parfois de proposer des fonctionnalités originales, voire révolutionnaires.

La rigueur

La rigueur est fondamentale dans l'activité de programmation : le code conçu devient de plus en plus gros et aucun compromis ne doit être fait sur la qualité de ce qui est produit, sous peine d'accoucher d'une pieuvre à six têtes, impossible à maintenir et à faire évoluer. Cette rigueur s'exprime par la nécessité pour chaque développeur de cultiver une résistance à l'animillement intellectuel que représente l'écriture de lignes de code⁶. Rester alerte et critique sur sa propre production, et réussir à ne pas se laisser emporter par les passages monotones, c'est toute la difficulté de ce métier.

La pratique de cette rigueur se retrouve dans :

- La suppression immédiate du *bad smelling code*.
- L'utilisation de conventions de codage et de nommage.
- Le respect des rythmes de conception et de *coding*.

Suppression immédiate du *bad smelling code* *Bad smelling code* pourrait être traduit en *code bouche*. C'est ce code qui accroche notre regard lorsque l'on parcourt un fichier, car il paraît mal conçu, illisible ou lourd. Parfois faux. Si la modification est rapide, elle est appliquée directement. Dans le cas contraire, un *todo* est ajouté en commentaire⁷.

³Rassurez-vous, il est tout à fait possible de se faire plaisir dès le premier jour.

⁴Ceux que j'ai pu rencontrer...

⁵La différence fondamentale est que l'art par définition ne sert à rien, alors qu'un logiciel a souvent une finalité.

Peter Knuth, le père de TeX, a inventé le *literate programming* [8], qui permet d'intégrer à un fichier TeX des programmes qui peuvent être exécutés. Cette association originale entre documentation et programmation a ensuite abouti à la création des *doctest* (voir section 7.3.2) en Python, qui ont permis d'insérer des exemples de code exécutables dans la documentation. Enfin, les doctest ont été utilisés comme principal outil dans la programmation dirigée par les tests dans de nombreux frameworks actuels, et ont contribué à une augmentation sensible de la qualité des programmes : concevoir du code en expliquant son fonctionnement dans une documentation permet de prendre le recul nécessaire, dans le rôle de l'utilisateur final.

Utilisation de conventions de codage et de nommage L'utilisation de conventions permet d'homogénéiser le code et de faciliter sa compréhension. Lorsqu'elles sont mises en place, il est nécessaire de rester rigoureux sur leur adoption, qui est de moins en moins pertinente par la suite : la convention est instinctivement appliquée dans le temps.

Respect des rythmes Respecter les rythmes de conception, d'écriture de code et de tests, suivant la méthodologie définie dans le chapitre 7 est difficile à mettre en pratique et ne s'acquiert qu'avec le temps. L'erreur la plus récurrente consiste à continuer le design d'un code de tête en même temps qu'il est écrit. Cette mauvaise habitude nécessite souvent de revenir sur ce qui a été codé pour le refaire, car aucune réflexion n'a été effectuée. Il peut même arriver que des erreurs de design ajoutées par ce biais perdurent dans le code : le développeur, au lieu d'affronter le problème en s'éloignant du clavier pour *réfléchir*, continue à rafistoler son code bancal pour tenter d'arriver au résultat malgré tout. Et tant pis pour la qualité.

La sirène du code écrit en flux continu est forte, car plus confortable intellectuellement : elle nous place dans un état de semi-conscience, où le code est produit de manière systématique. Elle continue à hanter les développeurs les plus expérimentés. Il faut se faire violence pour casser ce mauvais réflexe. Un bon moyen d'y arriver est d'arrêter d'écrire du code si aucun test n'a encore été imaginé pour ce dernier.

L'adaptabilité

L'adaptabilité n'est pas l'imitation. C'est un puissant moyen de résistance et d'assimilation

Gandhi

Comment réussir à conserver la rigueur, l'esprit de synthèse, et tout ce qui a été cité au préalable, sans avoir envie de détourner le regard de ce petit écran ? La solution la plus efficace consiste à associer au code une notion d'esthétisme et à le cultiver. Le vocabulaire utilisé pour décrire le code devient alors comparable à celui employé pour l'art ou encore la cuisine.

Exemples entendus réellement :

- "Il faut que tu fasses mijoter les données avant de les renvoyer"
- "Tu peut pas laisser ta classe comme ça, elle va pourrir"
- "Cette API est imbuvable"

Le goût

Comment supporter un métier qui :

- casse le dos ;
- rend aveugle ;
- peut rendre asocial ;
- nous oblige à faire la hotline informatique pour toute la famille, les voisins, les amis, et les amis des amis ?⁸

L'esprit de synthèse

La règle d'or dans la conception d'un logiciel est de ne jamais avoir deux portions quasi identiques de code. Si c'est le cas, il faut *factoriser* et supprimer les doublons. Ce principe s'applique à tous les niveaux du code. Il faut toutefois garder en tête qu'à partir d'une certaine granularité seuil, on retrouve évidemment des passages similaires dans le code sans avoir forcément besoin de le changer : c'est le cas des boucles de traitement sur des séquences, qui sont plus courtes et plus simples à écrire qu'un éventuel équivalent factorisé. Ce dernier serait alors plus long à paramétriser qu'une écriture directe. On parle du *tissu* du programme.

La informatique est en perpétuel mouvement, et il est essentiel pour les développeurs de s'adapter rapidement aux évolutions. Ce changement perpétuel nécessite l'assimilation en continu de nouveaux concepts mais aussi un désapprentissage de fond et une modification continue de ses habitudes et convictions.

Être informaticien aujourd'hui, c'est :

- **Rester alerte.** Une des tâches journalières est de continuer un apprentissage de fond en lisant les actualités informatiques, articles et autres livres.
- **Intégrer le changement continual comme philosophie.** Garder à l'esprit que toute parcelle de code ou de design est

⁸Le jour où ils comprendront qu'un développeur n'est pas forcément un technicien matériel ou un spécialiste des OS.

- **Susceptible de changer.**
- **Expérimenter:** Effectuer des tests, des prototypes, pour évaluer les outils du marché.
- **Se projeter:** Imaginer le code dans des conditions extrêmes, ou l'application dans d'autres frameworks ou langages.
- **Prendre du recul.** Alterner les visions locales ou globales pour ne pas se noyer dans les détails, à l'image d'un joueur de Go qui observe le plateau complet à chaque coup, au lieu de se laisser emporter par une série de coups concentrés en une seule région.

Ces bons réflexes, accompagnés de pratiques de programmation éprouvées, rendent *agile*.

Le reste de ce chapitre présente un rappel rapide de la syntaxe de Python.

4.2 Syntaxe de Python

Python est probablement l'un des langages les plus agréables à écrire : sa syntaxe se rapproche beaucoup du pseudo-code, et un développeur issu d'un autre langage s'y adapte en quelques heures. Cette expressivité permet de se passer très rapidement des habitudes *cheatsheet*⁹ et de coder de manière fluctuelle. Cette section présente très rapidement la syntaxe de Python¹⁰.

4.2.1 L'indentation

L'originalité du langage provient de la structuration du code : l'indentation est utilisée en lieu et place des habituels accolades ou point-virgule, et le caractère : précède chaque niveau :

```
>>> for i in range(2):
...     if i == 0:
...         print 'un'
...     else:
...         print 'deux'
...
un
```

deux

Des espaces ou des tabulations peuvent servir à l'indentation, mais la convention est d'utiliser quatre espaces. En général, l'éditeur de code est réglé pour que la saisie d'une tabulation soit remplacée par quatre espaces.

4.2.2 Les variables

Python est typé dynamiquement et les références à des objets en mémoire, ou *variables* n'ont pas à être associées à un type donné :

```
>>> a = 1
>>> a = 'un'
>>> a = 'un',
```

'un'

Cette particularité que l'on retrouve dans d'autres langages facilite grandement l'écriture du code et réduit sa taille. Mais cet avantage entraîne aussi un défaut qui doit être comblé par de bonnes pratiques de programmation : une variable pouvant être associée à tout type d'objet, les erreurs de programmation relatives à l'affectation de valeurs ne se voient qu'au moment de l'exécution du code :

```
>>> a = 1
>>> b = 2
>>> a + b
3
>>> b = 'deux'
>>> a + b
Traceback (most recent call last):
...
unsupported operand type ... 'int' and 'str'
```

4.2.3 Le tout objet

Sur le même principe que Smalltalk, tout élément est objet en Python, que ce soit les chaînes de caractères, les entiers, ou encore les booléens :

⁹Littéralement : feuilles de pompe.

¹⁰Pour des informations plus détaillées, voir [12].

```
>>> dir(a)[-20:-1]
[ 'ljust', 'lower', 'lstrip', 'partition',
  'replace', 'rfind', 'rindex', 'rjust',
  'rpartition', 'rsplit', 'rstrip', 'split',
  'splitlines', 'startswith', 'strip',
  'swapcase', 'title', 'translate', 'upper' ]
>>> a.replace('o', 'u')
'pumpuku'
>>> a.title()
'Pompoko'
```

Ce modèle minimise le défaut présenté au point précédent : les fonctionnalités applicables à un objet y sont directement attachées.

4.2.4 Les types et objets notables

Python fournit un certain nombre de types de base, qui permettent de construire facilement tout type de représentation de données. Les types peuvent être regroupés en types de base, comme les entiers ou les chaînes de caractères, et en type conteneurs, comme les listes et les dictionnaires.

Les types de base

Les types de base notables sont :

- **int** : entiers variant de -999999999 à 999999999;
- **long** : entiers en dehors de l'intervalle couvert par le type **int**;
- **float** : (pseudo-)réels;
- **str** : chaînes de caractères;
- **unicode** : chaînes de caractères unicode;
- **list** : séquence ordonnée de valeurs hétérogènes;
- **tuple** : séquence ordonnée et figée de valeurs hétérogènes;
- **dict** : dictionnaire de valeurs hétérogènes (ou *mappings*).

int et *long* sont également des primitives, qui permettent de créer une instance avec une valeur initiale qui peut être d'un autre type. Si le *transnstypage* est possible, il est effectué. Une erreur *ValueError* est levée le cas échéant :

```
>>> int('1')
1
>>> int('1.mais.bon.2.est.bien.aussi')
Traceback (most recent call last):
...
ValueError: invalid literal for int()
```

Les entiers : int et long Les entiers sont représentés en Python par le type *int*, qui passe le relais de manière transparente au type *long* dès que l'intervalle *[-999999999, 99999999]* est dépassé :

```
>>> entier = 999
>>> entier
```

Enfin, le type *long* ne connaît pas de limites à part la taille physique de la mémoire, car sa représentation est basée sur des matrices :

```
>>> euro_million = 100000000000000000000000000000000L
>>> euro_million
```

Les réels : float Les réels sont représentés par des nombres à virgule flottante, c'est-à-dire deux entiers séparés par un point. La précision de ces valeurs dépend de la machine, et ce type n'est pas utilisé dans les calculs scientifiques. Il l'est parfois pour les calculs monétaires dont la précision n'excède pas quelques chiffres après le séparateur.

```
>>> 7.9 + 7.
14.9
```

```
>>> 9 + .1
9.099999999999996
```

```
>>> 9 + 1.98
10.98
```

Il est aussi possible d'utiliser des exposants.

```
>>> 10e3
10000.0
>>> 10.6e3
10600.0
```

À noter que Python 3000¹¹, prévu fin 2008, unifiera ces deux types, pour ne conserver que l'implémentation *long*.

Les chaînes de caractères : str et unicode Les chaînes de caractères ont deux représentations : le type *str* et le type *unicode*. Historiquement, *unicode* a été introduit pour permettre aux applications internationales d'utiliser un jeu de caractères universel, qui possède l'intégralité des signes utilisés dans tous les langages écrits. Cette table représente plus de 180 000 signes.

En pratique, les applications utilisent le type *str* pour les chaînes dédiées aux données, et *unicode* pour celles spécifiques au langage, comme les éléments d'interface ou les données typées comme des noms

ou des textes saisis. Les chaînes *unicode* sont juste précédées du signe *u*.

```
>>> chaine = "l'enfer _c'est _les _autres"
>>> chaine2 = u"Jean-Paul_Sartre"
>>> chaine
"l'enfer _c'est _les _autres"
>>> chaine2
u'Jean-Paul_Sartre'
```

Cette différenciation n'est justifiée que par l'introduction progressive du support d'*unicode* dans Python. Un type unifié sera mis en place pour la version 3000. Toutes les chaînes seront alors en *unicode*.

Quelques opérateurs En plus des opérateurs classiques comme *+*, *** et */*, Python fournit un élément de comparaison supplémentaire *is*, qui permet de vérifier l'égalité entre deux objets. Cette comparaison ne concerne pas la valeur prise par ces objets mais l'adresse mémoire. Deux objets de type *list* ne seront donc jamais égaux, même si ils ont la même série de valeurs. Les objets à valeur constante comme les objet de type *string* pourront être égaux, puisque Python leur attribue le même espace mémoire.

```
>>> t = "Youhou"
>>> y = "Youhou"
>>> t is y
True
```

```
>>> t = [1]
>>> y = [1]
>>> t is y
False
```

L'usage le plus fréquent de *is* est avec *None*, pour vérifier par exemple la sortie d'une fonction.

Les séquences : list et tuple Les séquences sont des listes d'éléments hétérogènes. Python fournit nativement deux types de séquences : le type *list* et le type *tuple*.

Le type *list* est le plus fréquemment employé, et fournit un jeu de méthodes pour lire et modifier la séquence. Il est caractérisé par des crochets.

¹¹ Version 3, après la série des 2.x.

```
>>> ma_liste = []
>>> ma_liste.append(1)
>>> ma_liste.append("deux")
>>> ma_liste.append(3.0)
>>> ma_liste.append(u"allon-z-o-bois")
>>> ma_liste
[1, 'deux', 3.0, u'allon-z-o-bois']
>>> ma_liste.insert(0, 'eeeeettt')
['eeeeettt', 1, 'deux', 3.0,
u'allon-z-o-bois',
>>> ma_liste.pop()
u'allon-z-o-bois',
>>> ma_liste
['eeeeettt', ..., 1, 'deux', 3.0]
```

L'ensemble des méthodes du type *list* peut être obtenu en invoquant la commande *help(l)* dans le prompt, ou en utilisant le code ci-dessous, qui filtre les méthodes disponibles, et affiche les docstrings associés :

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

```
>>> print_api([])
```

L.*append*(*object*) — append *object* to end

L.*count*(*value*) — integer — return number of...

L.*extend*(*iterable*) — extend list by...

L.*index*(*value*, [*start*, [*stop*]]) — integer ...

L.*insert*(*index*, *object*) — insert *object* ...

L.*pop*([*index*]) — item — remove and return ...

L.*remove*(*value*) — remove first occurrence ...

L.*reverse*() — reverse *IN PLACE*

L.*sort*(*cmp*=None, *key*=None, *reverse*=False) ...

```
cmp(x, y) -> -1, 0, 1
```

Cet exemple utilise également l'opérateur d'appartenance **in**, qui permet de vérifier si un élément est contenu dans une séquence.

list, qui conserve l'ordre des éléments, est probablement le type le plus utilisé, car il répond à un besoin fréquent en programmation : manipuler des séquences d'éléments et effectuer des opérations sur chacun. De nombreuses fonctionnalités du langage utilisent les séquences comme paramètres d'entrée ou de sortie.

Cette caractéristique rend ce type légèrement plus rapide d'accès mais surtout fournit une indication sur la nature des informations qu'il contient. Le *tuple* est représenté par des parenthèses, qui peuvent être omises lorsque la séquence définie contient plus d'un élément. Lorsqu'elle ne fait qu'un élément, une virgule terminale permet de différencier le *tuple* d'un simple élément parenthésé.

Les éléments de ces deux types peuvent être accédés par leur position grâce à la notation suivante : *sequence[position]*. Voici une série d'exemples de *tuple* :

```
>>> ("je_ne_suis_pas_un_tuple",
...     je_ne_suis_pas_un_tuple,
...     ("je_suis_un_tuple", hin_hin_hin),
...     ('je_suis_un_tuple', hin_hin_hin),
...     1, 2, 3
...     (1, 2, 3)
...     >>> object(), 'truc', u'U'
...     (<object object at 0x23460>, 'truc', u'U')
...     >>> wish_list = ('Wii', 'PSP',
...                     1, pass_monaistère)
...     >>> wish_list[1]
...     'PSP'
```

Les mappings : dict Les mappings sont des séquences hétérogènes non-ordonnées, dont chaque élément est étiqueté avec une valeur unique. Cette étiquette peut être de n'importe quel type, du moment qu'il ne soit pas modifiable¹².

On parle de dictionnaires, et les *curly brackets* sont employés.

```
>>> mon_dico = {'1': 1, u'2': '2',
...               ('trois', 'oui', 'trois'):
...               ['trois']}
```

¹² *int*, *tuple* et *string* par exemple.


```

...
print mot
...
benny
hill
me
manque
snif
...
if i == 3:
    ...
continue
...
print i
...
if i == 4:
    ...
break
...
print i
...

```

La variable utilisée dans la boucle reçoit tour à tour chaque élément¹³. Lorsque la séquence utilisée est modifiable, il est nécessaire de prendre garde à ne pas impacter le déroulement de la boucle : *for* évalue dynamiquement les éléments à chaque cycle, et des effets de bords peuvent se produire. Une bonne pratique consiste à toujours utiliser des *tuples* pour éviter le problème. Voici un exemple de cette erreur :

```

>>> liste = [1, 2, 3, 4]
>>> for i in liste:
...     print liste[i]
...
liste = liste[:-2]
...

```

0	0
1	1
1	1
2	2
3	3
4	4

Traceback (most recent call last):

IndexError: list index out of range

4.2.6 pass, continue et break

Pour faciliter l'écriture des boucles, Python fournit trois mots-clés :

- **pass** : ne fait rien ;
- **continue** : interrompt l'exécution d'une boucle mais reprend au début de cette boucle à l'itération suivante ;
- **break** : arrête définitivement la boucle.

```
>>> for i in range(100):
...     if i == 2:
...         pass
...     else:
...         print i
```

¹³ Après la fin de la boucle, la variable existe encore, avec le dernier élément.

4.2.7 Les fonctions, classes, modules et paquets

Le code Python est structuré en :

- **fonctions**, qui sont des séquences de lignes de codes, composées de commandes et de variables ;
- **classes**, qui regroupent des fonctions, alors appelées *méthodes* ;
- **modules**, qui sont des fichiers regroupant des variables, fonctions et classes ;
- **paquets**, qui sont des dossiers contenant des modules.

Les fonctions

Une fonction, définie par le mot-clef **def** et un nom, regroupe une séquence de commandes, et peut être appelée avec des paramètres en entrée :

```
>>> def acronym(sentence):
...     letters = [word[0].upper()
...               for word in sentence.split()]
...     return u''.join(letters)
```

```

Le mot-clef return permet à la fonction de renvoyer un résultat.
Lorsqu'aucun résultat n'est renvoyé, c'est None qui est renvoyé :
>>> def le_vide():
...     pass
...
>>> print le_vide()
None

Les paramètres d'une fonction peuvent avoir des valeurs par défaut :
>>> def acronyme(sentence="gnu's_not_unix"):
...     letters = [word[0].upper()
...                for word in sentence.split()]
...     return ''.join(letters)
...
>>> acronyme()
'G.N.U,'

Il existe également des paramètres spéciaux capables d'intercepter un nombre indéfini de valeurs. La signature de la fonction devient polymorphe. Le paramètre, nommé par convention args et précédé d'une étoile, est placé à la fin de la liste des paramètres et aggège dans un tuple tous les paramètres qui n'ont pas été définis :
>>> def acronyme(sentence=u"gnu's_not_unix",
...                 *args):
...     letters = [word[0].upper()
...                for word in sentence.split()]
...     return u'%s_%s' % \
...            (u''.join(letters), u''.join(args))
...
>>> acronyme(u'bande_original',
...             u'par_enio_maurikonne',
...             u'B.O_par_enio_maurikonne')

```

Les paramètres d'une fonction peuvent avoir des valeurs par défaut :
...>>> def acronyme(sentence="gnu's-not-unix"):
... letters = [word[0].upper()
... for word in sentence.split()]
... return ''.join(letters)

Le mot-clé `return` permet à la fonction de renvoyer un résultat. Lorsqu'aucun résultat n'est renvoyé, c'est `None` qui est renvoyé :

```
>>> def le_vide():
...     pass
...
>>> print le_vide()
None
```

```
>>> acronym(u"ce_qu'il_fallait_demontrer")
u'C.Q.F.D'
>>> acronym(u'hors_service')
u'HS'
```

Enfin, un paramètre unique précédé de deux étoiles et placé juste après `*args` permet d'intercepter tous les paramètres nommés qui n'ont pas été interceptés par des paramètres classiques. Ils sont rangés dans un dictionnaire, conventionnellement nommé `kw` ou `keyword` :

Les classes

Les paramètres variables *args* et *kw*, par leur aspect magique, doivent être utilisés avec retenue car ils rendent les signatures des fonctions moins précises. La pratique la plus prudente consiste à les utiliser uniquement pour les traitements génériques de séquences de valeurs.

```
args=(u'quatre',)  
kw={',et': u'ca', ',prend': u'ca',  
,mais': u'encore'}
```

Les classes sont des regroupements de variables et de fonctions¹⁴. Le mot-clé `class` est utilisé et les éléments y sont définis comme ils le seraient dans une fonction :

```
>>> class MaClasse(object):
...     a = 3
...     b = "uu"
...
...     def concat(self):
...         return str(self.a) + self.(self.b)
...
>>> mon_objet = MaClasse()
>>> mon_objet.concat()
'3uu'
>>> mon_objet.a
```

¹⁴ Notion abordée en détail dans la section 5.2.

```
3
>>> mon_objet.b
'uu'
```

Une classe peut être vue comme un espace de noms.

Les modules

Les modules sont les fichiers qui contiennent le code Python, que ce soient des séquences d'opérations directes, des fonctions ou des classes. Par convention, les fichiers ont comme suffixe *py*. Lorsqu'un fichier est fourni à l'interpréteur, le code est exécuté.

Le fichier suivant, appelé *je-dis-oui.py* :

```
def dis_le():
    print "oui_!"
```

aura pour résultat :

```
dabox:/home/tarek tarek$ python je-dis-oui.py
```

oui !

Directives import et from Un module peut être utilisé par un autre module, via la directive *import*. Tous les éléments définis dans le module peuvent être accédés une fois le module importé, en les préfixant du nom du module :

```
>>> import je-dis-oui
oui !
>>> je-dis-oui.dis_le()
oui !
```

La directive *from* permet de préciser l'élément du module à importer :

```
>>> from je-dis-oui import dis_le
>>> dis_le()
oui !
```

Un joker peut être utilisé pour importer l'intégralité d'un module dans l'espace de nom. Dans notre exemple, *from je-dis-oui import ** aura pour effet d'importer tous les éléments du module. Cette écriture reste toutefois déconseillée car elle n'est pas explicite.

Directive d'encodage Lorsqu'un fichier est chargé, Python utilise par défaut le jeu de caractères ASCII. Ce format ne permet pas de charger des caractères étendus, comme les accents français. Une erreur est alors déclenchée :

```
dabox:/home/repos/guide tarek$ python essai.py
File "essai.py", line 2
SyntaxError: Non-ASCII character '\xc3' in file essai.py
on line 2, but no encoding declared;
see http://www.python.org/peps/pep-0263.html for details
```

Pour préciser le type d'encodage, il convient d'utiliser la directive *encoding* en première ou deuxième ligne du fichier, en utilisant ce format :

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

Pour les codes sources *francophones*, l'encodage conseillé est l'UTF-8 ou l'ISO-8859-15.

Variable *_name_* Lorsqu'un module est importé, tout son code est interprété. Pour différencier le module qui est exécuté par l'interpréteur des modules importés, il est possible de tester la variable *_name_* qui est automatiquement disponible dans le module. Cette variable prend la valeur *_main_* lorsque le module est celui qui est exécuté. Cette différenciation permet d'exécuter du code d'initialisation que lorsque le module est le module principal.

```
# fichier trucmuch.py
# -*- coding: UTF-8 -*-
```

```
def truc():
    """fait_le_much"""
print "much"
if __name__ == '__main__':
    truc()
```

Dans l'exemple ci-dessus, si le module est directement exécuté, la fonction sera appelée. S'il est importé dans un autre module, le code conditionné par le test sur la variable `_name_` sera omis.

Autres variables de module Lorsqu'un module est chargé, Python lui associe d'autres variables globales, comme :

- `_file_` : le nom du fichier ;
- `_doc_` : le docstring placé en tête du module.

Il est aussi possible d'ajouter des variables globales optionnelles pour enrichir les informations, comme `_all_`, qui permet de lister les éléments du module que le développeur souhaite rendre disponible lorsqu'un utilisateur utilise `import *`¹⁵.

```
--mib
|
|-- flashouillage.py
|
|-- shadoks
|
|-- gegene.py
```

...pourra être utilisée ainsi :

```
from xfiles.roswell import verite
from xfiles import mib
import xfiles
```

Les paquets

Les paquets sont les dossiers contenant des modules. Ils peuvent contenir eux-mêmes des paquets. Pour transformer un dossier en paquet il est toutefois nécessaire d'ajouter un module spécial, nommé `_init__.py`. Il sera chargé par l'interpréteur et peut contenir du code d'initialisation. Les espaces de noms sont alors les mêmes que pour les modules.

Les paquets sont chargés par Python lorsqu'ils sont trouvés dans le chemin de l'interpréteur, à savoir :

- le répertoire depuis lequel l'interpréteur est lancé ;
- les chemins définis par la variable système `PYTHONPATH`.

Ces chemins sont stockés dans la variable `path` du module `sys`, qui est un objet de type list qui peut être modifié programmatiquement pour ajouter ou enlever des chemins.

La structure suivante, pour peu que le répertoire `xfiles` soit dans le chemin... :

```
-- xfiles
```

```
  |
  |-- roswell
  |   |
  |   |-- verite.py
```

¹⁵Ce type d'import étant déconseillé, `_all_` n'est pas souvent utilisé, sauf dans certains frameworks.

Chapitre 5

Bonnes pratiques



Tout peut être amélioré.

C. W. Barron

5.1 Philosophie

Passée la compréhension de la syntaxe, un développeur doit maîtriser un certain nombre de concepts et de pratiques pour tirer le meilleur du langage qu'il utilise. Ce chapitre présente quelques notions essentielles complémentaires au chapitre précédent, à savoir :

- **le bon usage de l'objet**, qui creuse d'avantage sur la grammaire objet et ses avantages;
- **le bon usage de Python 2.5**, qui fournit les *code patterns* les plus pratiques;
- **la maîtrise des outils d'assurance qualité**, qui explique comment contrôler la qualité du code produit.

5.2 Du bon usage de l'objet

Python est un langage *multiparadigme*, c'est-à-dire qu'il ne force pas le développeur à utiliser une technique particulière pour concevoir son code, comme la programmation orientée objet ou la programmation fonctionnelle. Cette souplesse est une des raisons du succès du langage auprès des communautés scientifiques non-informatiques : écrire un programme qui marche ne nécessite pas un bagage informatique conséquent, contrairement à des langages comme Java, où une compréhension du modèle objet est nécessaire.

Mais ce modèle devient inévitable dès lors que les projets grossissent, pour éviter que le code devienne monolithique et difficilement modifiable. Cette section regroupe une synthèse de la programmation orientée objet, suivie d'un recueil de *design patterns*, qui sont des modèles d'organisation de classes destinés à répondre à des cas d'utilisation récurrents.

5.2.1 Fonctionnement et intérêt des objets

Un objet est une représentation d'un concept, d'une idée ou d'une entité du monde physique. Il permet de :

- regrouper les données d'un programme en structures logiques ;
- regrouper le code qui s'applique spécifiquement à ces données ;
- vectoriser le code et faciliter son évolution.

L'objet comme structure de données

Pour comprendre le fonctionnement des objets, prenons l'exemple d'une application en charge de gérer des livres et leur contenu. Un livre peut être représenté par un objet *Livre* qui contient un certain nombre d'objets *Page*. L'objet *Livre* contient les pages référencées par des numéros de page et des caractéristiques supplémentaires, comme le titre et le nom des auteurs. L'objet *Page* peut contenir le texte et les notes de bas de page. Ces modèles d'objets sont définis dans des classes, et chaque *instanciation* de classe représente un objet en mémoire. Une instantiation est effectuée en invoquant la classe comme une fonction :

```
>>> class Book(object):
...     def __init__(self, title, authors=None):
...         self.pages = {}
```

Les objets peuvent donc être considérés comme des regroupements de données, avec un état initial défini au moment de la création par le biais d'une fonction associée à la classe, ou *méthode*, qui s'applique spécifiquement à l'objet créé. Cette méthode est appelée *constructeur* et son premier argument est l'objet lui-même, et est noté *self* par convention. Lorsque *Livre('Python', 'Tarek Ziadé')* est appelé, un objet de type *Livre* est créé en mémoire, plus *__init__(self, 'Python', 'Tarek Ziadé')* est appelé avec cet objet pour *self*. Le corps de la fonction se charge ensuite d'initialiser les variables de l'objet, ou *attributs*.

```
...     self.title = title
...     if authors is None:
...         self.authors = []
...     else:
...         self.authors = authors
...
...     def __init__(self, text=u'contenu_page'):
...         self.text = text
...         self.notes = []
...
...     my_book = Book(u'Python', [u'Tarek Ziadé'])
>>> page_1 = Page(u'Introduction', 'et blablabla')
>>> my_book.pages[1] = page_1
>>> my_book.pages[1].text
u'Introduction', et blablabla'
>>> my_book.title
u'Python'
```

Plusieurs remarques concernant cette portion de code :

- Les classes sont des sous-types du type *object*, qui offrent un ensemble de primitives utilisées pour des fonctionnalités avancées comme les *slots*, les *descriptors* ou encore les *properties*.
- La relation entre le livre et ses pages est définie par un dictionnaire, dont les clefs sont les numéros de page.
- *self* fait référence à l'instance de l'objet.
- Les variables sont initialisées dans la fonction *__init__* et leurs valeurs sont de fait spécifiques à chaque instance d'objet.

Le découpage des données d'un programme dans des définitions de classe est similaire au principe des enregistrements dans les langages impératifs comme le C ou le Pascal : il offre un moyen simple de structurer son programme pour la manipulation des données et permet en outre de formaliser l'initialisation de ces données.

L'objet comme structure de code

Il est possible d'ajouter d'autres méthodes à la définition d'une classe, pour regrouper tout le code qui manipule les données. Ces méthodes constituent les *API* utilisées par le code appelant et forment une couche au-dessus des données. Dans l'exemple du livre, il s'agit d'ajouter des méthodes de lecture et d'écriture des données¹ :

Cet accès indirect permet de masquer aux utilisateurs l'implémentation interne des classes, ce qui offre une protection en termes d'impact lorsque le code change : les attributs n'étant plus explicitement appelés, ils peuvent être modifiés librement sans impacter le reste du programme.

```
def __init__(self, text=u'content\u2014page'):
    self._text = text
    self._notes = []
def getContent(self):
    return (self._text +
            '\n'.join(self._notes))
```

Les attributs privés Par convention, les attributs internes sont préfixés d'un espace souligné. Lorsqu'ils sont préfixés de deux espaces soulignés, ils deviennent *privés* et inaccessibles :

```
>>> class Book(object):
...     def __init__(self, title, authors=None):
...         self._pages = {}
...         self._title = title
```

```
>>> l = Affaire()  
>>> l._luis-trio  
Traceback (most recent call last):  
...  
,Affaire' object has no attribute
```

...
else:

```
...  
self.-authors = authors  
def setPage(self mm text):
```

```
...  
def getPage(self, num):  
    return self._pages[num].getContent()  
    ...
```

```
...  
...  
def getPages(self):  
    for key, page in self._pages.items():
```

```
...  
def yield key, page.getContent()  
    getTitle(self):
```

```
...  
    def generate(self):  
        ...  
        return self._title
```

¹ Aussi appelées *setters* et *getters*.

```
>>> my_book = Book(u'Python', [u'Tarek_Ziade'])
>>> my_book.setPage(1,
...     u'Introduction', u't_blablabla')
>>> my_book.getPage(1)
u'Introduction', u't_blablabla'
>>> my_book.getTitle()
u'Python'
```

Cette protection est purement conventionnelle, puisqu'il est possible de retrouver la valeur en insistant un peu :

```
>>> l. -Affaire_-luis-trio
u 'chic_planete ,
```

Python à
règles rigides

Les *properties* Python propose également un système de propriété pour simplifier l'écriture des accesseurs simples :

```
>>> class Book2(object):
...     def __init__(self):
...         self._title = u'titre'
...     def _getTitle(self):
...         return self._title
...     def _setTitle(self, value):
...         self._title = value.title()
...     title = property(fget=_getTitle,
...                      fset=_setTitle,
...                      doc='Title')
```

```
>>> le = Book2()
>>> le.title
u'titre'
>>> le.title = u'substance-mort'
>>> le.title
u'Substance-Mort'
```

Cette écriture offre un moyen souple et transparent de mettre en place des contrôles sur les modifications des attributs d'une classe, et est équivalente en C# ou Delphi.

Objectif modularité Les méthodes et *properties* permettent d'assurer la *modularité* d'une classe, c'est-à-dire son autonomie vis-à-vis du reste de l'application. Dans l'exemple du livre, la deuxième version n'engendre que des dépendances simples, contrairement à la première version :

- Le code client n'interagit qu'avec les *API* de la classe *Book* et ignore l'existence de la classe *Page* ainsi que le fonctionnement interne.
- La classe *Page* se contente de gérer son texte et ses notes et ne connaît pas la classe *Book*.

La difficulté de la création de classes modulaires consiste à trouver le bon niveau de découpage :

- des classes trop volumineuses complexifient leur évolution ;
- un nombre trop élevé de classes affaiblit le *code glue* du programme, c'est-à-dire l'ensemble du code en charge de gérer les

interactions entre les classes : il devient plus sensible aux modifications.

Des modèles plus complexes ont été mis au point pour répondre à cette problématique et simplifier l'évolution du code, et sont détaillés dans la prochaine section.

Coder objet en Python revient donc à organiser le code et les données en classes, en se focalisant sur la modularité de chacune d'entre elles. Ce découpage est garant de l'évolutivité et de la simplicité du programme.

L'objet, élément réutilisable

La capacité d'un objet à masquer une partie de son implémentation au code appelant offre la possibilité de rendre les éléments de l'application réutilisables. La classe *Livre* implémentée dans la section précédente peut être reprise dans un autre programme sans aucune difficulté. On parle alors de *componentization* et toute la subtilité du métier de designer est de projeter les classes imaginées pour une application dans des environnements différents afin de s'assurer qu'elles ne seront pas jetées aux orties pour les développements suivants. Là où les développeurs pratiquaient encore il y a quelques années du *code one-shot*, c'est-à-dire répondant à un cas concret d'utilisation sans aucune reflexion sur sa réutilisation possible, la POO a offert un modèle qui facilite la capitalisation du code, grâce à des mécanismes d'évolution de code que sont l'**héritage** et le **polymorphisme**. Enfin, un modèle spécifique d'architecture objet a émergé ces dernières années pour réussir à maîtriser plus facilement l'évolution de gros logiciels : **la programmation orienté composants**, qui offre une disposition nette entre l'implémentation et les relations entre les classes d'une application.

Héritage Un livre est composé de pages, mais peut également contenir un index qui regroupe les termes notables du livre. Pour simplifier l'exemple, l'index est composé de la liste des mots du livre, avec pour chacun la référence des pages dans lesquelles il apparaît. Pour réaliser cette extension il suffit d'ajouter une méthode *_createIndex* en charge de collecter les mots dans un nouvel attribut, puis une méthode

getIndex et une propriété *index* qui publie l’index. *getIndex* peut également prendre en charge le déclenchement d’une réindexation lorsqu’une page a été ajoutée au livre. Cette mécanique d’indexation est spécifique à un projet et le souhait est de conserver la version de base.

L’héritage résout ce besoin, en permettant à une nouvelle classe d’hériter de l’ensemble des fonctionnalités d’une classe existante, afin de l’étendre. Dans notre exemple, une classe *IndexableBook* est créée, dérivée de *Book* :

```
>>> class IndexableBook(Book):
...     def __init__(self, title, authors=None):
...         self._pages = {}
...         self._title = title
...         if authors is None:
...             self._authors = []
...         else:
...             self._authors = authors
...         self._index = {}
...         self._index_upToDate = True
...
...     def getIndex(self):
...         if not self._index_upToDate:
...             self._createIndex()
...         index = self._index.items()
...         index.sort()
...         return index
...
...     index = property(fget=getIndex)
...
...     def _createIndex(self):
...         self._index = {}
...
...         for (page_num, content) in
...             self._getPages():
...             for word in content.split():
...                 word = word.strip().lower()
...                 if word not in self._index:
...                     self._index[word] = \
...                         (page_num,)
...                 else:
...                     page_nums = \
...                         self._index[word]
```

Cette classe peut ensuite être utilisée en lieu et place d’une classe *Book*, car c’est une classe *Book*, augmentée d’une nouvelle fonctionnalité, l’exemple précédent continuera à fonctionner :

```
>>> my_book = IndexableBook(u'Python',
...                           [u'Tarek Ziadé'])
>>> my_book.setPage(1,
...                   u'Introduction, et blablabla')
>>> my_book.getPage(1)
u'Introduction, et blablabla'
>>> my_book.getTitle()
u'Python'
>>> my_book.index
[((u'blablabla', (1,)), (u'et', (1,)),
  (u'introduction', (1,))),
 ((u'et', (1,)), (u'et', (1,)),
  (u'et', (1,)),
  (u'hop', (2,)), (u'introduction', (1,)))]
```

Ces ajouts par couches successives d’héritage permettent d’isoler des fonctionnalités et de faciliter leur évolution.

Polymorphisme Le problème de l’exemple précédent concerne les méthodes *__init__* et *setPage*, qu’il a fallu réécrire pour injecter des spécificités : une bonne partie de ce code est similaire à la classe parent et se retrouve dupliquée. En cas d’évolution de *Book*, *IndexableBook* peut devenir obsolète. Le *polymorphisme* résout ce problème en permettant à une classe de surcharger des méthodes et d’invo-

quer les méthodes de la classe parent. Le type *super* est utilisé pour remonter les dérivations :

```
>>> class A(object):
...     def meth(self):
...         print 'A:meth'
...
>>> class B(A):
...     def meth(self):
...         print 'B:meth'
...
>>> class C(B):
...     def meth(self):
...         print 'C:meth'
...
>>> class D(C):
...     def meth(self):
...         print 'D:meth'
...     super(D, self).meth()
...     super(C, D).meth(self)
...     super(B, D).meth(self)
...
>>> d = D()
>>> d.meth()
D:meth
C:meth
B:meth
A:meth

Le premier appel, de type super(class, instance), renvoie un objet qui donne accès aux méthodes de la classe juste au-dessus. Les appels suivants, de type super(class, subclass), renvoient un objet qui donne accès aux méthodes de la classe donné, sous leurs formes détachées (unbound). self doit dans ce cas être passé en premier paramètre.
```

Ce principe, appliqué à *IndexableBook*, donne :

```
>>> class IndexableBook(Book):
...     def __init__(self, title, authors=None):
...         parent = super(IndexableBook, self)
...         parent.__init__(title, authors)
...         self._index = {}

def setPage(self, num, text):
    parent = super(IndexableBook, self)
    parent.setPage(num, text)
    self._index_upToDate = False
    my_book = IndexableBook(u'Python',
                           [u'Tarek Ziadé'])
    my_book.setPage(1,
                   u'Introduction', 'et blablabla')
    my_book.getPage(1)
    u'Introduction', 'et blablabla'

self._index = {}
```

Les métaclasses Tout est objet en Python, et les définitions de classe ne dérogent pas à la règle. Elles possèdent donc aussi, comme la classe pour ses objets, un patron qui permet de les instancier et de faire varier leur fonctionnement. Ce patron est une métaclassse. Les métaclasses sont définies par le biais de la variable spéciale `_metaclass_`, à ajouter à la définition d'une classe. Cet attribut doit être une fonction de la forme `def function(cls, bases, dict)`, appelée lorsque Python lit la définition. `cls` est le nom de la classe, `bases` la liste des classes de base, et `dict` un dictionnaire des attributs et leurs valeurs. Ces paramètres sont ceux utilisés pour appeler la fonction `type`.

Les métaclasses Tout est objet en Python, et les définitions de classe ne dérogent pas à la règle. Elles possèdent donc aussi, comme la classe pour ses objets, un patron qui permet de les instancier et de faire varier leur fonctionnement. Ce patron est une métaclassse. Les métaclasses sont définies par le biais de la variable spéciale `_metaclass_`, à ajouter à la définition d'une classe. Cet attribut doit être une fonction de la forme `def function(cls, bases, dict)`, appelée lorsque Python lit la définition. `cls` est le nom de la classe, `bases` la liste des classes de base, et `dict` un dictionnaire des attributs et leurs valeurs. Ces paramètres sont ceux utilisés pour appeler la fonction `type`.

odes de la classe à la volée.

1

L'héritage et il faut actualiser le code d'une méthode.

te polymorphe, mais ne sont plus e.

us garants de l'

aspect fonctionnel

Programmation orientée composants Dans les gros programmes³, la maintenance et la compréhension des arborescences de dérivation peuvent devenir très complexes à gérer. Le code qui manipule des objets devient sensible aux types et doit souvent tester l'appartenance d'un objet à un type : les appels à *instance* et autres *issubclass* deviennent légion.

Pour solutionner ce problème, la programmation orientée composants (POC) offre une simplification du modèle en désolidarisant la *signature*⁴ de l'implémentation. La signature est appelée *interface* et devient un élément de programmation à part entière. Les objets manipulés sont de ce fait vus à travers les interfaces qu'ils implémentent.

Programmation orientée composants Dans les gros programmes³, la maintenance et la compréhension des arborescences de dérivation peuvent devenir très complexes à gérer. Le code qui manipule des objets devient sensible aux types et doit souvent tester l'appartenance d'un objet à un type : les appels à *instance* et autres *issubclass* deviennent légion.

Pour solutionner ce problème, la programmation orientée composants (POC) offre une simplification du modèle en désolidarisant la *signature*⁴ de l'implémentation. La signature est appelée *interface* et devient un élément de programmation à part entière. Les objets manipulés sont de ce fait vus à travers les interfaces qu'ils implémentent.

```
>>> class LesLubies(object):
...     __metaclass__ = capitalizer
...     def deLili(self):
...         print "c'est-'youpi"
...
allons-y
>>> mais = LesLubies()
>>> mais.deLili()
Traceback (most recent call last):
...
'LesLubies' object has no attribute 'deLili'
>>> mais.Delili()
c'est-'youpi
```

• une interface
on de classe dé-
es ne contiennent
IPage permet de
IBook celui d'u
classiques, le par-
om zope. int
lass IPage(I
def get ()
''' ge
def set (c
''' se

Dans *zope.interface*, une classe est vivée de la manière suivante : aucun code. Dès lors, il suffit de définir le fonctionnement d'un objet dans un livre. Contrairement à ce que l'on peut penser, l'attribut *self* est accessible à l'extérieur de l'interface.

ans l'exemple ci-dessous, nous démontrons comment d'une manière très simple et directement à des fins pratiques, nous pouvons nous servir de la méthode de bas *Interface*, une intégration de l'ensemble des fonctions d'un programme.

surface est une face, dont les dessous, l'impage, et l'interface : rédefinitions de

Dans le même esprit, la méthode spéciale *-new-* permet d'effectuer des opérations avant que l'objet ne soit instancié, mais récupère une classe déjà instancee.

³Plusieurs dizaines de milliers de lignes.

41 *ensemble des méthodes*

L'ensemble des noms des méthodes

<http://www.zope.org/Products/ZopeInterface> von Eli Zaitsev example.

```

...
def get(page_number):
    ...
    """get_the_page_content"""
def set(page_number, content):
    """
    set_the_page_content
    """
def getPageCount():
    ...
    """returns_the_number_of_pages"""

Utiliser une interface Ces définitions peuvent ensuite être utilisées dans les classes qui souhaitent respecter le contrat défini, via la fonction implements.
>>> from zope.interface import implements
>>> class Page(object):
    implements(IPage)
    def __init__(self, content=''):
        self.set(content)
    def set(self, content):
        self.content = content
    def get(self):
        ...
        return content
    ...

Pour faciliter leur utilisation, les frameworks utilisant les interfaces fournissent en général des usines en charge de générer des instances d'objets qui implémentent une interface donnée. Cette fonctionnalité permet au code utilisateur de ne manipuler que des interfaces, sans avoir à connaître les classes qui les implémentent, sur le modèle de Java. Dans l'exemple ci-dessous, la fonction register se charge d'associer la classe à l'interface par le biais de directlyProvides, qui est équivalent à implements, puis de mettre à jour une liste pour savoir quelle classe doit être instanciée. Enfin, la fonction getInstance renvoie une instance correspondant à l'interface demandée.
>>> from zope.interface import implements
>>> from zope.interface import directlyProvides
>>> registered = {}
>>> def register(interface, class_):
    directlyProvides(interface, class_)
    ...
    global registered

```

```

    ...
    registered[interface] = class_
    ...
    >>> def getInstance(interface, *args, **kw):
    ...
        return registered[interface](*args, **kw)
    ...
    >>> class Page(object):
        def __new__(cls, name, bases, dct):
            register(IPage, cls)
            return type.__new__(cls, name,
                               bases, dct)
        ...
        def __init__(self, content=''):
            self.set(content)
        def set(self, content):
            self.content = content
        def get(self):
            ...
            return content
        ...
    >>> class Book(object):
        def __new__(cls, name, bases, dct):
            register(IBook, cls)
            return type.__new__(cls, name,
                               bases, dct)
        ...
        def __init__(self):
            ...
            self.pages = []
        def set(self, page_number, content):
            self.pages[page_number] = \
                getInstance(IPage)(content)
        def get(self, page_number):
            ...
            return self.pages[page_number]
        ...

Dans ce cas, la classe Book ne manipule pas directement la classe Page mais l'interface IPage. Cet exemple d'usine décrit bien le mécanisme qui permet de décorer les classes par le biais des interfaces (appelé couplage lâche), mais reste toutefois incomplet, car :
    - il ne gère pas les problématiques de priorité lorsque plusieurs classes implémentent la même interface ;
    - il suppose une connaissance des arguments du constructeur de la classe, qui ne sont pas toujours définis dans l'interface.
```

Utilisation concrète des interfaces À moins de créer son propre framework, l'utilisation des interfaces est fortement liée aux bibliothèques utilisées, et ces dernières fournissent des mécanismes complets pour mettre en œuvre des *couplages lâches*. Zope 3 par exemple, propose un système d'association basé sur des descriptions XML⁷, ou des couples classe-interface sont formés pour un contexte d'exécution donné⁸.

5.2.2 Design patterns communs

Les *design patterns* (DP) fournissent aux développeurs une bibliothèque de structures d'objets réutilisables, qui répondent chacune à un besoin précis. Ces patterns portent un nom unique, qui est entré dans le vocabulaire technique des architectes logiciel, et qui permet aux développeurs de partager une culture objet commune. Les patterns les plus communs sont :

- Proxy
- Adapter
- Singleton
- Visitor
- Observer

Proxy

Proxy définit une classe en charge de fournir un service obtenu en invoquant une autre classe. Cette encapsulation permet de publier les fonctionnalités d'une classe de manière indirecte dans les cas suivants :

- l'objet manipulé est sur une ressource distante (Remote Proxy) ;
- l'objet manipulé n'est instantié que lorsque c'est nécessaire (Virtual Proxy) ;
- l'accès à l'objet manipulé est sécurisé (Security Proxy) ;
- les résultats, contenus à calculer, sont mis en cache par le proxy (Cache Proxy).

Il n'y a pas de modèle-type pour ce pattern, qui peut être codé de différentes manières en fonction du contexte, mais le principe est toujours le même : une instance de proxy est associée à une classe ou une

instance de classe. Dans l'exemple ci-dessous, le proxy *protect*, défini comme un decorator, permet de sécuriser l'accès à des méthodes. Il vérifie que l'utilisateur défini dans les variables globales possède le rôle nécessaire, sans quoi une erreur est levée⁹.

```
>>> class User(object):
...     def __init__(self, roles):
...         self.roles = roles
...
>>> tarek = User(( 'admin', 'user' ))
>>> bill = User(( 'user', ))
>>> def protect(role):
...     def _protect(function):
...         def wrap(*args, **kw):
...             user = globals()['user']
...             if role not in user.roles:
...                 msg = ( 'Hahaha! '
...                         ', Jamais je le dirai' )
...                 raise Exception(msg)
...             return function(*args, **kw)
...         return wrap
...     return _protect
...
>>> class MesSecrets(object):
...     @protect('admin')
...     def crepesDelicieuses(self):
...         print 'mettre du beurre dans la pate'
...
>>> secrets = MesSecrets()
>>> user = tarek
>>> secrets.crepesDelicieuses()
mettre du beurre dans la pate
>>> user = bill
>>> secrets.crepesDelicieuses()
Traceback (most recent call last):
...
Exception: Hahaha ! Jamais je le dirai
```

⁷Appelées ZCML.
⁸Voir <http://wiki.zope.org/zope3/FrontPage>.

⁹C'est ce modèle qui est utilisé dans Zope.

Le module *xmlrpclib* implémente également un Proxy, de type remote. Il fournit un objet instancié localement, qui accède à des ressources distantes. Dans l'exemple ci-dessous, un proxy est lié à la méthode rpc *weblogUpdates.ping* du site *ping-o-matic*, qui permet de prévenir les annuaires de blogs lors de la mise à jour d'un blog :

```
>>> from xmlrpclib import ServerProxy
>>> service = 'http://rpc.pingomatic.com'
>>> server = ServerProxy(service)
>>> site = 'http://programmation-python.org'
>>> server.weblogUpdates.ping(site)
{'message': 'Pings forwarded to 16 services!', 'ferror': False}
```

Les appels effectués sur ServerProxy sont envoyés au serveur distant, via *__getattr__*.

Adapter

Le pattern Adapter, ou *Wrapper*, permet d'adapter une classe pour son utilisation dans un contexte pour lequel elle n'a pas été initialement prévue. Cette couche offre une compatibilité au niveau des signatures des méthodes, et effectue si nécessaire les conversions des données qui transsident. Le module *cStringIO* est un bon exemple d'*adapter* : il implémente dans la classe *cStringIO* les méthodes du type *file* mais gère en interne une chaîne de caractères. Il adapte donc le type *string* vers le type *file* et permet d'utiliser *cStringIO* là où un fichier est attendu. Dans l'exemple ci-dessous un flux XML est écrit dans un objet de type *cStringIO*, via *write*, initialement conçu pour le type *file* :

```
>>> from lxml import etree
>>> html = etree.Element('html')
>>> body = etree.SubElement(html, 'body')
>>> from cStringIO import StringIO
>>> res = StringIO()
>>> etree.ElementTree(html).write(res)
>>> res.seek(0)
>>> res.read()
'<html><body/></html>'
```

La technique la plus simple pour écrire des adapters, est de fournir une classe qui prend en paramètre de son constructeur une instance de la classe à adapter.

```
>>> class Doc(object):
...     def __init__(self, ob):
...         self.ob = ob
...     def __str__(self):
...         doc = [',Attributes:', '%s', '%s attribute']
...         doc += ['%' * 100, '%s', '%s attribute']
...         for attribute in dir(self.ob):
...             if (not attribute.startswith('_')):
...                 attribute.startswith(' '))
...         return '\n'.join(doc)
...
...     class MaClasse(object):
...         e = 1
...
...     def __init__(self):
...         j = MaClasse()
...         infos = Doc(j)
...     def print infos
...     Attributs:
...         e
```

La classe *Doc* adapte tout type d'objet pour fournir une interface d'affichage d'informations sur les attributs.

Singleton

Singleton permet de s'assurer qu'une seule instance d'une classe donnée puisse exister à un instant T. Ce mécanisme est utilisé lorsque le code exécuté par la classe doit être coordonné à l'échelle de l'application. C'est le cas par exemple des connecteurs vers les bases de données, ou encore des utilitaires qui fournissent des services qui appellent des ressources externes, comme des générateurs de rapport PDF, ou des annuaires.

La technique la plus élégante pour écrire ce pattern a été introduite par Alex Martelli dans le Cookbook [10] et se nomme *Borg*. Elle part du principe que ce n'est pas l'unicité de l'instance qui compte pour assurer un état unique pour la classe, mais l'état des attributs. Or Python permet de faire partager très simplement les attributs

entre les instances d'une même classe, sans en faire pour autant des attributs statiques.

```
>>> class Borg(object):  
    channel = 'channel'
```

```
def __new__(cls, *args, **kw):
    """shares the state"""
    parent = super(Borg, cls)
    ob = parent.__new__(cls, *args, **kw)
    ob.__dict__ = cls.__state
    return ob

def __init__(self, name, previous):
    self.name = name
    if previous is not None:
        previous.next = self
    self.next = None

def __str__(self):
    ...
```

```
>>> class PrintNode(Visitor):
...     def __call__(self, visitable):
...         chain = [str(visitable)]
...         while visitable.next is not None:
...             chain.append(str(visitable.next))
...             visitable = visitable.next
...
...         print(" ".join(chain))
...
...         if visitable == self.visitable:
...             return
...
...         self.visitable = visitable
...         self.visit(visitable)
...
...     def visit(self, visitable):
...         if visitable is None:
...             return
...
...         if visitable.is_leaf():
...             print(visitable.value)
...         else:
...             self.visit(visitable.left)
...             self.visit(visitable.right)
...
...     def __str__(self):
...         return "PrintNode"
...
...     def __repr__(self):
...         return "PrintNode()"

>>> un = Borg()
>>> deux = Borg()
>>> un.truc = 1
>>> deux.truc
1
>>> un is deux
False
```

La méthode `new` est utilisée afin d'éviter qu'une éventuelle surcharge du constructeur dans une classe dérivée ne bloque le code, si le constructeur de base n'est pas appelé. Ce modèle a quelques limites, dans le cas où des techniques qui court-circuitent `dict` sont mises en oeuvre, comme `__getattribute__`.

Visitor

Le pattern Visitor permet de séparer un traitement particulier d'une classe sur laquelle il s'opère. Cette technique autorise l'extension des fonctionnalités liées à des classes d'objets sans avoir à modifier leur signature ou à les dériver.

```
>>> class Visitable(object):
...     def accept(self, visitor):
...         visitor(self)
...
... >>> class Visitor(object):
...     def __call__(self, visitable):
...         raise NotImplementedError
```

Visitor est souvent utilisé pour l'affichage de structures de données par les programmes de *prettyprint* qui parcourront l'arborescence d'un code source pour le présenter formaté. HTML Tidy¹⁰ permet par exemple d'ordonner et reformater des contenus SGML.

Observer

Observer permet de mettre en place une surveillance lâche de l'état d'un objet donné. Le principe est basé sur le déclenchement d'événements : des *observateurs* souscrivent à la surveillance d'un

<http://tidy.sourceforge.net/>

événement donné qui est déclenché par code dans les objets observés.

On parle alors de programmation événementielle. Ce principe est très souvent utilisé dans les systèmes d'interfaces, pour :

- provoquer des appels de fonctions à chaque événement provoqué par l'utilisateur, comme un click souris ou une frappe de clavier;
- gérer la mise à jour de l'interface lorsque les données sont modifiées.

La technique la plus souple en Python pour implémenter Observer est de créer une classe *Event*, qui représente l'événement, et de gérer une liste d'observateurs associés à cette classe d'événement. Une instance de la classe étant générée et passée aux observateurs à chaque déclenchement :

```
>>> class Event(object):
...     _observers = []
...
...     @classmethod
...     def register(cls, observer):
...         if observer not in cls._observers:
...             cls._observers.append(observer)
...
...     @classmethod
...     def unregister(cls, observer):
...         if observer in cls._observers:
...             cls._observers.remove(observer)
...
...     @classmethod
...     def notify(self, event):
...         print 'je confirme'
...
...         >>> AieEvent.register(Robert())
...         >>> AieEvent.register(Joe())
...         lami = Marcel()
...         >>> lami.surSonPied()
...
arg
Il s'agit de marcher sur son pied
Ils ont marché sur son pied
je confirme
```

Ce fonctionnement peut être enrichi en :

- ajoutant des informations contextuelles supplémentaires dans l'instance de l'événement;
- mettant en place un ordonnancement des observers, avec possibilité d'arrêter la boucle d'appel en gérant une valeur de retour sur *notify*.

Les patterns présentés ou certaines de leurs variations sont les plus fréquemment utilisés en Python, mais dépendent du style de programmation des développeurs. Il en existe des centaines de différents, qu'il serait impossible et inutile de lister de manière exhaustive ici. L'essentiel reste de systématiser le principe de généralisation lorsque le code est écrit : dès qu'une interaction entre plusieurs classes se reproduit, un pattern peut être extrait. Pour approfondir le sujet, les ouvrages du GoF¹¹ restent la référence en la matière [5].

Exemple d'utilisation :

```
>>> class AieEvent(Event):
...     pass
```

```
...
>>> class Marcel(object):
...     def surSonPied(self):
...         try:
...             print 'Ils ont marché sur son pied'
...         finally:
...             AieEvent.notify(self)
```

¹¹Gang of Four.

5.3 Du bon usage de Python 2.5

Python est comparable à du bon vin : au fur et à mesure de son évolution, sa syntaxe et sa bibliothèque standard évoluent, pour se bonifier et faciliter l'écriture de programmes. Ce dynamisme est possible grâce au modèle de développement communautaire du langage : une bande de joyeux développeurs, formant un noyau autour du BDFL¹², propose des évolutions en écrivant des PEP¹³. Ces mini-cahiers des charges sont soumis à l'approbation dans un premier temps de l'ensemble des développeurs de Python, pour être affinés ou rejetés, puis sont acceptés ou non par le BDFL et prévus dans le calendrier pour réalisation dans une version future. L'intérêt de ce modèle est la vitesse avec laquelle le langage évolue, comparé à d'autres qui subissent une inertie plus forte, due à des considérations économiques et stratégiques induites par les entreprises qui les soutiennent. Le défaut majeur de Python est lié à cette qualité : il est difficile, comme pour Linux il y a quelques années, d'en faire un standard industriel, terrain de jeu cloisonné par les grandes entreprises informatiques à coup de millions de dollars.

Pour nous autres développeurs, ce modèle prend tout son sens lorsque l'on écrit du code : l'expressivité de Python est un avantage réel face à Java ou aux déclinaisons de C. Elle s'exprime à travers un ensemble de **code patterns** et de **bonnes pratiques** qu'il convient de connaître et maîtriser pour une productivité et un plaisir maximum. La version 2.5 de Python, sortie un peu plus d'un an après la 2.4, a inclus beaucoup d'améliorations syntaxiques et exprime pleinement cet avantage.

5.3.1 Code patterns communs

La distinction entre un *code pattern* et un *design pattern* est relative à son niveau d'application dans le code : un *code pattern* permet de modéliser un style de programmation pour les petites séquences de code qui ne dépassent pas quelques lignes, et qui répondent à un besoin précis et récurrent. Ces patterns sont sans cesse répétés et constituent le *tissu* des programmes. Cette section présente une liste de ces patterns, organisés en besoins :

- Les *list comprehension*, c'est bon.
- Intérêt et usage des generators et iterators.
- La chasse au point.
- Memoize.
- *try, except* et *finally*.
- Quand et comment utiliser *with* ?

Les *list comprehension*, c'est bon

Les *list comprehension*¹⁴ permettent d'exprimer de façon concise des traitements sur des séquences. Si chaque élément d'une séquence de chaînes doit être traité (pour transformer toutes les majuscules en minuscules), la première écriture qui vient à l'esprit d'un développeur issu d'un autre langage est :

```
>>> phrase = ['Children', 'of', 'Men', 'est',
...             'un', 'bon', 'livre']
>>> i = 0
>>> for mot in phrase:
...     phrase[i] = mot.lower()
...     i += 1
...
>>> phrase
['children', 'of', 'men', 'est', 'un', 'bon',
 'livre']
```

Ou une variation équivalente. Les *list comprehension* proposent :

```
>>> phrase = ['Children', 'of', 'Men', 'est',
...             'un', 'bon', 'livre']
...
>>> phrase = [mot.lower() for mot in phrase]
>>> phrase
['children', 'of', 'men', 'est', 'un', 'bon',
 'livre']
```

C'est élégant et concis, plaisant à écrire. Des conditions peuvent être ajoutées :

```
>>> phrase = ['Children', 'of', 'Men', 'est',
...             'un', 'bon', 'livre']
```

¹²Le *Benevolent Dictator For Life*, Guido van Rossum.

¹³*Python Enhancement Proposal*.

```
...
    'un', 'bon', 'livre']
>>> [mot.lower() for mot in phrase]
... if mot not in ('est', 'un')]
['children', 'of', 'men', 'bon', 'livre']
```

On nage en plein bonheur. Les traitements peuvent bien évidemment être déportés dans une fonction :

```
>>> def _accentDijonnais(mot):
...     return mot.replace('r', 'Rrr')
...
>>> phrase = ['Children', 'of', 'Men', 'est',
...             'un', 'bon', 'livre']
>>> [-accentDijonnais(mot) for mot in phrase
...     if mot not in ('est', 'un')]
['ChildRrren', 'of', 'Men', 'bon', 'livrRrre']
```

Cette écriture est en outre exécutée plus rapidement par l'interpréteur, qui reconnaît le bon goût du développeur et le gratifie de quelques cycles CPU. L'exemple ci-dessous compare la vitesse d'exécution en *PyStones* de deux versions différentes du code. Le decorator *timedtest* est disponible dans la section *scripts* du site <http://programming-python.org>¹⁵.

```
>>> nombres = [i for i in range(50000)]
>>> def _calcul(nombre):
...     return nombre + 1
...
>>> @decorators.timedtest
... def super_chouette():
...     return [_calcul(nombre)
...         for nombre in nombres]
>>> @decorators.timedtest
... def trop_pas_beau():
...     i = 0
...
...     for nombre in nombres:
...         nombres[i] = _calcul(nombre)
...         i += 1
...
...     return phrase
```

map et filter Avant l'avènement des *list comprehension*, Python proposait deux fonctions pour effectuer des manipulations sur les séquences, à savoir *map* pour des modifications de chaque élément et *filter* pour filtrer les éléments par le biais d'une fonction booléenne :

```
>>> def youpla_ficator(phrase):
...     return '%s,%syoupla_boun,%s phrase'
>>> phrases = ['Bonjour', 'au_revoir']
>>> map(youpla_ficator, phrases)
['Bonjour, youpla_boun', 'au_revoir, youpla_boun']
```

```
>>> def positif(entier):
```

```
...     return entier >= 0
>>> entiers = [1, -4, 7, 8, -2, 3, -8]
>>> filter(positif, entiers)
[1, 7, 8, 3]
```

Ces fonctions sont vouées à disparaître et ne doivent pas être utilisées¹⁶.

Intérêt et usage des generators et iterators

Les generators, introduits à la version 2.2, offrent une technique de programmation originale qui permet à une fonction ou une méthode de renvoyer des résultats intermédiaires. Le mot-clé *return* est remplacé par *yield*. La première fois que ce mot est rencontré, l'exécution est stoppée et un objet de type *generator* est renvoyé, qui contient le contexte local de la fonction et une méthode *next*. À chaque appel de cette méthode, la fonction est appelée jusqu'à au prochain *yield* :

```
>>> def hospitalier():
...     yield 'Bonjour'
...     yield 'Re-Bonjour'
...     yield 'Encore_vous_!_Fabuleux'
```

¹⁵ *faster* renvoie True si le delta de temps entre le premier test mesuré et le deuxième est supérieur au nombre de PyStones fournis en paramètres.

¹⁶ Le temps du plaidoyer qu'il fallait encore faire il y a quelques années pour convaincre les inconditionnels de *map* et *filter* est heureusement révolu.

```

...
>>> gen = hospitalier()
>>> gen.next()
'Bonjour'
>>> gen.next()
'Re-Bonjour'
>>> gen.next()
'Encore..vous..!..Fabuleux'
>>> gen.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

```

next et le déclenchement de *StopIteration* rendent les generators compatibles avec les boucles :

```

>>> gen = hospitalier()
>>> for element in gen:
    print element
...
Bonjour
Re-Bonjour
Encore vous ! Fabuleux

```

Ce principe est appliqué lorsqu'une fonction génère une série de valeurs, et que le code appellant traite un à un les éléments de cette série, sans avoir à récupérer l'intégralité de la série. De nombreuses optimisations peuvent être imaginées grâce à cette technique, comme :

- le traitement d'un flux continu de données;
- un feedback constant sur le traitement d'une séquence, en évitant ainsi l'ajout d'un système d'événements ou de *hooks*;
- une accélération du châmage de traitements, sans avoir à utiliser un système basé sur des piles et des threads.

L'exemple ci-dessous implémente la suite de Fibonacci (suite infinie)¹⁷ :

```

>>> def fibonnaci():
    a, b = 0, 1
    while True:
        ...

```

¹⁷Exemple tiré du PEP à l'origine de cette fonctionnalité.

generator expressions Une écriture encore plus concise pour les generators simples est possible, sur le même modèle que les *list comprehension* : les *generator expressions*¹⁸. Les crochets sont remplacés par des parenthèses, et permettent d'utiliser une séquence dont chaque élément sera transmis comme un *yield* l'aurait fait :

```

>>> gen = (word for word in ['qui', 'roule',
...                           'boule', "n'amasse",
...                           'pas', 'mousse'])
>>> gen.next()
'qui'
>>> gen.next()
'roule'
>>> gen.next()
'boule'

```

itertools Le module *itertools* offre des fonctions d'aide à la création de generators, dont les plus utiles sont :

- **cycle(seq)** : renvoie un à un les éléments d'une séquence seq, et revient au début lorsque la fin est atteinte
- **izip(seq1, seq2, ...)** : renvoie (*seq1[0]*, *seq2[0]*, ...), (*seq1[1]*, *seq2[1]*, ...)
- **ifilter(func, seq)** : renvoie les éléments de seq tels que *func*(*seq*) renvoie True
- **imap(func, seq1, seq2, ...)** : renvoie *func(seq1[0], seq2[0])*, ...
- **chain(seq1, seq2, ...)** : renvoie *seq1[0]*, *seq1[1]*, ..., *seq1[n]*, *seq2[0]*, *seq2[1]*, ..., *seq2[n]*

¹⁸*genexpr* pour les intimes.

- `takewhile(pred, seq)` : renvoie les éléments de `seq` tant que `pred(seq[n])` renvoie `True`
- `dropwhile(pred, seq)` : renvoie les éléments de `seq` dès que `pred(seq[n])` renvoie `False`

`izip`, `ifilter` et `imap` peuvent être considérés comme les remplaçants des primitives `zip`, `filter` et `map` :

```
>>> from itertools import izip, ifilter, imap
>>> seq1 = ('rouge', 'pomme', 'chien')
>>> seq2 = ('vert', 'banane', 'chat')
>>> seq3 = ('bleu', 'poire', 'poule')
>>> serie = izip(seq1, seq2, seq3)
>>> serie.next()
('rouge', 'vert', 'bleu')
```

```
>>> serie.next()
('pomme', 'banane', 'poire')
>>> serie.next()
('chien', 'chat', 'poule')
```

```
>>> def filter(word):
...     return 'o' in word
...
... 
```

```
>>> serie = ifilter(filter, seq1)
>>> serie.next()
'rouge'
>>> serie.next()
'pomme'
>>> serie.next()
'poire'
>>> serie.next()
'poule',
```

La chasse au point

Python est un langage interprété, et ne connaît donc pas à l'avance le code qu'il va exécuter, même s'il génère un *bytecode*¹⁹. Ce mode avantage à un inconvénient majeur : il n'est pas possible d'optimiser les appels de fonctions ou méthodes par un appel à une adresse mémoire unique²⁰. Dans l'exemple de code suivant, Python doit rendre visite successivement à *Pim* et *Pam* avant de pouvoir atteindre *Poum* :

```
>>> serie = imap(filter, seq3)
>>> serie.next()
'poire'
>>> serie.next()
'poule',
...
>>> def mapper(*elements):
...     return ','.join(elements)
...
>>> serie = imap(mapper, seq1, seq2, seq3)
>>> serie.next()
'rouge_et_vert_et_bleu',
>>> serie.next()
'pomme_et_banane_et_poire',
>>> serie.next()
```

'chien_et_chat_et_poule'

Les utilitaires d'`itertools` peuvent également être combinés pour former des séquences complexes :

```
>>> from itertools import chain
>>> serie = ifilter(filter,
...                   chain(seq1, seq2, seq3))
...
>>> serie.next()
'rouge'
>>> serie.next()
'pomme'
>>> serie.next()
'poire'
>>> serie.next()
'poule',
```

¹⁹Le bytecode n'est qu'une traduction du code en clair.

²⁰Les labels en assembleur.

```
>>> pim = Pim()  
>>> pim.pam().poum()  
on y est
```

Si *poum* est une méthode appelée de nombreuses fois dans une boucle de traitement, l'affecter à une variable locale évite à l'interpréteur d'aller la chercher à chaque appel, surtout si le chemin est coûteux :

Memoize

Lorsque certaines fonctions sont coûteuses et souvent appelées, leurs résultats en fonction des paramètres d'entrée peuvent être mis en cache pour une réutilisation ultérieure. Le plus simple consiste à créer un decorator en charge de tenir à jour un dictionnaire de résultats, dont les clefs représentent les arguments passés. Ce decorator est paramétré pour permettre de définir une durée de vie pour le cache :

```
>>> @decorators.timedtest
... def classique():
...     for i in range(5):
...         pim.pam().poum()
...
>>> @decorators.timedtest
... def racourci():
...     poum = pim.pam().poum
...     for i in range(5):
...         poum()
...
>>> classique()
on y est
...
>>> def memoize(duration=10):
...     def wrap(function):
...         def _wrap(*args, **kw):
...             key = getKey(args, kw)
...             if (key in cache and not
...                 obsolete(cache[key], duration)):
...                 print ('all the cache are',
...                       'belong_to_us')
...             return cache[key][0]
...         result = function(*args, **kw)
...         cache[key] = result, time.time()
...         return result
...     return _wrap
...
>>> decorators.faster(1000)
True
```

La chasse aux points dans les bocages est souvent davantage deur

L'utilisation de ce decorator est ensuite directe, comme en témoigne l'exemple ci-dessous, sur la fonction *factorial* :

```

...     def factorial(n):
...         import math
...         if ((not n >= 0) or (math.floor(n) != n)
...             or (n + 1 == n)):
...             raise ValueError('Calcul impossible')
...         result = 1
...         factor = 2
...         while factor <= n:
...             result *= factor
...             factor += 1
...         return result
...
...         >>> factorial(5); factorial(5)
120     all the cache are belong to us
120     all the cache are belong to us
120     time.sleep(1)
>>> factorial(5)
120

Cette technique peut être améliorée, en utilisant un serveur dédié au cache, comme memcached21, qui propose un bind Python.


```

Tri rapide

Pour effectuer un tri d'objets sur une caractéristique donnée (la clé), la technique la plus rapide est d'extraire cette clé, de trier la liste composée des *tuples* (*clef, objet*) avec *sort* puis de reformer cette séquence. Cet algorithme est basé sur la transformée Schwartziennne.

```

>>> class Peche(object):
...     def __init__(self, val):
...         self.noyau = val
...     def __str__(self):
...         return 'Peche(%d)' % self.noyau
...     def __cmp__(self, other):
...         return cmp(self.noyau, other.noyau)


```

try, except et finally

La gestion des exceptions en Python est comparable à celle pratiquée en C++ ou en Pascal : lorsqu'une erreur se produit, elle remonte toutes les couches de code comme une bulle d'air remonte à la surface de l'eau. Si elle atteint la surface sans être interceptée, le programme s'arrête et l'erreur est affichée :

```

>>> def _20_metres():
...     3 / 0
...
>>> def _10_metres():
...     -20.metres()
...
>>> def _surface():
...     _10_metres()


```

²¹<http://danga.com/memcached/>

```

...
>>> _surface()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 2, in _surface
      File "<stdin>", line 2, in _10_metres
        File "<stdin>", line 2, in _20_metres
ZeroDivisionError: integer division or modulo by zero

```

Le *traceback*, c'est-à-dire le message complet d'erreur affiché, présente l'ensemble des couches traversées entre la surface et l'erreur.

```

except Pour éviter un arrêt du programme, il est possible d'intercepter l'erreur lorsqu'elle remonte, grâce à la directive try...except :

>>> def _10_metres():
...     try:
...         _20_metres()
...     except ZeroDivisionError:
...         print 'Amiral, il y a eu un souci'
...
>>> _surface()
Amiral, il y a eu un souci

```

Cette interception peut être réalisée à tout niveau :

```

>>> def _10_metres():
...     _20_metres()
...
>>> def _surface():
...     try:
...         _10_metres()
...
except ZeroDivisionError:
    print 'Amiral, il y a eu un souci'
...

```

Le *try...except* intercepte l'erreur et empêche l'arrêt du programme. Cependant, si l'erreur n'est pas interceptée, une autre erreur sera générée.

```

...
>>> def avant_le_film():
...     try:
...         return tirage_loto()
...     except:
...         # traitements utiles
...         msg = ('Souci technique ...',
...                'A vous les studios')
...         raise Exception(msg)
...
...     except ZeroDivisionError:
...         print 'Mon dieu, appelez Thalassa'
...
>>> _surface()
Mon dieu, appelez Thalassa

```

except doit toujours être *typé*, c'est-à-dire spécialisé dans la capture d'un ou plusieurs types d'exception donnés, et ceci afin d'éviter

des erreurs silencieuses qui peuvent avoir des effets de bord désastreux :

```

>>> def tirage_loto():
...     joueurs = ['samantha', 'jenifer', 'kevin',
...               'leeroy']
...     return joueurs[8]

```

Lorsque les erreurs non typées sont interceptées, une erreur doit être redéclenchée :

```

>>> def avant_le_film():
...     try:
...         return tirage_loto()
...     except:
...         # traitements utiles
...         msg = ('Souci technique ...',
...                'A vous les studios')
...         raise Exception(msg)
...
...     except ZeroDivisionError:
...         print 'Mon dieu, appelez Thalassa'
...
>>> _surface()
Mon dieu, appelez Thalassa

```

La seule exception à cette règle concerne les programmes qui ne peuvent pas s'arrêter, comme les daemons.

finally *finally* permet de s'assurer qu'une portion de code particulière est bien exécutée, même si le programme provoque une erreur. Cette fonctionnalité permet de terminer proprement certaines opérations comme des écritures fichiers ou des échanges réseaux.

```
>>> data = [ 'youpi', 'dansons', 'la_Carioca' ]
>>> f = open( 'paroles.txt', 'w' )
>>> try:
...     for element in data:
...         f.write( "%s\n", % element )
...
... finally:
...     f.close()
```

finally est toujours visité, même si un *return* est placé dans le bloc protégé. Dans ce cas, *finally* est visité juste avant que la fonction ou la méthode rende la main.

```
>>> def suite():
...     try:
...         return 'Faissez_tous_comme_moi'
...     finally:
...         print "C'est_bien"
...
>>> suite()
C'est_bien
'Faissez tous comme moi'
```

Quand et comment utiliser *with*?

Une nouvelle directive a été intégrée au langage pour la version 2.6, et peu d'ores et déjà être utilisée sous 2.5 par le biais du module *future*. **with** remplace un code pattern communément adopté pour protéger une portion de code qui manipule un fichier ou un lock de thread.

L'écriture jusqu'alors était :

```
>>> fichier = open( 'ilela.txt', 'w' )
>>> try:
...     fichier.write( 'okay', 'okay' )
...
... finally:
```

```
...     fichier.close()
```

Ici, *finally* permet de s'assurer que le fichier est bien écrit. L'écriture équivalente avec **with** est :

```
>>> from __future__ import with_statement
>>> with open( 'ilela.txt', 'w' ) as fichier:
...     fichier.write( 'okay', 'okay')
```

Il est possible d'écrire des nouveaux environnements d'exécution pour **with** en définissant le code à exécuter en amont et en aval du bloc. Un objet qui implémente les méthodes spéciales *_enter_* et *_exit_* peut être passé à **with**. *_enter_* doit renvoyer un objet, qui est lié à la variable définie par *as*. La technique la plus simple est de rajouter ces deux fonctions à l'objet qui doit être utilisé dans le bloc, en renvoyant *self* dans *_enter_*:

```
>>> class Poli( object ):
...     def __enter__( self ):
...         print 'Bonjour'
...         return self
...     def __exit__( self, type, value,
...                 traceback ):
...         print 'Au_revoir',
...         def sson( self ):
...             print 'et_galipettes',
...
...     with Poli() as poli:
...         poli.sson()
```

Bonjour
et galipettes
Au revoir

exit qui correspond au bloc *finally*, récupère les informations sur l'exécution du bloc, comme le traceback en cas d'erreur :

```
>>> milettes = []
>>> class Poli( object ):
...     def __enter__( self ):
...         print 'Bonjour',
```

```

...
return self
...
def __exit__(self, type, value,
            traceback):
...
if traceback is not None:
...
    miettes.append("c'est par terre")
else:
...
    print 'Au revoir'

```

```

...
def sson(self):
...
    raise Exception('pouf')
>>> try:
...
    with Poli() as poli:
...
        poli.sson()
...
except:
...
    print str(miettes[0])

```

```

Bonjour
c'est par terre

```

Un nouveau module nommé *contextlib* fournit des fonctions qui permettent une écriture rapide des contextes pour **with**, à savoir :

- **contextmanager**, un decorator qui permet de définir une fonction qui prend en charge le travail de *_enter_* et *_exit_*.
- **nested**, qui permet de combiner des *with* imbriqués.
- **closing**, qui automatise l'appel à la méthode *close* de l'objet, pour peu qu'elle existe.

Exemple d'utilisation de *contextmanager* :

```

>>> class Poli(object):
...
    def sson(self):
...
        print 'et galipette'
>>> from contextlib import contextmanager
>>> @contextmanager
...
    def context():
...
        print 'Bonjour'
...
        yield Poli()
...
        print 'Au revoir'
...
>>> with context() as poli:
...
    poli.sson()
...
Bonjour
et galipette
Au revoir

```

Cette écriture permet de désolidariser le contexte de l'objet. Elle est appelée deux fois grâce au *yield*.

5.3.2 Bonnes pratiques

Outre les codes patterns présentés, certains réflexes permettent de mieux contrôler l'évolution du code écrit. Ces bonnes pratiques sont vitales dès lors qu'un programme dépasse plusieurs milliers de lignes, et est modifié par plusieurs intervenants. Sans ces précautions, le code tend naturellement vers le bloc monolithique²².

Les cinq lois présentées ci-dessous sont la synthèse de ces bonnes pratiques et peuvent s'appliquer à tout type de code :

- **Les bons noms font les bons amis**
- **Soigne ta signature**
- **Écourter les conversations**
- **Ne jamais radoter**
- **Diviser pour mieux régner**

Les bons noms font les bons amis

Le choix d'un bon nom pour chaque variable, fonction, classe ou méthode est fondamentale car :

- il aide à la compréhension du rôle de l'élément par le code utilisateur;
- il permet au développeur de prendre du recul sur l'utilité et l'objectif de l'élément;

Des API mal nommées sont souvent source d'incompréhension ou de mauvaise utilisation. Un élément bien nommé au contraire est auto-dокументé : son utilisation coule de source. Les conventions principales pour le nommage sont explicitées pour chaque type d'éléments ci-dessous.

Variables Une variable doit avoir un nom court mais précis, en minuscule. C'est en général un simple mot. Si un mot ne suffit pas, une séquence courte de mot est employée, avec des espaces soulignés entre chaque mot. Lorsque la variable est dédiée à gérer des valeurs booléennes, les suffixes *is* et *has* peuvent être utilisés pour fournir une indication supplémentaire. Les variables concernant des

²²Encore appelé *effectus spaghettiacus*.

séquences peuvent également êtres conjuguées au pluriel. Les variables constantes respectent les mêmes règles mais sont en majuscule.

```
>>> has_empty = True
>>> elements = [1, 2, 3]
>>> size = 12
>>> ELEVEN = 11
>>> PRIMARY_COLORS = ('blue', 'yellow', 'red')
```

Sauf projet particulier, l'anglais²³ est la langue utilisée pour les noms. N'en déplaît pas aux puristes, l'anglais est *la* langue universelle de l'informatique, même si cette règle ne s'applique que partiellement à ce livre pour des raisons évidentes.

Modules Un module est un mot écrit en minuscule²⁴, et suffixé de *lib* lorsqu'il concerne l'implémentation d'un protocole.

```
>>> import smtplib
>>> import imaplib
>>> import os, sys
>>> import zipfile
```

Notons qu'il est rare de rencontrer des modules dont le nom est une séquence de mots.

Classes Les classes s'écrivent avec un ou plusieurs mots dont la première lettre est en majuscule. Cette séquence de mots doit indiquer explicitement la nature de la classe. C'est le plus souvent un nom.

Voici quelques exemples de noms :

- *Generator*²⁵
- *DataEngine*
- *TemplateFile*
- *Publisher*
- *StringWidget*

– *Iterator*

Lorsque la classe est une classe de base, un suffixe *Base* ou un préfixe *Abstract* peuvent être ajoutées pour plus de précisions. L'essentiel est de respecter la même norme dans l'intégralité du code et de trouver des noms concis mais précis.

Méthodes et fonctions Les fonctions sont écrites en *mixedCase*²⁶ ou en *lower-case*²⁷. Choisir l'une ou l'autre des conventions est une affaire de goût, sachant que la plupart des bases de code optent pour le *lower-case* pour les fonctions et pour le *mixedCase* pour les méthodes.

Le nom des méthodes ou fonction informe également sur le type renvoyé dans certains cas :

- préfixe *is* ou *has* lorsque la fonction renvoie un booléen : *hasElement*, *isConnected*, etc. ;
- un *s* ou un *list* terminal lorsque la fonction renvoie une séquence.

Enfin, les noms des méthodes doivent être choisis en continuité avec celui de la classe, pour que la concaténation des deux soit complémentaire et non redondante :

- *MyList.getList()*, à remplacer par : *MyList.get()*
- *RS232.read()*, à remplacer par : *RSR232.read()*
- *TemplateFile.open()*

Soigne ta signature

Les signatures des fonctions et méthodes peuvent être à l'origine de fuites au niveau des dépendances entre éléments : un **args* ou ***kwargs* fourre-tout, ou tout autre paramètre variable mal contrôlé, affaiblit le code et rend son évolution difficile. Un nombre limité d'arguments de types non modifiables est à préconiser. Les arguments respectent la même sémantique que les variables.

Quelques exemples de cas particuliers pour lesquels la règle ne s'applique pas :

²³ En évitant les franglismes et autres erreurs de sens.

²⁴ Cette règle n'est pas encore respectée à 100 % dans la librairie standard mais le sera dans la version 3000.

²⁵ Les classes génériques qui implémentent un pattern se terminent souvent par *or*.

²⁶ Concaténation de mots commençant par un mot minuscule, puis par des mots dont la première lettre est en majuscule.

²⁷ Mots en minuscule séparés par des espaces soulignés.

- les fonctions de traitement de formulaires;
- les decorators qui s'appliquent à des fonctions dont la signature n'est pas connue;
- plus globalement, toutes les briques intermédiaires de code en charge de relayer des informations sans en connaître la forme précise.

Écourter les conversations

Lorsqu'un élément (classe, module, etc.) est utilisé dans du code, si plusieurs appels successifs sont effectués pour l'utiliser (sans ou avec peu d'interactions), ses API doivent être modifiées pour écourter les échanges au maximum. Le module `_init_` d'un paquet est un bon outil pour synthétiser son utilisation.

Exemple de conversation trop bavarde :

```
from pack.template import generator
from pack.view import list

params = generator.make_params()
view_params = generator.adapt_params(params)
my_list = list(view_params)
```

Les API du paquet et de ses modules peuvent être améliorées pour obtenir :

```
from pack import create_list
view = create_list()
```

Le module `_init_` du paquet `pack` joue alors le rôle d'intermédiaire entre `view` et `template`.

Ne jamais radoter

L'ennemi du code est le doublon. Dès que deux blocs présentent des similitudes, il est nécessaire de les réunir.

Diviser pour mieux régner

Une fonction ne devrait jamais dépasser un écran. Il est nécessaire de la subdiviser le cas échéant. Il en va de même pour les classes, modules, et paquets : il ne faut pas avoir peur de diviser son paquet en une multitude de petits paquets.

5.4 Outils d'assurance qualité

La criminalité dans le métro new-yorkais a chuté dans les années quatre-vingt, grâce à l'application d'une théorie imaginée par le criminologue *Kelling* et décrite dans son ouvrage [6]. C'est la théorie de la fenêtre cassée, qui part du principe qu'un bâtiment dont un carreau est cassé est considéré par les passants comme un bâtiment laissé à l'abandon : bientôt d'autres fenêtres seront cassées. *Kelling*, pendant des années, a appliqué ce principe pour venir à bout des crimes dans le métro new-yorkais : chaque graffiti qui apparaissait était immédiatement nettoyé, les policiers arrêtaient les petits malfrats pour le moindre petit délit. Le contexte devenait de moins en moins propice à l'émergence de crimes, et le métro new-yorkais, réputé pour être un coupe-gorge, devint l'un des métros les plus sûrs du monde.

Cette théorie est applicable au génie logiciel, et a été présentée dans ce domaine par Andrew Hunt et David Thomas [2] : il ne faut faire aucune concession sur la qualité, sous peine de voir la base de code devenir de moins en moins bonne. On parle alors d'*entropie logicielle*. En effet, les développeurs auront tendance à ne pas être rigoureux s'ils manipulent du code qui est visiblement de mauvaise qualité. Inversement, si le code est de très bonne qualité, les développeurs seront plus vigilants.

Faire un contrôle régulier sur la qualité du code produit, pour supprimer au plus tôt toute faiblesse, est donc une activité essentielle pour le développement d'une base de code.

Par qualité il faut entendre :

- le respect des conventions de codage;
- le respect de normes de programmation, comme la complexité des algorithmes, la taille des fonctions ou encore le nombre

- de boucles imbriquées²⁸. Ces normes sont mesurées via des métriques ;
- enfin, le fonctionnement de l'intégralité des tests²⁹.

Il existe plusieurs outils de contrôle qualité, capables de lancer sur la base de code une batterie de tests et de fournir un rapport détaillé. Le plus complet et configurable est **PyLint**. Un projet complémentaire, appelé **Cheesecake**, propose des métriques originales pour tester la présence d'éléments standard lorsque le code est distribué.

5.4.1 PyLint

- PyLint* est un script de contrôle de code qui effectue des tests sur :
- les erreurs de programmation, comme des oubli\$ d'import ou des appels à des méthodes qui n'existent plus;
 - les déchets de programmation, comme les importations de module qui ne sont jamais utilisées, ou les portions de code qui ne sont jamais appelées;
 - les conventions de nommage, ou de disposition de code.

Le contrôle des erreurs de programmation n'est pas le plus important si le code est conçu par les tests (voir chapitre 6). Les contrôles des conventions et des déchets sont quant à eux très utiles pour améliorer la qualité du code. Il peut être installé par le biais d'`easy-install`, ou par un tarball classique³⁰.

Une fois l'outil en place, la commande `pylint` permet de lancer l'analyse du code. Elle peut être appelée avec un nom de fichier ou de dossier. Les jokers sont autorisés pour les noms de fichier. Voici un exemple de sortie :

```
dabox: ~ tarek$ pylint *PY
***** Module __init__
C0111: 1: Missing docstring
***** Module event
W0142: 36:lock.wrap: Used * or ** magic
***** Module converters
C0103: 26:getPGTimeStamp: Invalid name "getPGTimeStamp"

```

Les contrôles effectués par PyLint sont par défaut très stricts, et peuvent parfois ne pas correspondre aux conventions adoptées par le projet. Mais l'outil permet de configurer entièrement les contrôles, par le biais d'un fichier de configuration `.pylintrc`. Il est généré par la commande `pylint --generate-rcfile > .pylintrc`. Ce fichier est à placer dans un des chemins parcourus par PyLint au démarrage. Le plus simple est de le placer dans son répertoire personnel.

Voici un extrait de fichier de configuration, concernant le paramétrage basic sur le contrôle des noms :

```
[BASIC]
enable-basic=yes
required-attributes=
no-docstring-rgx=_.*__
min-name-length=3
module-rgx=(( [a-z_] [a-zA-Z0-9_]* )|([A-Z] [a-zA-Z0-9]+) )
class-rgx=[A-Z] [a-zA-Z0-9]+$
function-rgx=[a-z_] [a-zA-Z0-9_]*$
method-rgx=[a-zA-Z] [a-zA-Z0-9_]*$
argument-rgx=[a-zA-Z] [a-zA-Z0-9_]*$
variable-rgx=[a-zA-Z] [a-zA-Z0-9_]*$
good-names=os,_,ip,js,x,y,up
bad-names=foo,bar,baz,toto,tutu,tata
bad-functions=map,filter,apply,input
```

Chaque contrôle de nom est régi par une expression régulière, qui peut être facilement adaptée.

Ces tests ne remplacent cependant pas les contrôles manuels, qui permettent de corriger les portions de *bad smelling code*. Les *bug days* sont propices à ce genre de contrôle (voir section 8.2).

5.4.2 Cheesecake

Dans la même optique de contrôle qualité automatique, **Cheesecake**³¹ est un outil qui propose de mesurer la qualité d'un paquet Python en se basant sur un certain nombre de critères. PyLint est

²⁸Complexité de MacCabe.

²⁹Ce point est abordé dans le chapitre 6.

³⁰Voir www.logilab.org/857.

utilisé pour mesurer la qualité du code, et d'autre tests sont menés, comme la vérification de la présence de tests, d'un fichier README, etc.

Exemple de sortie :

```
$ python cheesecake_index -n nose
py-pi_download ..... 50
unpack ..... 25
setup.py ..... 25
install ..... 50
=====
INSTALLABILITY INDEX (ABSOLUTE) 165
required_files ..... 110
docstrings ..... 43
formatted_docstrings ..... 0
=====
DOCUMENTATION INDEX (ABSOLUTE) 153
DOCUMENTATION INDEX (RELATIVE) 44
unit_tested ..... 30
pylint ..... 37
pep8 ..... -16
=====
CODE KWALITEE INDEX (ABSOLUTE) 51
CODE KWALITEE INDEX (RELATIVE) 64
=====
OVERALL CHEESECAKE INDEX (ABSOLUTE) 369
OVERALL CHEESECAKE INDEX (RELATIVE) 62
```

Bien configuré, il permet de définir des critères de validation pour les paquets développés pour un projet.

5.5 Ce qu'il faut retenir

Ce chapitre a regroupé l'essentiel des bonnes pratiques de programmation, et les bons réflexes que le développeur doit acquérir pour produire du code de qualité.

Il est le tremplin pour les deux chapitres suivants, qui introduisent des techniques de programmation agiles : le *développement dirigé par les tests*, et le *développement dirigé par la documentation*.

Chapitre 6

Développement dirigé par les tests



Un programme sans tests est un programme mort.

Bud, chef de projet.

6.1 Philosophie

Dès lors qu'un programme dépasse le petit script système conçu sur un coin du disque dur, des problèmes fondamentaux se posent :

comment réussir à le faire grandir de manière harmonieuse, par le biais de différents intervenants et à long terme ? Comment parvenir à corriger et maintenir une base de code composée de dizaines de milliers de lignes, de centaines de fichiers ? Comment corriger un bug sur une portion de code écrite par Bernard, parti pour trois semaines à un stage de tir à l'arc dans le Périgord ?

Outre une gestion de projet méthodique, décrite dans le chapitre 8, les tests sont la solution incontournable.

6.1.1 Qu'est-ce qu'un test ?

Chaque fonction ou méthode du programme est écrite avec un certain nombre de paramètres et effectue une tâche donnée, qui renvoie éventuellement un résultat et change l'état interne du programme si nécessaire. Pendant la phase de conception de cette fonctionnalité, le développeur a en tête quelques *use cases*, c'est-à-dire des cas d'utilisation réels dans le programme, qui l'ont amené à faire ces choix de développement.

Un test consiste à appeler la fonctionnalité avec un scénario qui correspond à un cas d'utilisation, et à vérifier que cette dernière se comporte comme prévu.

6.1.2 Quand interviennent les tests dans le cycle de développement ?

Les tests sont écrits en même temps que le code, et dans l'idéal juste avant. On parle de *développement dirigé par les tests*, ou *test-driven development* en anglais (TDD), et le développeur navigue entre le code et le test qui lui correspondent pour faire grossir l'application. Si le principe est correctement appliqué, et si tous les cas d'utilisation sont agrémentés de tests, le code est entièrement couvert par une batterie de tests.

Dans l'exemple ci-dessous, la fonction *puissance*, qui renvoie la puissance d'un nombre, est conçue par deux *uses cases* qui déterminent le résultat attendu.

```
>>> def test_1():
...     Traceback (most recent call last):
...     NameError: global name 'puissance' is not defined
>>> test_2()
...     Traceback (most recent call last):
...     NameError: global name 'puissance' is not defined
```

L'exécution des tests échoue puisque la fonction n'est pas encore codée. Mais leur construction a pu être faite sans avoir à élaborer le code dans un premier temps : *test_1* et *test_2* sont les définitions directes des cas d'utilisations imaginées par le développeur, qui peut ensuite concevoir *puissance* :

```
>>> def puissance(nombre, exposant):
...     return nombre**exposant
...
>>> test_1()
OK
>>> test_2()
OK
```

Le code peut être intégralement conçu par ce biais, même pour les fonctionnalités les plus complexes, et ceci grâce à des outils de la librairie standard décrits dans la suite de ce chapitre. La batterie de tests ainsi constituée peut être lancée dans une *campagne de tests*, chargée de tester l'intégralité du programme. Ces tests ne font évidemment pas partie du code de l'application, mais sont l'outil de conception central.

6.1.3 Quels sont les avantages des tests ?

Cette technique de programmation offre de nombreux avantages, à savoir :

- un recul sur son travail pour le développeur, qui réfléchit et corrige, par ce simple exercice, de nombreuses erreurs de conception et autres coquilles ;
- une garantie pour la *non-régression* : relancer l'intégralité des tests à chaque modification permet de s'assurer qu'une mo-

dification du programme pour une fonctionnalité donnée ne casse pas d'autres fonctionnalités;

- la meilleure des documentations techniques : la lecture des tests permet de comprendre précisément les objectifs d'une fonctionnalité. Cette documentation est en outre toujours à flot.

Comme pour toutes les méthodologies, les développeurs non-initiés opposent parfois une résistance aux principes des tests, car l'expérience n'est pas intellectuellement naturel. Mais les gains sont très rapidement perceptibles et rares sont ceux qui n'adhèrent pas au principe. La critique la plus fréquemment mise en avant concerne la perte de temps à concevoir les tests. Mais le temps passer à débugguer une base de code non testée est beaucoup plus long car chaque correctif apporté est susceptible d'entraîner des régressions. Il n'est pas possible de mesurer précisément le gain obtenu car chaque projet est unique. Mais mesurés lors de la réalisation d'une application, le temps passé à écrire les tests et le temps passé à corriger le code indiqueront très rapidement les gains obtenus.

On distingue deux types majeurs de tests : les *tests unitaires* et les *tests fonctionnels*.

6.1.4 Tests unitaires

Un test unitaire est chargé de valider de manière isolée du reste du programme le fonctionnement d'une classe, d'une méthode ou d'une fonction. Cet isolement permet de garantir une impartialité du test, qui fournit lui-même un contexte d'exécution et s'assure qu'aucun autre élément du programme ne vient interférer. Par convention, chaque module de code dans un programme Python est associé à un module de tests unitaires, placés dans un répertoire *tests* du paquet. Ainsi un module *calculs.py* aura un module de tests correspondant *test.calculs.py* dans un sous-répertoire *tests*.

La préparation d'un contexte d'exécution, similaire aux contextes rencontrés par le module lors d'une exécution réelle, est appelée *test fixture*. Ce travail préliminaire peut nécessiter la mise en place d'éléments qui simulent des fonctionnalités connexes du programme, appels bouchons.

Les bouchons

Un module qui fait appel à un serveur externe, comme un serveur IMAP¹, ne peut pas être testé de manière efficace s'il accède à un serveur réel : les tests doivent pouvoir s'exécuter sans aucune dépendance externe et fonctionner sans conditions d'environnements particulières. La technique la plus simple à mettre en place, possible grâce aux caractéristiques dynamiques de Python, consiste à créer des objets en charge de remplacer les éléments externes. Dans l'exemple d'IMAP, il s'agit de concevoir un *faux serveur IMAP*, docile et prêt à réagir aux appels du code pour que les tests se déroulent sans obstacles². Élaborer un bouchon s'effectue de manière implicite : il est d'abord ajouté vide, puis s'étoffe de fonctionnalités jusqu'à ce que les tests passent.

6.1.5 Tests fonctionnels

Les tests fonctionnels prennent quant à eux l'application comme une boîte noire et la manipule comme le ferait l'utilisateur final, pour valider que toutes les fonctionnalités sont là et réagissent comme prévu. Ces tests doivent passer par les mêmes interfaces que celles fournies aux utilisateurs ; c'est pourquoi ils sont spécifiques techniquement à la nature de l'application (interface Desktop, interface web, etc.) et plus délicats à mettre en œuvre.

Les tests fonctionnels sont en général les parents pauvres des projets pour cette raison, mais aussi parce qu'ils sont plus difficiles à ajouter au cycle de développement. En effet, leur conception demande un effort supplémentaire car ils sont globaux à l'application et obligent les développeurs à s'y consacrer spécifiquement. Cette tâche devient alors un *job à plein temps* et beaucoup de projets les boudent.

Le reste de ce chapitre présente les outils disponibles dans la bibliothèque standard pour effectuer des tests unitaires et élude volontairement les outils pour les tests fonctionnels, trop spécifiques à l'interface des applications développées. Le lecteur intéressé peut se référer à ces projets :

- Selenium : la référence en tests fonctionnels pour les applications Web.³

¹Serveur de mails.

²Un peu comme une YES card.

³<http://www.openqa.org/selenium>

- **guitest** : teste les applications GTK.

Une page disponible sur le web résume également l'intégralité des outils de tests disponibles pour Python :
<http://pycheesecake.org/wiki/PythonTestingToolsTaxonomy>

6.2 unittest et doctest

Python propose deux modules pour l'élaboration des tests, à savoir :

- **unittest**, outil calqué sur *JUnit* ;
- **doctest**, outil basé sur le principe du literate programming.

```
if dirname == '':
    dirname = '.'
dirname = os.path.realpath(dirname)[0]
updir = os.path.split(dirname)[0]
if updir not in sys.path:
    sys.path.append(updir)
```

6.2.1 unittest

JUnit, outil d'aide à la conception de tests en Java, inspiré d'un outil similaire développé pour SmallTalk, a été à son tour un modèle pour la plupart des langages modernes. On retrouve ainsi *NUnit* pour .Net, *JSUnit* pour Javascript et *PyUnit* pour Python, développé dans le module *unittest*.

PyUnit propose une classe de base, *TestCase*, qui peut être utilisée pour concevoir des tests. Chaque méthode implementée dans une classe dérivée de *TestCase* et préfixée de *test* sera considérée comme un test unitaire. Lancer les tests de la classe consiste alors à exécuter toutes ces méthodes les unes après les autres. Les classes de tests sont ensuite regroupées dans des séquences, appelées *tests suites*, qui sont utilisées pour lancer des campagnes de tests. Dans un quel Python, chaque module de test implémente conventionnellement une seule classe de tests, qui pourra être ajoutée à une campagne de tests ou lancée individuellement. Cette classe teste l'ensemble des éléments du module cible, et aucun autre module.

Dans l'exemple ci-dessous, *tests/test_calculs.py* est le module de tests de *calculs.py*.

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

if __name__ == '__main__':
    unittest.main(defaultTest='test_suite')
```

Quelques remarques sur ce script :

- Par convention, les tests sont renvoyés par une fonction globale appelée *tests suite*, sous forme de *suite*. Cette forme permet à des scripts de tests de récupérer les tests du module.
- La fonction *main* permet d'exécuter les tests lors d'un appel direct.
- Le script étant conventionnellement placé dans un sous-répertoire, le path du répertoire parent est ajouté à *sys.path* pour permettre les imports du code. Cette manipulation devient superflue lorsque les modules de tests sont invoqués par un script de tests qui se charge lui-même de peupler les chemins de recherche de l'interpréteur.

Une exécution directe de ce script donne :

```
dabox:~ tarek$ python test_calculs.py
test 1
.test 2
```

```
dirname = os.path.dirname('..file..')
```

```
updir = os.path.split(dirname)[0]
if updir not in sys.path:
    sys.path.append(updir)

Ran 2 tests in 0.016s
```

OK

La classe *TestCase* fournit des méthodes qui permettent de valider des valeurs pour vérifier les résultats rentrés par les fonctions testées. Elles effectuent des assertions sur les valeurs et déclenchent une *AssertionError* en cas de problème. Les méthodes les plus utilisées sont :

- `assertEquals(el1, el2)` : provoque une erreur si *el1* est différent de *el2*;
- `assertNotEqual(el1, el2)` : teste l'inégalité;
- `assertRaises(Exception, func, arg1, arg2, ...)` : s'assure que la fonction *func* provoque une exception de type *Exception*.

Prenons le module *calculs.py* suivant :

```
def moyenne(*args):
    """ renvoie la moyenne """
    length = len(args)
    sum = 0
    for arg in args:
        sum += arg
    return float(sum) / float(length)

def division(a, b):
    """ renvoie la division """
    return a / b

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

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

Le module de tests pourrait alors être modifié ainsi :

import unittest
import os
import sys

dirname = os.path.dirname(__file__)
if dirname == '':
    dirname = '.'
dirname = os.path.realpath(dirname)

from calculs import moyenne
from calculs import division

class CalculsTest(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)

La construction du code consiste donc à effectuer des allers-retours entre le module et son module de tests. Lorsque la tâche du développeur consiste à corriger un bug, il insère un test qui provoque ce bug et corrige ensuite le code. Cette capitalisation de tests assure la non-régression car l'intégralité des tests est relancée sur la base de code fraîchement modifiée.

Pour effectuer des campagnes de tests, chaque framework Python propose un petit script en charge de collecter les modules de tests et de construire une liste de suite. Zope propose un script extrêmement complet, avec des dizaines d'options4. Il peut être utilisé sur tout projet Python.

Une autre option est de créer soi-même ce script pour qu'il :
```

⁴<http://www.python.org/pypi/zope.testing>

- recherche tous les modules de test. Leur 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.

6.2.2 doctest

Les *doctest* proposent une alternative originale aux tests unitaires : basés sur le principe du *literate programming*⁵, ils autorisent le développeur à insérer dans ses documents texte des exemples de code, qui sont exécutés ensuite comme des tests : le module *doctest* extrait des textes les exemples et les exécute pour vérifier qu'ils fonctionnent réellement. Cette technique permet la mise en place d'une méthodologie de programmation qui fait l'objet du chapitre suivant.

6.3 Ce qu'il faut retenir

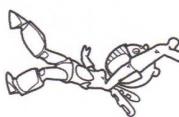
Les tests sont devenus incontournables car **ils offrent aux dévelopeurs les moyens d'être agiles sur la base de code**, même si cette dernière est immense : il n'est plus risqué d'effectuer de profondes modifications avec un tel fillet de sécurité.

Cette philosophie de programmation permet également d'augmenter drastiquement la qualité du code, par le recul imposé par l'acte de tester.

Enfin, les tests deviennent la meilleure documentation, et ce principe est à la base du *développement dirigé par la documentation*, présentée au prochain chapitre.

Chapitre 7

Développement dirigé par la documentation



Lorsque Chuck Norris tombe à l'eau, ce n'est pas Chuck Norris qui est mouillé, c'est l'eau qui est chucknorrisée.

Fabrice

7.1 Philosophie

La documentation est omniprésente dans l'informatique, du casier des charges, en passant par les spécifications fonctionnelles, jusqu'au manuel utilisateur. Elle joue un rôle fondamental, pour la compréhension et l'adoption de la base de code. C'est paradoxalement le pôle le plus bâclé dans la plupart des projets de logiciel : peu ou aucun budget n'est alloué pour développer une documentation digne de ce nom, et elle évolue de manière inégale au gré des bonnes intentions.

On distingue plusieurs types de documents :

- les documents de spécification : l'**upstream documentation** ;
- les documents techniques, susceptibles d'évoluer au même ry-

⁵Présenté au prochain chapitre.

thme que le code : **le mainstream documentation** ;

- les manuels d'utilisation de *haut niveau* et autres documents de vente et marketing, **le downstream documentation**.

Dans un projet logiciel au développement ouvert, le mainstream est le pôle de documentation le plus important pour faire entrer dans le projet de nouveaux intervenants, et pour maintenir une base de connaissance commune. Le downstream joue également un rôle fondamental mais sa conception et sa mise à jour sont très simples et très rapides s'il se base sur un mainstream solide.

Pour cette raison, et aussi parce que ce type de documentation est produit la plupart du temps par les développeurs, pour des développeurs, ce chapitre se concentre sur le mainstream et propose une méthodologie pour intégrer sa conception dans le cycle de développement logiciel. La technique présentée a en outre l'avantage de se combiner avec la programmation dirigée par les tests, pour proposer une nouvelle discipline agile pour le développeur : **le développement dirigé par la documentation** (DDD).

Avant de présenter les outils et les techniques de DDD spécifiques à Python, puis des sujets plus avancés, il convient de bien comprendre ce qu'est un document mainstream, et les rôles qu'il peut avoir dans un projet informatique, et de maîtriser les techniques de base de l'écriture technique.

Ce chapitre est organisé en cinq sections :

- Les sept lois de l'écriture technique
- Les outils d'écriture pour Python
- L'organisation de la documentation
- Le développement dirigé par la documentation
- Le *team writing*

7.2 Les sept lois de l'écriture technique

Contrairement aux idées reçues, concevoir une bonne documentation technique est à la portée de tous les développeurs, quelles que soient la qualité et la richesse de leur écriture. Cette section présente les règles de base pour produire des documents de qualité. On compte

sept règles importantes, qui s'appliquent dans tous les contextes :

- une technique d'écriture : **l'écriture en deux étapes** ;
- un style simple ;
- un lectorat ciblé ;
- une information ciblée ;
- des exemples réalistes ;
- l'approche *light but sufficient* ;
- des documents structurés.

7.2.1 Écriture en deux étapes

Dans *Writing with Power* [4], Peter Elbow fournit une série de techniques pour aider les écrivains à gagner en efficacité. Il part d'un constat simple : notre cerveau n'est pas capable de synthétiser en temps réel les idées qui le traversent pour qu'elles soient directement retranscrites, sous une forme grammaticale parfaite, suivant un plan précis et ordonné, et *sans fautes d'orthographe si vous plait*. La plupart des écrivains en herbe passent donc leur temps à buter sur la forme de leurs phrases, en reprenant encore et encore le paragraphe en cours, avec un objectif en tête : accoucher d'une écriture parfaite au premier jet. Autant dire que cette technique n'est pas à la portée du commun des mortels. Seul Mozart¹ et une poignée de surdoués en sont capables. Écrire un document est après tout similaire à l'écriture de code : on y revient sans cesse.

Le problème de *l'écriture directe* est que la dépense d'énergie et de concentration pour essayer de concevoir un document fini, se fait au détriment du flot d'idées qui nous traverse lorsque nous écrivons, et ce quel que soit notre niveau d'écriture. Elbow propose une technique simple pour éviter de perdre cette richesse : *l'écriture en deux étapes*.

Étape un : retrancrire les idées

Un bon exercice pour comprendre l'intérêt de procéder à l'écriture en deux phases consiste à écrire librement sur un sujet donné en notant tout ce qui traverse l'esprit, quelle que soit sa forme. L'écriture doit être réalisée sans arrêts, et le réflexe de revenir en arrière pour corriger une faute d'orthographe ou une forme grammaticale bancale doit être oublié. Vous devenez une secrétaire en charge de prendre des notes sur une émission de radio en direct : *Radio Cerveau*.

¹Et c'est sûrement une légende, du buzz marketing de l'époque.

Si le principe est bien appliqué, les phrases produites seront beaucoup plus proches du langage parlé et ressembleront à un enchaînement d'idées mal organisées mais portant pour la plupart sur le sujet. Les idées *connexes*² qui traversent l'esprit pendant cet exercice, peuvent être conservées sur une deuxième page ou entre parenthèses : dès qu'elles surgissent, le stylo ou le clavier peut passer sur une *annotation* pour les noter, puis reprendre le flux principal. L'essentiel étant de ne pas interrompre le processus.

L'exemple ci-dessous a été produit en suivant ce principe, et concerne le sujet suivant : "Que pensez-vous du réchauffement climatique ?"³ Il a été écrit d'une traite, sans se soucier du style ou de l'orthographe :

le réchauffement c'liamtique j'ai vu un article avec des ours sur une banquise fondu ca fait peur, mais je pense qu'il faudrait des chiffres précis on est un peu maniuplé par les mediais (tele ou internet media de masse ca devient pareil non ?) ca devint un sujet politisé et instrumentalisé mais bon a priori ca nous concerne tous des efforts minimes ,l'education des générations (le one laptop per child ca va être un outil d'education formidable) a venir pour pouvoir minimiser les problemes c'est plutot important car on va pouvoir minimiser radicalement les émissions de pollutions et tout ca vivement la voiture à oxygène (et les pubs sans tabac l'an prochain) aussi (j'ai vu des voiture à air comprimé c'est terrible)

Ce bloc peut ensuite être organisé et mis en forme.

Étape deux : mettre en forme

Reprendre le bloc produit précédemment et le mettre en forme peut être réalisé sans aucune difficulté :

Le réchauffement climatique est, semble-t-il, provoqué par l'activité humaine. Il est responsable de la fonte

²Qui ne sont pas en rapport direct avec le sujet mais qui restent dignes d'intérêt.

³Par une personne qui ne connaît absolument rien au sujet.

des glaces et de bouleversements climatiques qui mettent en danger les espèces vivantes sur terre.

Le sujet est au cœur des débats de société, en cette période électorale, et une prise de conscience collective est en train de s'opérer, même si le sujet est instrumentalisé par les politiciens.

L'éducation, et notamment la modification de nos habitudes de vie semble être la solution pour inverser les tendances actuelles.

Les énergies renouvelables, et les voitures non polluantes, comme les voitures à hydrogène sont autant de pistes pour réduire les émissions de gaz à effet de serre.

Cette deuxième phase a permis d'ordonner les idées et de les enrichir et les consolider grâce au recul de la relecture. En accumulant le temps nécessaire pour chacune de ces deux phases, la durée globale pour obtenir ce résultat sera toujours plus courte qu'une écriture directe : l'écrivain qui ne fait pas deux tâches à la fois est plus efficace.

7.2.2 Style simple

Dans les textes techniques, il n'est pas question d'enroulées lyriques ou de formes stylistiques complexes. L'objectif est de faire comprendre le fonctionnement d'un programme, ou de faire passer des idées. Le texte est évidemment plus captivant lorsque l'auteur a une bonne plume ou est capable d'ajouter de l'humour ici et là, mais il ne faut pas en faire un objectif.

Privilégier :

- des phrases courtes : deux lignes (de 80 signes chacunes) tout au plus ;
- un vocabulaire simple mais précis ;
- des paragraphes concis : 5 à 6 phrases tout au plus.

Éviter :

- l'usage d'un style personnel (*je, vous, ils*), qui a tendance à romancer les textes⁴;
- un autre temps que le présent.

7.2.3 Lectorat ciblé

Un document technique doit s'adresser à un lectorat bien défini : chef de projet, client, intégrateurs, graphistes, développeur, etc. Définir la cible dès le début de l'écriture est primordial, et cette dernière doit rester à l'esprit pendant toute la conception : l'écrivain doit faire preuve d'empathie envers ses lecteurs, et employer le ton et les référents culturels qui leur correspondent.

En effet, un développeur n'aura pas les mêmes connaissances ou la même culture qu'un client. Même si cette règle paraît évidente, elle est rarement appliquée et les documents produits sont moins efficaces.

Deux actions pour éviter de tomber dans cet écueil :

- commencer le document par cette question : "À qui s'adresse ce document ?" suivi d'une réponse, la plus précise possible;
- faire lire le document à une personne cible, pour des retours.

7.2.4 Information ciblée

Chaque document produit doit traiter d'un sujet précis, et un seul. Dès lors que cette règle n'est pas respectée, la documentation devient difficile à utiliser car l'information est noyée. Passer du temps à chercher une réponse à une question dans l'ensemble documentaire d'un logiciel est un signe de non respect de cette règle : le lecteur sait⁵ que ce qu'il cherche se trouve dans un des documents, mais ne le trouve pas facilement.

Pour éviter ce problème, il suffit de se mettre à la place des lecteurs : rechercher des informations sur Internet se fait en tapant une phrase courte ou une question précise dans un moteur de recherche.

De la même manière, un document doit pouvoir correspondre à une phrase ou une question précise. Le principe peut également être appliqué à l'ensemble des sections du document⁶ : pour chaque partie,

il convient de se poser la question : "Quelle phrase serait saisie sur Google pour retrouver ce passage ?", et de l'utiliser. Voici un exemple de trame, sur un document concernant l'indexation :

Comment fonctionne l'indexation ?

- A/ Normaliser le texte
...
- B/ Retirer les mots trop courts ou trop communs
...
- C/ Créer un dictionnaire des mots
...

Un bon moyen de vérifier la qualité de la trame est de l'utiliser pour composer l'index du document.

7.2.5 Exemples réalistes

Documenter du code ne change en rien les principes énoncés ci-dessus. La seule différence concerne les extraits de code susceptibles d'agrémenter les explications : il s'agit de présenter des *exemples réalisistes*.

Le module "graph" permet de créer un objet 2D, à partir d'une série de points fournis.

```
>>> from package import graph
>>> foo = graph.compute_square(1, 1, 1, 1)
>>> bar = graph.render_square(foo)
```

Cet extrait de documentation est relativement clair, mais l'exemple est surréaliste : en pratique, le développeur n'utilisera jamais *foo* ou *bar* comme nom de variable, ou des coordonnées *1, 1, 1, 1*. En modifiant légèrement l'exemple pour le rendre réaliste, la

⁴ Je vous jure que c'est vrai, ils me l'ont dit !

⁵ On lui a dit.

⁶ C'est-à-dire la trame.

⁷ La forme 2D rendue risque d'être plus plate qu'un soufflé au potiron qui n'est pas mangé dans les 20 min.

documentation devient plus précise et plus compréhensible. En effet, si l'utilisation du code présenté est évidente pour l'auteur, les lecteurs qui l'abordent pour la première fois n'ont pas forcément une compréhension instantanée.

Le module "graph" permet de créer un objet 2D, à partir d'une série de points fournis.

```
>>> from package import graph
>>> square = graph.compute_square(1, 7, 1, 10)
>>> square_view = graph.render_square(square)
```

Le principe à retenir pour les exemples de code est qu'ils doivent pouvoir être copiés-collos tels quels et fonctionner dans un vrai programme.

7.2.6 Approche *light but sufficient*

Écrire la documentation n'est pas une priorité dans la conception d'un logiciel. Il est toujours préférable d'avoir une application qui marche plutôt qu'une documentation exhaustive. Scott Ambler [1] explique qu'il faut pouvoir, comme pour tous les pôles de création d'un logiciel, justifier de l'utilité de créer une documentation : elle ne doit pas être écrite pour le principe mais fournir une véritable valeur ajoutée.

Ce principe s'exprime par les règles qui limitent la portée de chaque document (information et lectorat ciblés) et doit être complété par une modularisation du contenu. En effet, les documents doivent concerner une portion bien définie du logiciel et conserver une taille limitée : quatre à cinq pages⁸ est un maximum. Si cette taille est dépassée, il est préférable de faire plusieurs documents.

Cette granularité permet une meilleure exposition des idées présentées et évite les redondances : il devient plus facile de se concentrer sur l'essentiel. Enfin, limiter la taille des documents force le développeur :

- à synthétiser ses explications ;

– à avoir un regard critique sur son travail : s'il faut plus de cinq pages, le code n'est-il pas trop compliqué ou pas assez modulaire ?

Cette approche permet au final de n'écrire que le strict nécessaire et de rester *light but sufficient*.

7.2.7 Documents structurés

Lorsqu'un internaute parcourt un site d'achat en ligne comme Amazon, il enregistre au bout de quelques articles visualisés la position de chaque élément, comme le descriptif, le prix, les commentaires, et la structure de la fiche article. Cette carte mentale lui permet d'accélérer la lecture des informations et de s'approprier le contenu sans buter sur la forme.

Ce principe est à appliquer aux documents techniques en élaborant un *patron* pour chaque type de document : la *typologie documentaire* ainsi constituée offre au lecteur un mode d'emploi pour l'utilisation de la documentation et aux écrivains des rails pour sa conception.

Les documents contiennent souvent :

- un titre, sous la forme d'une phrase courte à l'infiniif;
- un résumé de quelques phrases ;
- un index, lorsque le document contient plusieurs sections ;
- le corps du texte, composé de section et de :

tableaux ;

- diagrammes ;
- une liste de références.

Un document ne doit jamais être écrit sans se baser sur un patron.

7.3 Outils d'écriture

La documentation technique mainstream évolue au même rythme que le code et doit donc être traitée de la même manière : il n'est

⁸Une page contient environ 4000 signes, soit une cinquantaine de lignes de

80 signes (12 mots par ligne en moyenne).

7.3. OUTILS D'ÉCRITURE

pas question d'utiliser des formats élaborés comme *Writer* d'OpenOffice.Org ou *LaTeX*. Chaque document doit pouvoir être lu et manipulé avec un éditeur de texte simple, même si l'utilisateur final peut disposer d'une version visuellement améliorée.

Il existe deux outils majeurs pour concevoir des documents pour les applications Python :

- le **reStructuredText**, un format texte enrichi ;
- les **doctests**, compatibles avec le format **reStructuredText**, qui permettent de combiner les textes explicatifs avec des exemples de code exécutables.

7.3.1 Le reStructuredText

Le *reStructuredText* ou *reST* est un système de balises utilisé pour formater des textes. Il est comparable à *LaTeX*, mais enrichit le document de manière non intrusive : les fichiers restent directement lisibles.

Exemple de fichier *reST* :

```
=====
Fichier au format reST
=====
```

Section 1

```
=====

```

On est dans la section 1.

Sous-section

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

C'est une sous-section

Section 2

```
=====

```

La section 2 est un peu vanteuse : la section 1 est
très bien.

Ce document au format texte peut être manipulé par tout éditeur sans aucun problème et offre un avantage indéniable par rapport aux formats binaires ou propriétaires : les documents peuvent être traités comme des codes sources dans le projet, que ce soit au niveau de la gestion des modifications⁹, ou au niveau de leur utilisation : un fichier *reST* peut être transformé en différents formats comme le *HTML*, le *RTF* ou encore le *LaTeX*. On parle alors de *rendu* et c'est cette forme finale qui servira à la conception de documents destinés aux utilisateurs ou à la communauté.

Le projet *docutils*¹⁰, qui inclut l'interpréteur *reST*, fournit également un jeu d'utilitaires pour transformer un fichier *reST* en un autre format. Les principaux sont :

- le **rst2html** qui génère un rendu *HTML* avec une *css* intégrée ;
- le **rst2latex** qui crée un fichier *LaTeX* équivalent ;
- le **rst2s5** qui construit une présentation au format *S5*¹¹, qui permet de créer des présentations interactives en *HTML*.

docutils peut être installé manuellement¹² ou sous Linux par le biais des paquets du gestionnaire de votre distribution¹³.

L'exemple précédent, rendu grâce à la commande *rst2html*, est présenté dans la figure 7.1. Pour la suite de cette section, les exemples de rendus sont tous effectués avec *rst2html*.

Pour les concepteurs de documentation, maîtriser le *reST* se résume à :

- maîtriser la structuration du document ;
- comprendre le formatage du texte ;
- savoir construire des listes ;

⁹Voir section 8.1.4.

¹⁰<http://docutils.sourceforge.net>

¹¹<http://meyerweb.com/eric/tools/s5>

¹²Voir <http://docutils.sourceforge.net/README.html#quick-start>.

La section 2 est ‘beaucoup’ plus intéressante que la section 1

Fichier au format reST

Section 1

On est dans la section 1.

Sous-section

C'est une sous-section

Section 2

La section 2 est beaucoup plus intéressante que la section 1

Section 3

La section 2 est un peu vanteuse: la section 1 est très bien.

FIG. 7.1 – Exemple de rendu au format HTML grâce à rst2html

- savoir construire des tableaux;
- connaître la syntaxe des hyperliens, bas de page et autres éléments, qui permettent d'enrichir les documents.

Structuration du document

La structuration d'un fichier reST se fait en soulignant les titres des sections avec des caractères de ponctuation (= - - :, etc.). À chaque fois qu'il rencontre un texte ainsi souligné, l'interpréteur associe le caractère utilisé à un niveau de section. Il est possible de *surligner* un texte pour augmenter sa visibilité. Par convention, le titre d'un document l'est souvent, comme dans l'exemple ci-dessus.

Les exemples de code ou autres extraits qui doivent être présentés sous forme de blocs à fonte monotypée¹⁴ doivent être :

- précédés du signe ::;
- indentées d'au moins un caractère¹⁵ ;
- entourées d'un saut de ligne.

Python utilise l'indentation pour délimiter les blocs:

```
>>> for nom in ('Pim', 'Pam', 'Poum'):
...     nom_complet = '%s %s' % nom
...     if nom == 'Pam':
...         moscou = " dit 'la Tigresse Moscovite'"
...     print nom_complet
...
Pim Ella
Pam Ella dit 'la Tigresse Moscovite'
Poum Ella
```

Les mauvaises langues trouvent cette syntaxe *contraignante*. Pourtant, ils ont statistiquement plus de chances de finir avec un rhumatisme des métacarpes: observez leurs claviers, les touches (), {}, et {} sont plus usées.

Pour votre santé, optez pour Python

FIG. 7.2 – Bloc de code

Le document reST suivant...

Python utilise l'indentation pour délimiter les blocs::

```
>>> for nom in ('Pim', 'Pam', 'Poum'):
...     nom_complet = '%s %s' % nom
...     if nom == 'Pam':
...         moscou = " dit 'la Tigresse Moscovite'"
...     print nom_complet
...
Pim Ella
Pam Ella dit 'la Tigresse Moscovite'
Poum Ella
```

Les mauvaises langues trouvent cette syntaxe **constraignante**. Pourtant, ils ont statistiquement plus de chances de finir avec un rhumatisme des métacarpes: observez leurs claviers, les touches (), {}, et {} sont plus usées.

****Pour votre santé, optez pour Python****

¹⁴Fonte dont tous les caractères ont la même largeur.

¹⁵Ils le sont en général de 2 ou 4.

Formatage du texte

Pour mettre en valeur le texte, reST propose les notations suivantes :

- **emphasis**, utilisé pour mettre en valeur des mots du texte, équivalent à *l'italique* et à la balise *em* en html : *texte*;
- **strong emphasis**, équivalent au **gras** et à la balise **strong** en html : **texte**;
- **interpreted text**, utilisé pour les noms propres ou les noms de variables, équivalent à la balise *cite* : ‘texte’;
- **inline literal text**, pour les extraits de code en ligne : “texte”.

Listes

Les deux types de listes majeurs sont les **listes à puces** et les **listes numérotées**. Pour les listes à puces, chaque ligne doit être précédée du signe -, + ou *. Les listes numérotées sont quant à elles directement précédées d'un numéro ou d'une lettre, suivi d'un point et d'un espace. Le signe # permet une numérotation automatique.

Le document suivant...

Kit de survie pour un trajet en train:

- + un laptop, batterie pleine
- + deux pommes
- + une Pink Lady
- + une Golden
- + une bouteille d'eau gazeuse
- + une bouteille d'eau gazeuse
- + un Fluide Glacial

Todo list pendant le trajet:

- 1. Manger la Golden
- 2. Écrire un chapitre du livre sur le laptop
- 3. Boire un coup
- 4. Manger la Pink
- 5. Lire Cosmic Roger
- 6. Roupiller

En arrivant à la gare:

1. Se faire réveiller en sursaut
2. Remballer les affaires en quatrième vitesse
3. Sortir du train
4. Verser une larme en le voyant repartir: le Fluide Glacial étant resté à bord...

FIG. 7.3 – Listes en reST

En arrivant à la gare:

- #. Se faire réveiller en sursaut
- #. Remballer les affaires en quatrième vitesse
- #. Sortir du train
- #. Verser une larme en le voyant repartir: le Fluide Glacial étant resté à bord...

...donnera le rendu de la figure 7.3.

Plusieurs remarques sur le résultat :

- Lors d'un rendu HTML, le décalage pour la sous-liste des pommes dépend de la feuille de style.
- Il est nécessaire pour les sous-listes de les entourer d'une ligne vide, pour que l'interpréteur les prennent en charge.

Kit de survie pour un trajet en train:

- un laptop, batterie pleine
- deux pommes
 - une Pink Lady
 - une Golden
- une bouteille d'eau gazeuse
- un Fluide Glacial

Todo list pendant le trajet:

1. Manger la Golden
2. Écrire un chapitre du livre sur le laptop
3. Boire un coup
4. Manger la Pink
5. Lire Cosmic Roger
6. Roupiller

Tableaux

Il y a deux techniques pour créer les tableaux. La première consiste à créer la garniture complète (*Grid table*) mais est fastidieuse à faire et à maintenir. La deuxième (*Simple table*) utilise une structure plus

légère¹⁶ :

```
+-----+-----+-----+
| Header 1 | Header 2 | Header 3 |
+-----+-----+-----+
| body row 1 | column 2 | column 3 |
+-----+-----+-----+
| body row 2 | Cells may span columns. |
+-----+-----+-----+
| body row 3 | Cells may | - Cells |
+-----+-----+-----+
| body row 4 | span rows. | - contain |
|               | - blocks. |
+-----+-----+-----+
```

- un rendu transparent (*fold-in*) lorsque la page reST est exportée en HTML : les liens deviennent clickables grâce à la balise *a*;
- un rendu explicite (*call-out*), où l'URL est visible en bas de page.

Pour cette raison, il convient de regrouper l'intégralité des liens d'un document à la fin de ce dernier. Le texte à lier est à écrire en *interpreted text*, suivi d'un espace souligné. L'URL notée en bas de document est précédée de la séquence .._`texte à lier` : *url*.

Voici un exemple de liens :

You êtes ‘jeune et beau’ ? Vous avez envie de faire de votre vie un enchaînement de succès, d’être adulé par tous ?

Python est la solution.

Connectez vous tous les matins sur ‘le site de l’Afpy’, pour maîtriser le langage des gagnants.

```
... - 'jeune et beau':
http://fr.wikipedia.org/wiki/Jeunisme
.. _`Le site de l'Afpy` : http://afpy.org
```

Il est également possible d’insérer directement le lien à côté du texte, mais cette notation est à proscrire : les regrouper en bas de document est une bonne pratique pour éviter les redondances et réunir les références.

Hyperliens, bas de page et autres éléments

Pour enrichir les textes en références, reST propose principalement :

- des hyperliens externes ;
- des hyperliens internes ;
- des notes de bas de page ;
- des fields ;
- des directives ;
- et enfin des substitutions.

Hyperliens externes Pour les hyperliens externes, reST propose une notation qui permet deux styles de rendus :

¹⁶Exemple tiré de l'aide en ligne de reST.

Voici un exemple de document avec liens internes :

```
=====
Le label Ninja Tune
=====
```

Fondé par le groupe Coldcut, Ninja Tune regroupe des artistes de musique électronique mais également de hip-hop par le biais, de son sous-label spécialisé ‘Big Dada’¹⁶.

Big Dada
=====

Ce label est orienté hip-hop et regroupe des artistes comme:

- Roots Manuva
- Spank Rock
- TTC
- Diplo

D'autres artistes Ninja:

.. _artistes:

- Dj Vadim
- DJ Food
- Treva Whateva
- ColdCut
- Amon Tobin

La ligne .. _artistes : est transformée en balise *ja name="artistes"*. Notons qu'il n'est pas nécessaire de faire de l'interprétation lorsqu'un seul mot forme le lien.

Notes de bas de page Les notes de bas de page sont des liens internes dont le texte est un numéro ou le caractère # pour une numérotation auto. La deuxième option est la plus pratique, pour ne pas avoir à gérer un changement fastidieux de numérotation lorsqu'une note est insérée ou supprimée dans le document.

Exemple de notes (rendu figure 7.4) :

La série Futurama [#]_, créée par Matt Groening[#]_ est considérée comme une de ses meilleures œuvres. Elle a

La série Futurama [1], créée par Matt Groening [2] est considérée comme une de ses meilleures œuvres. Elle a pourtant été arrêtée au bout de la quatrième saison par la Fox pour des problèmes d'audience mais devrait reprendre prochainement.

- [1] Diffusée en France sur Canal+
[2] Le créateur des Simpsons

FIG. 7.4 – Notes de bas de page à numérotation automatique

pourtant été arrêtée au bout de la quatrième saison par la Fox pour des problèmes d'audience mais devrait reprendre prochainement.

- .. [#] Diffusée en France sur Canal+
- .. [#] Le créateur des Simpsons

Fields Les fields sont des métadonnées ajoutées au document, qui enrichissent les informations. Ils sont automatiquement mis en page par le moteur.

- :Date: 2007-03-01
:Author: Tarek
:Abstract: Ce document ne parle de rien.

Rien.

Directives Les directives permettent d'ajouter des commandes interprétables directement dans le texte sans avoir à modifier la syntaxe de reST. C'est la technique la plus simple pour étendre les capacités de l'interpréteur. Les directives les plus utiles¹⁷ fournies avec reST sont :

- **image** : qui permet d'afficher une image;
- **figure** : qui permet d'afficher une image avec une légende et une mise en page avancée;
- **contents** : qui permet d'afficher un index automatique.

¹⁷Liste complète ici : <http://docutils.sourceforge.net/docs/ref/rst/directives.html>.

contents *contents* est à placer au début du document, et permet de construire automatiquement un index cliquable. Le titre peut être passé en paramètre.



FIG. 7.5 – Directive *image* à l'action

Les directives sont toutes construites suivant la syntaxe : .. *directive* :: *paramètres*.

image La directive *image* prend en paramètre le nom du fichier, qui doit être dans le chemin de recherche au moment de la compilation.

Exemple (rendu figure 7.5) :

Le logo de l'afpy est :

.. image:: afpy.jpg

figure La directive *figure* est similaire à *image* mais permet d'ajouter une légende sous la forme d'un bloc, et de définir la taille de l'image, via le paramètre *scale* ou *figwidth*. Le paramètre *align*¹⁸ permet quant à lui de définir la position horizontale.

Exemple :

Le logo de l'afpy est :

```
.. figure:: afpy.jpg
:scale: 75
:align: center
```

Inspiré du symbole de la francophonie

¹⁸Qui peut prendre la valeur *left*, *right* ou *center*.

Les séries TV à emporter sur Mars

Table des matières

- Futurama
- Battlestar Galactica
- Alias
- Six Feet Under

Futurama

Battlestar Galactica

Alias

(uniquement pour Jiminy)

Six Feet under

FIG. 7.6 – Les trucs à emporter pour les voyages un peu longs

Vous savez ce qu'un ☺ qui voyage sur le dos d'une ☺ dit ?

- Yeeeehaaaaaaa !

FIG. 7.7 – Tortue GTI Turbo-D

Vous savez ce qu'un `|escargot|` qui voyage sur le dos d'une `|tortue|` dit ?

- Yeeeehaaaaaaa !
 .. `|escargot| image:: escargot.jpg`
 .. `|tortue| image:: tortue.jpg`

Dans cet exemple, les mots seront remplacés par les images correspondantes (voir figure 7.7).

7.3.2 Les doctests

Le *bittere programming*, introduit par Donald Knuth [8], fusionne le code source d'un programme et sa documentation en un seul et

même document. L'intérêt est d'augmenter la qualité des deux grâce à cette proximité : la documentation est écrite avec le code sous les yeux, et inversement.

Les docstrings

Ce principe a été repris dans Python pour documenter les API via les docstrings : les classes, méthodes et fonctions peuvent être directement annotées. Des programmes comme Epydoc¹⁹ peuvent alors extraire des modules les textes et composer une documentation sur l'API.

Cette proximité immédiate de la documentation facilite aussi les mises à jour lorsque le code est retouché : le développeur modifie en une seule passe les deux et s'assure de ne pas oublier de mettre à jour la documentation. Ce type de *documentation en ligne* s'est naturellement étendu de textes plus complexes, dotés d'exemples d'utilisation des API. Dans l'exemple ci-dessous, un exemple d'utilisation de la fonction est présentée directement dans sa docstring :

```
def accentEtrange(texte):
    """Ajoute un accent étrange à un texte
```

Les r sont doublés, les e suivis d'u.

Exemple:

```
>>> texte = ("Est-ce que tu a regardé la télé "
... "hier ?\n"
... "Il y avait un théma sur les "
... "ramasseurs\n"
... "d'escargots en Laponie, ils "
... "en bavent...")
>>> accentEtrange(texte)

Est-ce queu tu a rrreugarRrdé la téle heurRr ?
Il y avait un théma surRr leus rRamasseurRrs
d'euscarRrgots eun Laponie, ils eun baveunt...
```

Cette technique permet d'internationaliser les applications pour les rendre compatibles avec

¹⁹<http://epydoc.sourceforge.net>

certaines régions françaises.

```
"""
texte = texte.replace('r', 'rRr')
print texte.replace('e', 'eu')
```

Les exemples, appelés *doctests*, sont écrits sous la forme de sessions, comme si le code avait été testé dans le prompt Python. Le module *doctest* de la bibliothèque standard permet ensuite d'extraire puis de lancer ces sessions pour vérifier qu'elles fonctionnent : chaque commande précédée de *>>>* est exécutée puis le résultat retourné comparé à celui présenté dans le texte. En cas d'erreur, une exception de type *AssertionError* est levée, de la même manière que les tests unitaires.

La fonction précédente peut être écrite dans un module dans lequel la fonction *testmod* du module *doctest* est appelée :

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

def accentEtrange(texte):
    """Ajoute un accent étrange à un texte

Les r sont doublés, les e suivis d'u.
```

Exemple:

```
>>> texte = ("Est-ce que tu a regardé la télé hier ?\n"
...     "Il y avait un théma sur les \"\n"
...     "ramasseurs\"\n"
...     "d'escargots en Laponie, ils \"\n"
...     "en bavent...")\n
>>> accentEtrange(texte)

Est-ce que tu a rReugarRrdé la téle hieurRr ?
Il y avait un théma surRr leus rRramasseurRrs
d'euscarRrgots eun Laponieu, ils eun baveunt...
```

2 items had no tests:

__main__

1 items passed all tests:

2 tests in __main__.accentEtrange

2 tests in 3 items.

2 passed and 0 failed.

Test passed.

Le paramètre *-v* permet de détailler l'exécution du test, qui reste le cas échéant silencieux lorsque les tests passent. En modifiant légèrement le résultat attendu pour faire échouer le test, on obtient une sortie qui présente la différence entre le résultat attendu et celui obtenu :

```
dabox: ~/Desktop/rst tarek$ python accent.py -v
*****
File "accent.py", line 14, in __main__.accentEtrange
```

Failed example:

```
accentEtrange(texte)
```

Expected:

Est-ceu queu tu a rrreugarrRrdé la télé hieurrR ?

je m'insère ici pour que le test échoue

Il y avait un théma surRr leus rRramasseurRs d'euscarRrgots eun Laponieu, ils eun baveunt...

Got:

Est-ceu queu tu a rrreugarrRrdé la télé hieurrR ?

Il y avait un théma surRr leus rRramasseurRs d'euscarRrgots eun Laponieu, ils eun baveunt...

1 items had failures:
1 of 2 in __main__.accentEtrange

Test Failed 1 failures.

Ces docstrings peuvent également venir enrichir les campagnes de tests unitaires classiques, puisque le module *doctest* fournit des classes dérivées des classes *TestCase* et *TestSuite* du module *unittest* : les *suites* classiques peuvent combiner les deux types de tests. Le module de code est dans ce cas importé dans un module de tests qui retourne une *suite*, comme dans l'exemple 6.2.1.

```
import unittest
import doctest

# importation du module contenant des doctests
import accent

def test_suite():
    suite = unittest.TestSuite()
    suite.addTest(doctest.DocTestSuite(accent))
    return unittest.TestSuite(tests)

if __name__ == '__main__':
    unittest.main(defaultTest='test_suite')
```

Les doctests séparés

Lorsque les tests ajoutés dans les docstrings deviennent importants, un problème de visibilité apparaît : il devient de plus en plus

difficile de faire la distinction entre le code réel et le code des tests. Il est possible pour éviter ce problème de déporter dans un fichier texte l'ensemble des docstrings pour constituer un seul et même docstring.

La proximité évolue alors vers un niveau plus haut : le module.

Ce changement provoque une modification de la façon d'écrire les doctests : ils ne sont plus modulaires à chaque fonction et méthode de la classe mais globaux, et deviennent une *histoire*, celle du module. Ces textes se rapprochent alors d'une documentation cohérente, réalisable sans effort d'extraction et réellement proches du code.

Chaque module peut donc être accompagné d'un fichier texte, contenant une alternance de textes et de séquences de code, qui présentent le fonctionnement et qui offrent des tests unitaires.

En appliquant le principe d'alternance entre conception de (doc-)tests et de code, la perte de proximité immédiate est compensée. En reprenant l'exemple précédent, le doctest peut être écrit dans un fichier texte, et bénéficier du format reST :

```
=====
Le module accent
=====

Ce module fournit une fonction 'accentEtrange' qui permet d'ajouter un accent à un texte:

    >>> texte = ("Est-ce que tu a regardé la "
               "télé hier ?\n"
               "... "
               "Il y avait un théma sur "
               "les ramasseurs\n"
               "d'escargots en Laponie, ils en "
               "bavent... ")
    >>> from accent import accentEtrange
    >>> accentEtrange(texte)
Est-ceu queu tu a rrreugarrRrdé la télé hieurrR ?
Il y avait un théma surRr leus rRramasseurRs d'euscarRrgots eun Laponieu
```

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.

Ce fichier joue alors un double rôle : celui de tester et celui de documenter. Ces principes permettent de modifier la façon de concevoir le code : il devient possible en appliquant les principes du TDD²⁰ de faire du *design by document*, c'est-à-dire d'écrire le doctest avant le code correspondant, en commençant par les exemples et leurs explanations. Cette technique est présentée en détail dans la section 7.5.

7.4 Organisation de la documentation

La documentation d'un projet logiciel est réalisée à trois niveaux :

- au niveau de chaque module, on parle alors de **documentation unitaire**;
- au niveau des paquets;
- au niveau du projet global.

7.4.1 Documentation unitaire

La documentation unitaire se concentre sur une sélection de modules d'un paquet, et fournit le mode d'emploi de chacun.

Quels modules faut-il documenter ?

Dans un paquet, il existe deux types de modules :

- les modules **publics**, qui peuvent être utilisées par d'autres paquets;
- les modules **privés**, qui ne servent qu'en interne au paquet.

Ce sont les modules publics qui doivent être documentés en priorité, puisqu'ils sont manipulés par des éléments externes. Le mode d'emploi doit se concentrer uniquement sur le module et ne jamais mentionner un autre module public. Si c'est le cas, le découpage n'est pas correct.

Quel contenu ?

Le bon ton pour un document unitaire consiste à se mettre à la place d'un développeur qui doit l'utiliser : comment doit-il s'y

prendre ? Sous forme de doctest, le document doit fournir un jeu d'exemples qui peut être copié-collé directement. La technique la plus simple est de lancer un prompt Python, avec le module cible dans le chemin de recherche, et de construire un exemple d'utilisation, qui sera copié-collé dans le document.

L'exemple ci-dessous a été écrit pour documenter le module smtplib de la bibliothèque standard :

```
=====
```

Le module ‘smtplib’ permet d’envoyer des emails au format RFC 2822 [#] – via la fonction ‘sendmail’.

Le message suivant est construit en respectant la syntaxe 2822, où chaque en-tête est présentée sur une ligne:::

```
>>> to_addr = ['tarek@ziade.org']
>>> from_addr = 'tarek@ziade.org'
>>> msg = """\
... From: %s
... To: %s
... Subject: Bonjour
...
... Bonjour, je suis un milliardaire au Niger,
... envoyez moi votre CB sous pli discret
... avec le code, et je vous crédite de
... 10 000 000 000 dollars votre compte
... et on devient copains
... %% (from_addr, ", ".join(to_addr))
```

Le message est ensuite envoyé via ‘smtplib’, qui utilise le protocole telnet:::

```
>>> import smtplib
>>> server = smtplib.SMTP('smtp.neuf.fr')
>>> server.sendmail(from_addr, to_addr, msg)
{}
```

```
>>> server.quit()
```

‘sendmail’ retourne un dictionnaire contenant les éventuelles erreurs. Dans l’exemple 1, l’envoi a été correctement effectué et un dictionnaire vide est renvoyé.

```
.. [#] voir http://www.faqs.org/rfcs/rfc2822.html
```

Comment lancer les tests ?

Ce document au format reST est conventionnellement placé dans un sous-répertoire *doc*, et peut être exécuté par le biais du module *doctest*, via la classe *DocFileTest*. Cette dernière prend comme paramètres principaux lors de son instantiation :

- le chemin du fichier contenant les doctests;
- **optionsflags** : les options d’exécution du doctest, à *None* par défaut ;
- **globs** : un dictionnaire contenant un environnement d’exécution à {} ;
- **setUp** et **tearDown** : les fonctions lancées au début et à la fin du test, par défaut à *None* ;
- **module.relative** : détermine si le chemin du fichier fourni est un chemin absolu ou relatif. À *True* par défaut.

optionsflags Lorsqu’un doctest est exécuté, *optionsflags* peut contenir un certain nombre de drapeaux pour définir le mode de fonctionnement. Les drapeaux les plus importants sont :

- **ELIPSIS** : prend en charge les ellipses ;
- **NORMALIZE_WHITESPACE** : ignore les différences d’espaces dans les résultats. Permet de ne pas se soucier d’éventuels problèmes d’espaces de fin de ligne ;
- enfin **REPORT_ONLY_FIRST_FAILURE** permet d’arrêter l’exécution lorsqu’une erreur est rencontrée. Sans ce drapeau la suite des exemples est exécutée.

La règle d’or pour la mise en place d’un environnement de test est de toujours revenir à l’état initial à la fin du test.

Les ellipses²¹ permettent de faire fonctionner les exemples susceptibles de renvoyer des résultats différents à chaque exécution. Dans

la séquence ci-dessous, l’adresse mémoire changera à chaque nouvelle exécution :

```
>>> object()
<object object at 0x23460>
```

Remplacer l’adresse mémoire par une ellipse, permet de résoudre cette problématique :

```
>>> object()
<object object at 0x...>
```

La technique est aussi utilisée pour couper un retour trop long, comme une page html. La seule restriction étant que l’ellipse ne peut pas être utilisée au début de la séquence de retour.

globs Permet de fournir un contexte d’exécution au doctest, de manière équivalente aux variables globales dans un environnement classique. Ce paramètre optionnel est utilisé lorsqu’il est nécessaire de fournir au doctest des variables qui ne peuvent pas être récupérées directement, ou pour masquer du code de préparation qui polluerait le doctest. *globs* est un dictionnaire où chaque clé représente le nom de la variable.

setUp et **tearDown** De la même manière que les tests unitaires, il est possible d’exécuter une fonction avant et après l’exécution du doctest. L’intérêt est de pouvoir gérer le *test fixture*, c’est-à-dire l’environnement requis pour les tests. Un exemple typique est la mise en place d’une structure de fichier ou d’une base de données de tests : elle est initialisée dans la fonction *setUp* et supprimée ou nettoyée dans *tearDown*.

Le script *gettests.py*, présenté dans la section *scripts* du site <http://programmation-python.org>, simplifie la construction d’une suite de tests en accumulant les doctests trouvés dans un répertoire. Dans une arborescence classique de paquet, où les tests sont dans un sous-répertoire *tests* et les documents dans un sous-répertoire *doc*, il

²¹Signe “...” en Python.

peut être utilisé dans un module dédié au lancement des doctests²², qui peut ressembler à l'exemple suivant :

```
import unittest
import os
# gettests est dans le sous-rep 'tests',
from gettests import doc_suite
# ce fichier est dans le sous-rep 'tests',
# ce fichier est dans le sous-rep 'tests',
# tests-dir = os.path.dirname(__file__)
tests-dir = os.path.dirname(tests-dir)[0]
package-dir = os.path.split(tests-dir)[0]
doc-dir = os.path.join(tests-dir, 'doc')
```

def test_suite():

```
# renvoie tous les doctests contenus dans
# le sous-repertoire 'doc' du paquet
return doc_suite(doc-dir)
```

```
if __name__ == '__main__':
```

```
unittest.main(defaultTest='test_suite')
```

Ce script, placé dans le répertoire *tests* de chaque paquet, peut être appelé comme un module de tests unitaires, et servir dans les campagnes de tests :

```
dabox:tarek$ test -m test_docs
```

```
Running tests via: python /usr/bin/test.py -v \
```

```
-m test_docs
```

```
Running unit tests:
```

```
Running:
```

```
.....
```

```
Ran 14 tests with 0 failures and 0 errors in 2 seconds.
```

7.4.2 Documentation d'un paquet

En complément des documents unitaires, un paquet qui est voué à être distribué doit être garni d'un certain nombre d'éléments, qui donnent des informations globales et permettent l'installation du paquet, à savoir :

- README.txt contient une présentation du paquet. Peut contenir des exemples de code;

²²Et porter le nom de test_docs.py par exemple.

README.txt global

Le fichier *README.txt* global est le premier fichier ouvert par l'utilisateur. Il doit être court²⁴ et :

- expliquer dans un premier chapitre l'objectif du paquet;
- lister ensuite les documents du répertoire *doc* avec une phrase courte pour chacun;
- faire référence au fichier *INSTALL.txt* si nécessaire;
- présenter éventuellement des exemples de code de très haut niveau;
- fournir des informations pratiques, comme le site web du projet ou du tracker, l'e-mail du mainteneur.

```
=====
zorgbrt
=====
```

```
=====
```

'zorgbrt' [#]_ est un paquet qui permet de remplir automatiquement le frigidaire. Issu de la technologie

spaciale, il se connecte à internet et passe commande, après avoir scanné le contenu du frigidaire [#]_.

Les modules notables du paquet sont :

- + chouf: scan le contenu du frigidaire, voir

²³GPL, LGPL, BSD, etc.

²⁴Un écran, voire deux.

- doc/chouf.txt.
- + heho: envoi les commandes, voir doc/heho.txt

Pour installer ‘zorgbrt’, veuillez vous référez au fichier INSTALL.txt.

Utiliser ‘zorgbrt’ est enfantin, il permet de scanner le frigidaire::

```
>>> from zorgbrt import chouf
>>> chouf.porte()
[‘confiture moisie’, ‘4 oeufs’,
 ‘tube harissa, reste 12%’]
>>> chouf.congeleur()
[‘demi-langouste, nouvel an 2001’,
 ‘12 steak hachés, 15% de M.G.’]
```

Et de passer des commandes::

```
>>> from zorgbrt import heho
>>> heho.bons_trucs()
[‘commande passée: mix de légumes et fruits’]
>>> heho.du_gras()
[‘1 chorizo Kipik’, ‘2 rillettes PaléMenvaleur’]
```

En cas de problème avec ce module [#]_, une seule ressource: <http://zorgbrt.com>.

```
.. [#] prononcez ‘zorbert’
.. [#] en option, nécessite la prise zorgbrt-fridge 2000
.. [#] explosion de frigidaire, note de 30K euros en VPC
```

Lorsque ce fichier contient des exemples de code susceptibles d'être exécutées par le script de tests, il est déporté dans le répertoire doc et un deuxième fichier README est créé à la racine avec pour seul contenu une référence vers le premier.

INSTALL.txt et setup.py

La bibliothèque standard fournit un module spécialisé dans l'installation des paquets : *distutils*. Ce dernier prend en charge :

- l'éventuelle compilation d'extensions C;
 - la recopie des modules dans le répertoire d'extension de Python;
 - la mise en place de liens symboliques lorsque le paquet fourni des commandes systèmes.
- Cette mise en place s'effectue par la création d'un fichier *setup.py* placé à la racine du paquet et contenant le paramétrage nécessaire à *distutils* pour déployer les éléments sur la machine cible. *INSTALL.txt* se résume alors à ces quelques phrases :

```
=====
Installation
=====
```

Pour installer le paquet, placez vous dans le répertoire sous un shell, et lancez la commande suivante, avec les droits de super-utilisateur::

```
$ python setup.py install
```

Si vous avez plusieurs Python installés, il suffit d'appeler le script ‘setup.py’ avec le Python cible.

Python propose en outre, à la manière de *CPAN* pour Perl, un dépôt central de paquets²⁵ appelé *CheeseShop*²⁶. L'extension *setup-tools*, basée sur *distutils*, permet de configurer et de déployer les paquets vers ce dépôt et de créer des *eggs*. Cette technique, qui est devenue un standard²⁷, peut être utilisée pour concevoir les fichiers *setup.py* des paquets.

7.4.3 Documentation d'un projet

Un projet Python peut être vu comme un *bundle*, c'est-à-dire un regroupement de paquets pouvant provenir de différentes sources. La consolidation se fait en général par un paquet en charge d'instancier

²⁵<http://python.org/pypi>

²⁶En référence au célèbre sketch des Monty Python.

²⁷<http://peak.telecommunity.com/DevCenter/PythonEggs>

et de manœuvrer les autres, et un paquet en charge de la configuration.

Une application web contiendra par exemple :

- les paquets du framework web utilisé ;
- un paquet d'accès aux données, comme *SQLAlchemy* ;
- un paquet d'accès aux annuaires LDAP ;
- les paquets *maison* de l'application, à savoir :
 - un paquet de configuration ;
 - un paquet *orchestrateur* ;
 - un paquet métier en charge de manipuler les données ;
 - un paquet en charge de générer les écrans.

Le déploiement de ce *bundle* peut être fait en suivant plusieurs stratégies, comme la création d'une seule archive, la diffusion via *CheeseShop*, ou par le biais d'un gestionnaire de version. Au final, un projet peut être vu comme une arborescence de paquets.

Documenter un projet consiste à écrire des documents de très haut niveau, regroupés dans un répertoire *doc* général et dont le contenu est transverse à l'ensemble des paquets. La difficulté majeure étant de réussir à conserver ces documents à jour en fonction des évolutions des paquets sous-jacents, l'organisation de cette documentation générale est toujours délicate.

Le découpage suivant s'avère être le plus efficace, et résiste bien aux grands projets :

- chaque fonctionnalité du projet est présentée à travers un **tutoriel** ;
- les détails d'implémentation de chaque tutoriel sont déroulés dans des **recettes** ;
- un **glossaire** regroupe tous les termes techniques importants.

Le découpage suivant s'avère être le plus efficace, et résiste bien aux grands projets :

- contenir très peu d'exemples de code, voire aucun ;
- utiliser un maximum de diagrammes et de tableaux synthétiques ;
- être compréhensible par les non-développeurs ;
- faire parti d'un *plan de lecture*, c'est-à-dire proposer des références à d'autres documents à lire en amont et en aval ;
- se concentrer sur un et un seul sujet, et ce quel que soit le nombre de paquets concernés ;
- se limiter à deux écrans²⁹.

Le patron utilisé par un tutoriel peut être le suivant :

- un **titre**, sous la forme d'une phrase courte ;
- un **résumé**, de deux à trois phrases ;
- un **prérequis**, qui détermine le lectorat cible et le niveau requis, ainsi que les documents à lire avant celui-ci ;
- un **index**, lorsque le document contient deux sections ou plus ;
- un **corps**, limité en nombre de sections et en taille (2 écrans) ;
- une liste de **références**, *internes* comme d'autres tutoriels, ou *externes* comme des adresses web ou des livres ;
- une liste de **tutoriels** qui peuvent être lus après ce tutoriel. Cette dernière section permet également d'organiser les documents en graphe, dont chaque chemin est un *parcours documentaire*³⁰ possible. Tracer ce graphe est un bon exercice pour automatisée.

²⁸ 8 000 signes.

²⁹ Un arbre de recouvrement.

Tutoriels

Un tutoriel est un document de très haut niveau qui explique une fonctionnalité du logiciel, comment l'utiliser et parfois comment la modifier. Lorsqu'un projet est voué à être utilisé par une communauté de développeurs susceptibles de le faire évoluer, les tutoriels sont les documents les plus importants : ils doivent permettre de comprendre les rouages du système sans avoir à plonger dans le code, puis offrir des points d'entrée balisés pour creuser d'avantage un sujet précis. Les tutoriels sont également lus par des non-développeurs, que ce soient des chefs de projet, des intégrateurs, des graphistes ou des traducteurs.

Pour jouer pleinement son rôle et satisfaire l'ensemble du lectorat, un tutoriel doit :

- contenir très peu d'exemples de code, voire aucun ;
- utiliser un maximum de diagrammes et de tableaux synthétiques ;
- être compréhensible par les non-développeurs ;
- faire parti d'un *plan de lecture*, c'est-à-dire proposer des références à d'autres documents à lire en amont et en aval ;
- se concentrer sur un et un seul sujet, et ce quel que soit le nombre de paquets concernés ;
- se limiter à deux écrans²⁹.

Le patron utilisé par un tutoriel peut être le suivant :

- un **titre**, sous la forme d'une phrase courte ;
- un **résumé**, de deux à trois phrases ;
- un **prérequis**, qui détermine le lectorat cible et le niveau requis, ainsi que les documents à lire avant celui-ci ;
- un **index**, lorsque le document contient deux sections ou plus ;
- un **corps**, limité en nombre de sections et en taille (2 écrans) ;
- une liste de **références**, *internes* comme d'autres tutoriels, ou *externes* comme des adresses web ou des livres ;
- une liste de **tutoriels** qui peuvent être lus après ce tutoriel. Cette dernière section permet également d'organiser les documents en graphe, dont chaque chemin est un *parcours documentaire*³⁰ possible. Tracer ce graphe est un bon exercice pour automatisée.

mieux organiser les tutoriels et repérer les éventuels manques.

L'exemple ci-dessous est un tutoriel type, imaginé pour un logiciel de paiement en ligne :

Intégration du paiement en ligne Pépale

:abstract:

Ce document explique comment intégrer le paiement Pépale dans une application.

Ce tutoriel est destiné aux intégrateurs de la solution de paiement en ligne. Aucun prérequis n'est nécessaire.

.. contents: Index

Comment configurer Pépale ?

La configuration pour faire fonctionner le paiement est stocké dans une base de données. Le paquet en charge de ce stockage est ‘configurator’. Il permet de définir une valeur pour chaque élément nécessaire au paiement:

- + ‘url’: l’url du site pépale
- + ‘login’: le login du compte pépale
- + ‘password’: le password du compte pépale
- + ‘validated’: l’url du site local, lorsque le paiement a marché
- + ‘failed’: l’url en cas de disfonctionnement

Une fois ces valeurs sauvegardées en base, elles deviennent disponibles aux autres paquets.

Comment se déroule un paiement ?

Le paquet ‘pepale’ récupère les informations dans

la base, et s’en sert en trois phases:

- + une demande de paiement, avec un montant est envoyée à Pépale, qui retourne une url.
- + l’utilisateur procède au paiement sur le site de Pépale.
- + Pépale redirige l’utilisateur sur la page ‘validated’ ou ‘failed’ en fonction du résultat de la transaction.

‘pepale’ est donc piloté par l’application qui lui transmet une demande de paiement avec une somme, puis redirige l’utilisateur sur l’url fournie.

Effectuer le suivi des paiements

Chaque paiement est stocké en base, le paquet ‘dataaccess’ fournit une visualisation des données en lui indiquant la table à afficher, en l’occurrence ‘pepale_paiements’.

Pour aller plus loin...

Les tutoriels suivants apportent des éclaircissements sur des fonctionnalités complémentaires au paiement:

- + Etude statistique de la base: pour étudier les moyens de paiement;
- + Utilisation des événements de workflow: pour envoyer un mail à chaque paiement.

Quelques remarques sur ce tutoriel:

- il est totalement dénué de code;
- il est compréhensible par le commun des mortels;
- chaque section reste relativement succincte et l’ensemble se lit en quelques minutes;
- l’intégrateur prend rapidement connaissance du processus de paiement et des paquets à utiliser.

Les seuls éléments qui manquent pour en faire un document totalement utilisable sont les exemples de code. Ils ne doivent pas être insérés directement dans chacune des sections du document pour éviter que ce dernier soit à la merci d'une modification de code. Si le tutoriel se focalise sur la présentation d'une fonctionnalité sans entrer dans les détails de son intégration, son cycle de modification ralentit. Les exemples de code sont à développer dans un autre type de document : les recettes, qui elles ne contiennent pour ainsi dire que des exemples de code, et sont modifiées beaucoup plus souvent. Ce découplage permet en outre de partager d'un tutoriel à l'autre les mêmes recettes : elles deviennent moins spécifiques à une fonctionnalité mais toujours ciblées sur un seul sujet.

Enfin, les écrivains découvrent le code et son organisation sous un nouvel angle en procédant à ce découpage de la documentation : des recettes qui apparaissent comme très proches lorsqu'elles sont écrites mettent souvent le doigt sur un besoin de refactoring.

Dans chacune des sections présentées, un lien vers la recette correspondante est inséré. L'intégrateur n'a plus qu'à suivre la série de recettes dans le contexte du tutoriel.

Recettes

La recette est l'atome, la brique de base, de la documentation d'un logiciel. Elle offre au développeur un mode d'emploi concis pour effectuer une opération avec le logiciel. Elle est équivalente en termes de construction aux doctests de la documentation modulaire mis à part le fait qu'elle peut concerner plusieurs modules et parfois plusieurs paquets de l'application.

Une recette doit :

- contenir très peu de texte, dont le seul rôle est de faire le lien entre chaque exemple de code, pour une meilleure compréhension ;
- présenter des exemples concrets d'utilisation (voir 7.2.5) ;
- se concentrer sur un et un seul sujet, et ce quel que soit le nombre de paquets concernés ;
- se limiter à deux écrans.

Le patron utilisé par une recette peut être le suivant :

– un titre, sous la forme d'une question ;
 – un corps sans aucune section, et limité à deux écrans ;
 – une liste de recettes qui peuvent être réalisées après celle-ci.

L'exemple ci-dessous est l'une des recettes correspondant au tutoriel précédent :

```
=====
Comment utiliser configurator la sauvegarde des données
=====
```

'configurator' gère la persistance des données dans toutes les bases de données supportées par son connecteur [#]_. Il est paramétré avec une 'uri':

```
>>> from configurator import configurator
>>> db = 'sqlite:///memory:'
>>> engine = configurator.connect(db)
>>> engine.connected
True
```

Toutes les données serialisables, c'est-à-dire qui ne contiennent pas d'éléments comme des threads ou des locks peuvent être sauvegardées dans la base, sous une étiquette :

```
>>> engine.save('ici', ['mes données', 1, 2,
... , 'youpi'])
```

Elles peuvent être ensuite récupérées avec 'get' :

```
>>> engin.get('ici')
['mes données', 1, 2, 'youpi']
...
[#] SQLAlchemy
```

Cette recette est écrite de manière à être réutilisable dans plusieurs tutoriels : elle n'est pas liée à un contexte d'utilisation figé. Rendre une recette *générique* n'est pas non plus l'objectif premier : écrite pour un premier tutoriel, elle peut évoluer ensuite pour être réutilisable dans un deuxième tutoriel.

Glossaire

Le partage des connaissances sur un projet informatique passe par le partage d'un champ lexical commun. Éviter d'utiliser deux termes pour parler d'un même concept est essentiel pour conserver une documentation claire et faciliter la compréhension des développeurs et autres membres du projet. Un glossaire est un bon moyen de centraliser tous les termes techniques du domaine.

Consultée, utilisée et enrichie pendant l'écriture et la lecture des tutoriels et recettes, cette liste de mots est le pivot central de la documentation. Chaque terme est accompagné d'une définition courte³¹ et la liste est ordonnée alphabétiquement.

Voici un exemple de glossaire, qui est un simple fichier texte :

Python: Le langage des gens biens

Postgres: Une base de données libre

SQLAlchemy: mapper relationnel-objet

Lorsque la liste dépasse la centaine de mots, elle peut être organisée en groupe de mots, pour chaque lettre de l'alphabet.

Consolidation

L'ensemble des documents produits peuvent être liés pour constituer une documentation facile à utiliser, que ce soit par le biais d'un site web ou de la génération d'un livre. Sont mis en place des :

- liens vers le glossaire ;
- liens entre les documents ;
- index pour les tutoriels et les recettes.

Liens vers le glossaire

Tous les documents peuvent contenir des termes du glossaire. Un rendu HTML, par exemple, optera pour une balise *ACRONYM* afin d'afficher dans une fenêtre flottante la définition lorsque l'utilisateur passe la souris sur le mot. Une référence vers un lexique sera employée dans un rendu de type papier.

7.5 Développement dirigé par la documentation

La souplesse des doctests, et le besoin d'intégrer l'écriture de la documentation technique au cycle de développement d'une application ont abouti à une nouvelle discipline en termes de programmation agile : le développement dirigé par la documentation (DDD). Cette technique associe le principe de la programmation dirigée par les tests à l'écriture des documents associés au code. Ces derniers prennent la place des tests classiques et le cycle classique de développement,

également être injecté dans une balise *DIV* escamotable, que les lecteurs déplient et replient lors de la lecture.

Liste de tutoriels et Cookbook Présenter un index des tutoriels disponibles offre un outil supplémentaire aux lecteurs pour rechercher des informations. Lorsque le nombre de tutoriels dépasse la trentaine, des mots-clés peuvent leur être associés, et l'index peut les utiliser pour présenter une liste avec des regroupements par thèmes. Mais ce tri sémantique est souvent inutile si l'ensemble de la documentation est indexée par un moteur de recherche classique : si les écrivains ont choisi de bons titres pour les documents et chacune de leurs sections, les informations sont faciles à retrouver.

De la même manière, un index des recettes, ou *Cookbook* offre une vision technique moins fonctionnelle utile aux lecteurs : c'est un annuaire pratique lorsqu'ils savent ce qu'ils veulent faire et les étapes pour y arriver mais qu'ils butent sur *comment* le faire.

PyCommunity

PyCommunity³² est un script Python qui automatise la génération d'une arborescence HTML, en fonction de la documentation reST d'une application. Il s'occupe de générer la page HTML de chaque document et les liens vers les autres documents, ainsi que les pages d'index. Cette automatisation offre un moyen simple de mettre en ligne la documentation produite par les développeurs, et évite les problématiques de mises à jour manuelles de la documentation en ligne.

Liens entre documents

Lorsqu'un tutoriel fait référence à une recette, un lien hypertexte peut être inséré. Pour éviter des allers-retours lors de la lecture du tutoriel le contenu de la recette peut

³¹Une phrase courte.

³²<http://pycommunity.org>

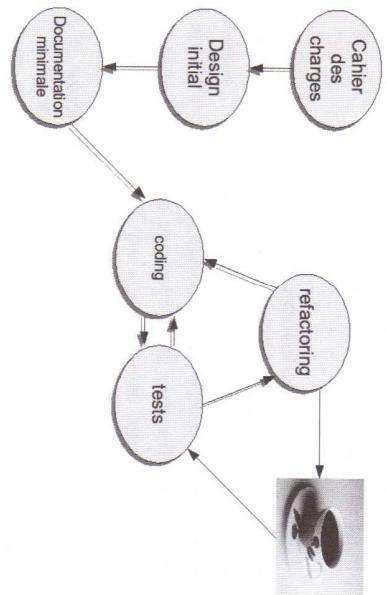


FIG. 7.8 – Cycle classique

présenté dans la figure 7.8, qui laisse la documentation au banc, et la condamne à ne plus évoluer, devient celui présenté dans la figure 7.9, où documentation et code progressent ensemble.

Le DDD est effectué en deux phases :

- une phase initiale de décomposition du cahier des charges ;
- une phase cyclique de construction du code et de la documentation.

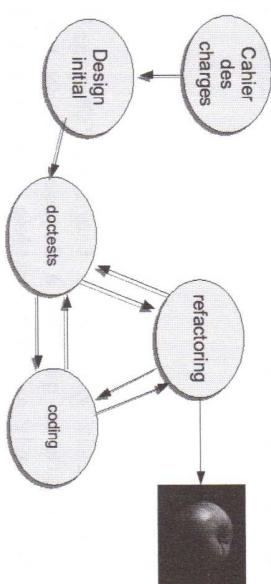


FIG. 7.9 – Cycle DDD

peut un plan pour l'organisation du code en paquets et parfois même en modules. Chaque section peut être reprise pour constituer les premières versions des doctests unitaires.

Voici un (micro) cahier des charges qui définit le fonctionnement d'une application en charge d'afficher un catalogue de vêtements en ligne. Cette application est pour l'instant développée dans un paquet unique. Le cahier des charges est ici au format REST mais aurait pu être dans un format plus classique pour ce genre de document (Writer, Word, etc.).

```
=====  
VetiWeb  
=====
```

stockage

Chaque vêtement est stocké dans une base de données 'sqlite' et dans un dossier du système (pour les images).

Un vêtement est défini par :

- une référence unique (string de 200)
- un libellé (string de 200)
- une photo, stockée sous la forme d'un nom de fichier

³³Voir chapitre 3.

Les spécifications techniques initiales définissent l'architecture³³ qui a été imaginée et une première vision du découpage de l'application. Ce document, s'il est fait convenablement, fournit au dévelop-

L'application fournie des API pour lire et modifier les vêtements.

affichage

L'affichage se fait grâce à un template ‘Cheetah’, combiné à une classe qui met à disposition du template les données. Il n'y a aucun système multipage et tous les articles sont affichés dans une page unique, avec un DIV pour chaque.

Ce document est une amorce pour deux doctests :

- stockage, qui définit les API d'accès aux données;
- affichage, qui définit le générateur de pages HTML.

Voici par exemple le doctest pour le stockage :

stockage

Le module ‘stockage’ est en charge de gérer l'accès aux informations concernant les vêtements. Chaque vêtement est stocké dans une base de données ‘sqlite’ et dans un dossier du système (pour les images).

Un vêtement est défini par:

- une référence unique (string de 200)
- un libellé (string de 200)
- une photo, stockée sous la forme d'un nom de fichier

Le module fourni un mapper ‘vêtement’::

```
>>> from stockage import vetement
```

Des vêtements peuvent être insérés dans la base par son biais, via un objet d'insertion::

```
>>> inserter = vetement.insert()
>>> inserter.execute('REF32', 'T-Shirt Groland',
...                  '/var/media/grd.jpg')
```

Puis lus via un objet de selection::

```
>>> selector = \
...     vetement.select(vetement.c.reference='REF32')
>>> result = selector.execute()
>>> result.fetchall()
[(REF32, 'T-Shirt Groland', '/var/media/grd.jpg')]
```

La collection de doctests ainsi extraits du cahier des charges est le point de départ de la construction du code.

7.5.2 Phase cyclique de construction

Les doctests générées par le cahier des charges sont ensuite complétés d'exemples d'utilisation, pour amorcer l'écriture du code :

Cette nouvelle version permet de développer le module suivant les préceptes du TDD : le code est écrit jusqu'à ce que le doctest fonctionne. L'avantage du procédé est de donner aux développeurs encore plus de recul qu'avec les tests unitaires sur leur code : ils doivent expliquer en français dans le texte le fonctionnement et ne peuvent plus partir dans un développement intellectuellement éloigné de l'objectif final.

Du point de vue de la documentation, l'adaptation continue des doctests en fonction des réalisés du code fournit un socle documentaire parfait. C'est l'approche préconisée par Andreas Rueping^[11] pour la documentation technique : une proximité immédiate. Les doctests séparés offrent en plus de cette proximité, qui est maintenue par le développement par les tests, des textes cohérents et lisibles.

Les dérives possibles

À chaque cycle de modification du code, il faut garder en mémoire le double objectif des doctests : fournir une documentation lisible et permettre de diriger le développement du code. L'équilibre doit être conservé et les documents ne doivent pas contenir du code ajouté pour faire fonctionner les tests³⁴. Dans l'exemple ci-dessous, le document est pollué par du code qui n'apporte rien au lecteur :

```
=====
Client SMTP
=====
```

Avant toute chose, il faut un faux serveur SMTP::

```
>>> from SocketServer import ThreadingTCPServer
>>> class FakeServer(ThreadingTCPServer):
...     def server_until_stopped(self):
...         ...
...         while not abort:
...             print 'working'
...             self.handle_request()
... 
...             smtp_server = FakeServer(port=25)
...             smtp_server.serve_until_stopped()
```

Ensuite, le client peut se connecter en utilisant notre paquet::

```
>>> from smtp_c import SMTPClient
>>> client = SMTPClient('localhost')
>>> client.connect()
'OK'
```

³⁴Test fixture.

La première partie n'a rien à faire dans un tel document, et doit être externalisée soit en test fixture du doctest soit en test unitaire classique.

Les tests qui nécessitent une préparation susceptible de polluer la lecture des doctests doivent être réalisés dans les tests unitaires classiques.

Inversement, la documentation qui ne fournit pas d'exemples concrets et complets sortira graduellement du cycle de développement et les tests basculeront de plus en plus vers les modules de tests unitaires classiques.

L'équilibre est trouvé lorsque les documents suffisent à un tiers pour utiliser le code sans avoir à l'étudier.

7.6 Team writing

Pour obtenir une documentation homogène et agréable à lire, la qualité doit être similaire d'un document à un autre. Le *team writing* permet de renforcer la qualité des documents, par le biais :

- du *document reviewing* ;
- et du *pair writing*.

7.6.1 Document reviewing

Les développeurs n'ont pas tous la même plume et peuvent produire des textes très inégaux en termes de qualité de synthèse et de style. Effectuer des relectures correctives permet de réduire les écarts d'un document à un autre. Lorsque le budget le permet, une personne dédiée à la documentation peut effectuer ce travail. Une rotation sur l'équipe des développeurs suffit le cas échéant.

Le meilleur relecteur est bien sûr le client final.

7.6.2 Pair writing

Comme pour le *pair programming*, où le code est conçu par deux développeurs sur la même machine, le *team writing* est pratiqué par des paires d'écrivains pendant des ateliers d'écriture. Ils prennent chacun un des rôles suivants :

- le spécialiste technique, qui connaît le fonctionnement de l'application à documenter;
- le journaliste, qui de préférence ne connaît pas le fonctionnement, et qui est chargé d'écrire le document.

Le journaliste s'entretient avec le spécialiste pour comprendre comment fonctionne le processus à documenter, et l'écrit en simultané. Le spécialiste surveille le texte écrit et apporte éventuellement des correctifs. Cette technique permet d'obtenir le point de vue du lecteur et produit des documents plus compréhensibles. Lorsque les ateliers d'écritures sont effectués avec plusieurs binômes, les documents produits sont passés à d'autres, qui effectuent une dernière lecture sans avoir participé à la construction du document.

Le *pair writing* est appliqué avant tout aux tutoriaux.

7.7 Ce qu'il faut retenir

La pratique du DDD qui vient renforcer le TDD classique est un véritable avantage de Python pour augmenter la qualité d'un projet, que ce soit pour le code ou la documentation. Bien acceptée et mise en place dès le début, elle permet d'automatiser toute la production documentaire mainstream.

Le prochain chapitre se concentre sur la mise en place d'un environnement de projet basé sur la programmation TDD et DDD.

Chapitre 8

Gestion de projet



Il faut éclater un projet en une multitude de petits projets, et se concentrer sur les priorités au lieu des temps.

Denis Waitley

8.1 Philosophie

Dès qu'un projet concerne plusieurs personnes, une inertie au niveau des actions se met fatallement en place. Ce ralentissement se reflète par une connaissance ciblée de certaines parties du projet par les développeurs, pouvant aller jusqu'à la rétention involontaire d'informations, et par une méthodologie de travail qui n'est pas toujours adaptée à un développement en groupe. Les méthodes présentées dans ce chapitre permettent de lutter contre cette stigmatisation des équipes et de brasser au maximum les informations et les connaissances. Mises en place dès le début, elles offrent des outils efficaces et une philosophie en accord avec la programmation agile.

Le chapitre met en œuvre une solution technique de gestion de projet complète basée sur les points suivants :

- visibilité continue ;
- propriété collective ;

- revues de code ;
- gestion centralisée du code ;
- cycle de développement itératif.

8.1.1 Visibilité continue

Tous les intervenants, que ce soit **les développeurs**, **les intégrateurs**, **les commerciaux**, doivent avoir une visibilité continue sur le projet. Cette fenêtre ouverte est le liant entre les différentes activités et la première brique de communication entre les membres des équipes.

Échange à la pause café entre Julien, chef de projet, et Ulysse, développeur :

- Je viens d'apprendre que tu avais passé 4 jours sur le développement d'un gestionnaire d'impression, pendant mon déplacement à l'étranger.
- Oui, c'était tonflu d'ailleurs, rétorque Ulysse, fier de son petit moteur. J'en avais besoin pour la partie statistiques.
- Bon et celui que Michel a codé le mois dernier sur la partie serveur, on le jette aux orties ?
- Ah ! Mais je ne savais pas qu'il en avait fait un !
- Tu peux voir avec lui pour que vous mettiez en commun vos travaux ?

Les développeurs doivent pouvoir sentir le cœur du logiciel battre, et observer en continu le flux des modifications qui lui sont apportées, par le biais d'indicateurs simples.

En réunion d'avancée globale :
Suite à une modification des API du noyau de l'applicatif, l'équipe d'intégration a passé deux semaines à répercuter ces changements. S'ils avaient été mis au courant avant, ils auraient pu être proactifs et préparer le terrain.

Face au client :
Je ne peux pas vous dire dans l'immédiat si Fred a travaillé sur les modifications de base, il est en vacances et ne rentre que lundi. Je vous enverrai un mail dès son retour.

8.1.2 Propriété collective

La notion de propriété collective est fondamentale et son importance s'intensifie proportionnellement au nombre de développeurs. Elle part du principe qu'une portion de code n'est pas associée à une personne en particulier et que chaque développeur peut être amené à travailler sur n'importe quelle partie. Cette approche présente plusieurs avantages :

- **Les développeurs prennent plus soin du code** qu'ils coïcient : il est susceptible d'être repris par d'autres personnes.
- **Le code est retouché et relu par plusieurs personnes** : sa qualité augmente.
- **Les développeurs acquièrent une expertise globale** : chacun peut intervenir sur toute partie du projet.

La rotation fréquente des ressources sur les différentes parties du logiciel augmente cette propriété collective, mais doit être faite en conservant à l'esprit le rythme de développement du projet. En effet, la conception d'un composant est en général à la charge d'une ou plusieurs personnes qui vont développer une expertise dans le domaine concerné, et être très productives. Il est nécessaire de laisser ce groupe gérer le composant jusqu'à ce qu'il soit fonctionnel avant d'amorcer une rotation.

Quoiqu'il en soit, en termes de gestion de risque, une portion d'un projet ne doit jamais être associée à une seule personne : si elle se tenir informés des évolutions.

fait enlever par des extraterrestres, ou décide de partir en pèlerinage au Tibet, le projet est en danger.

8.1.3 Revue de code

Dans le même esprit que la propriété collective, la revue de code permet d'augmenter la qualité technique du projet de manière drastique. Elle s'effectue de manière continue, ou ponctuellement.

Revue de code continue

Grâce à des outils d'observation qui permettent à tous les développeurs de suivre les modifications effectuées, il est possible de faire une revue de code en temps réel. Effectivement, cet exercice qui est l'apanage du *pair programming*, où un développeur observe un autre développeur écrire le code, est souvent remplacé par des listes de diffusion spécialisées. Chaque développeur du projet reçoit en continu des mails contenant les modifications. Il peut réagir directement lorsqu'il voit passer du code qui lui paraît problématique.

Échange de mails suite à une modification :

Attention, ne jamais mettre de types mutables en défaut de paramètres.

Joseph.

Original message :

```
Author: omaley
Date: Thu Nov 24 22:11:29 2005
New Revision: 119
=====
--- trunk/src/tests/__init__.py (original)
+++ trunk/src/tests/_init__.py Thu Nov 24 2005
@@ -31,7 +31,6 @@
 $ python setup.py test
@@ -46,33 +45,26 @@
+def doctest(tests=[]):
+    """Runing tests.
+    for test in tests:
```

Revue de code ponctuelle

Organiser des rendez-vous réguliers entre développeurs pour effectuer des revues de code permet de se concentrer sur cette tâche en mettant de côté les autres activités. Dans ce cas, de réels objectifs doivent être énoncés, sous peine de perdre du temps à relire inutilement le code. Un module trop lent, difficile à déboguer, ou affichant un score médiocre dans les outils d'assurance qualité, est un bon client pour ces rendez-vous.

Les revues ponctuelles sont souvent faites dans le cadre des *bug days*, journées complètes dédiées aux corrections de bugs. Ces rendez-vous sont organisés avant des releases, ou des démos. Durant ces journées, toutes les tâches de développement sont arrêtées et les développeurs sont chargés de corriger un maximum de bugs, et d'effectuer des refactorings lorsque ces derniers restent simples.

Ces journées sont aussi l'occasion de réunir toute l'équipe autour du projet et autour d'un objectif à court terme commun et très concret.

Bug day #4

```
Objectif : release de la version 2.2
-
- Correction des bugs marqués 2.2 dans le tracker
- Optimisation du connecteur LDAP (trop lent)
- Code review général sur BDManager
```

Réserver une place pour les revues de code dans ces journées est un bon moyen d'augmenter la qualité de la base de code.

```
+ doc_file_path = os.path.join(doc_folder, test)
+ suite.addTest(doctest.DocFileSuite(test))
+
-----
```

Atomisator-commits mailing list

Ces échanges sont aussi l'occasion pour l'équipe de monter en compétence de manière collective.

8.1.4 Gestion centralisée du code

Concevoir une application nécessite d'effectuer des centaines, voire des milliers de petites modifications sur le code source. Ces changements s'opèrent sur des laps de temps qui peuvent s'étaler sur plusieurs mois, et souvent plusieurs années. La première technique qui vient à l'esprit d'une équipe en charge de développer une application est de conserver sur un serveur centralisé, accessible à tous, le code source courant de l'application. Des copies sont ensuite réalisées par le développeur sur son poste local qui met à jour le code commun lorsqu'il a terminé sa modification. Cette technique est très dangereuse et peut aboutir à des scénarios catastrophes. Voici quelques exemples réels observés dans des entreprises :

Manu a posté sa modification mais a oublié de prévenir les autres développeurs. Résultat : Julien vient de poster ses modifications et écraser celles de Manu.

Julien a travaillé pendant deux semaines sur une modification complexe. Il a passé une demi-journée à mettre à jour son code, avec la toute dernière version courante, et poste le tout. Deux jours plus tard, l'équipe découvre qu'il faut revenir en arrière car les modifications entraînent des régressions imprévues. Ce retour en arrière est difficile.

Marie doit effectuer une modification similaire à celle que Julien a faite la semaine dernière. Elle lui demande de lui expliquer. Ce dernier se souvient du principe mais ne sait plus comment le faire. Marie ne part pas de zéro, mais n'a pas d'exemple concret. Manu, Julien et Marie dépriment.

Manu a posté sa modification mais a oublié de prévenir les autres développeurs. Julien tente de poster les siennes, le système refuse et lui demande de se mettre à jour avant. Julien a travaillé pendant deux semaines sur une modification complexe. Il a passé une demi-journée à mettre à jour son code, avec la toute dernière version courante, et poste le tout. Deux jours plus tard, l'équipe découvre qu'il faut revenir en arrière car les modifications entraînent des régressions imprévues. Ils récupèrent la version antérieure aux modifications, et réappliquent sur cette version les modifications des autres développeurs.

Marie doit effectuer une modification similaire à celle que Julien a faite la semaine dernière. Elle lui demande de lui expliquer. Ce dernier se souvient du principe et retrouve dans le système de gestion de code cette modification. Il envoie à Marie la liste des fichiers modifiés.

La souplesse que procure un gestionnaire de code source est telle que même des développeurs isolés, qui travaillent seuls sur le code source, l'utilisent.

Vocabulaire

Les gestionnaires de code introduisent un vocabulaire qui est entré dans le langage technique des développeurs, et est utilisé pour décrire les actions effectuées sur le code. Voici les principaux termes qu'il convient de connaître :

- **to commit** : poster une modification sur le serveur. Anglisme : *committer*.
 - **changeset** : numéro qui identifie de manière unique l'ensemble des modifications effectuées lors d'un commit.
 - **revision** : numéro qui identifie de manière unique l'état de la base de code entre deux commits.
 - **diff** : différence entre deux versions d'un code source.
 - **merge** : unification de deux versions d'un code source pour former une nouvelle version. Anglisme : *merger*.
 - **repository** : dépôt de données contenant les fichiers versionnés.
- C'est le rôle d'un gestionnaire de code source, ou **Source Configuration Manager** (SCM), qui fournit une application qui s'intercale entre les développeurs et le code.

Fonctionnalités

Les gestionnaires de code ont tous le même schéma de fonctionnement, à savoir :

1. récupération d'une version d'un fichier source ;
2. modification locale, avec possibilité de comparer son travail avec la version originale ;
3. commit des modifications, avec éventuelle gestion de conflit.

8.1.5 Cycle de développement itératif

Lorsqu'un programme dépasse un total de 10 000 lignes en Python (taille arbitraire mais communément adoptée pour cette définition [9]), il rentre dans la catégorie des **large scale applications** et nécessite un processus de développement itératif. Ce mode de progression, qui fait partie du Processus Unifié¹, regroupe les tâches à effectuer en sous-ensembles de manière à réduire les risques liés aux modifications. C'est ce mode qui est adopté pour la plupart des programmes Open Source conçus par des communautés de développeurs : le travail est découpé en tâches regroupées dans des *releases*, définies dans la *roadmap* ou feuille de route.

La progression est organisée autour de trois échelles de temps, schématisées dans la figure 8.1 :

- **Les modifications continues**, qui font passer le logiciel d'un état donné à un état supérieur. Ce changement d'état est défini par un *ticket*, qui peut prendre la forme d'un texte court qui définit une tâche à réaliser. Ces modifications peuvent durer d'une minute à quelques jours, en fonction de leur complexité.
- **Les itérations**, qui regroupent un certain nombre de tickets à réaliser pour faire passer l'application d'un état fonctionnel donne à un état fonctionnel supérieur. Ces itérations durent en général de une à cinq semaines, et rythment le travail des groupes de développement.
- **Les incrémentations**, qui sont de l'ordre de 6 mois, voire un an, et qui regroupent un certain nombre d'itérations, ponctuées de *bugs days* et *code reviews*. Elles correspondent à un état de l'application qui peut être déployée en production.

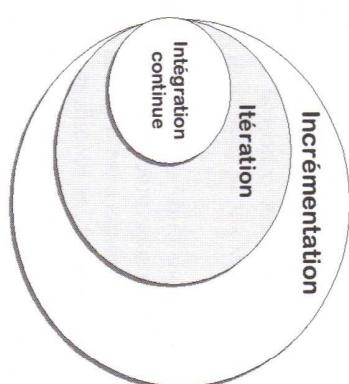


FIG. 8.1 – Le cycle de développement itératif

Ce découpage réduit les risques liés à la production de systèmes complexes : avec une feuille de route définie à l'avance et avec des rythmes de développement adoptés par tous, les débordements (ajout de fonctionnalités non prioritaires, refactoring risqués) sont limités.

Modifications continues

À chaque fois qu'un développeur modifie le code source, il doit être *mandaté* par un ticket.

Exemple de ticket :

Ticket 342

Modifier le fonctionnement de l'écran qui affiche la liste des articles : il faut pouvoir cliquer sur les en-têtes de colonnes pour trier les entrées.

- Pour la version: incrémentation 3 itération 2
- Tickets liés: Aucun

À terme, ce texte permet au mandataire de valider que la modification a bien été effectuée sur le système, et au développeur de gérer sa liste de tâches, qui devient une liste de numéros de tickets.

¹Plus d'informations sur le site de l'OMG : <http://www.omg.org>.

Lorsque le développeur modifie le système, il commit sa modification dans le système de gestion de sources, en notant dans son message le numéro de ticket. Le serveur peut dès lors automatiquement associer cette action au ticket concerné, de manière à ce que l'ensemble des actions effectuées sur le code pour la réalisation d'un ticket soient listées.

Cette procédure, outre la visibilité qu'elle offre, permet de fournir aux développeurs des informations précieuses : lorsqu'ils ont à gérer un ticket qui a des points communs avec d'autres tickets réalisés, ils peuvent se documenter en relisant les modifications en référence.

L'atomicité fonctionnelle des modifications est également importante : lorsqu'un développeur change le code de l'application, il rend cette évolution publique. Il doit donc laisser l'application en état de fonctionnement et ne doit jamais poster un travail partiellement achevé. Le système doit idéalement vérifier à chaque modification que l'application est fonctionnelle, en la reconstruisant et en opérant une campagne de tests. Ces tests sont écrits par les développeurs et sont décrits dans le chapitre 6.

Itérations

Une itération est un cycle durant lequel un certain nombre de tâches vont être réalisées, conformément à un dossier de spécification préparé en amont, afin de passer l'application à un état supérieur cohérent. La difficulté dans la sélection des tickets qui forment une itération réside dans le dosage. Une trop grande quantité de modifications casse l'intérêt de segmenter les avancées en itérations, une trop petite quantité rend l'évolution de l'application d'une itération à l'autre moins perceptible. L'équilibre se trouve en général en fixant la durée des itérations : à 75 % du temps restant, un point basé sur des estimations corrigées au cours du temps peut permettre de repousser pour la prochaine itération les tâches non réalisables ou à fortiori d'en ajouter. La fin de l'itération s'achève idéalement par un *bug day* (voir figure 8.2).

Incrémentation

Les incrémentations, ou *milestones*, sont décrites par une roadmap globale au logiciel, sorte de liste haut niveau des points à réaliser, discutées ou graduellement introduits dans les itérations. Elles correspondent à un moment fort de la vie du logiciel, qui s'apprête à souffrir

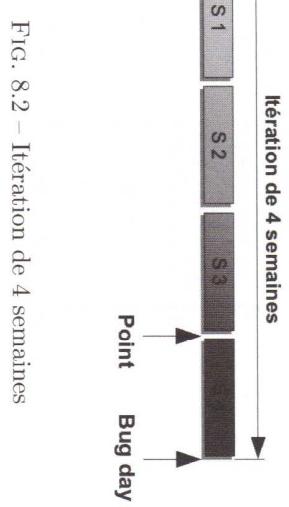


FIG. 8.2 – Itération de 4 semaines

une bougie de plus. D'un point de vue politique et marketing, c'est un événement qui est souvent accompagné de communiqués de presse et autre communications.

8.2 Mise en place technique

Pour répondre aux besoins énoncés, un serveur de développement doit fournir les éléments suivants :

- un gestionnaire de source ;
- un gestionnaire de tickets, qui permet de créer, conserver et effectuer le suivi des tickets ;
- des listes de diffusion, qui permettent aux développeurs d'échanger, d'observer l'évolution de l'application, et d'effectuer les revues de code continues ;
- un système d'intégration continue, qui teste et relance l'application à chaque modification, et qui facilite le travail de liaison.

Ces éléments s'intercalent entre les membres du projet et le code source, comme l'indique la figure 8.3.

8.2.1 Gestionnaire de source

À l'heure actuelle, il existe une myriade d'outils de SCM² sur le marché, que l'on peut regrouper en deux familles :

- les gestionnaires centralisés ;
- les gestionnaires distribués.

²Software Configuration Management en anglais.

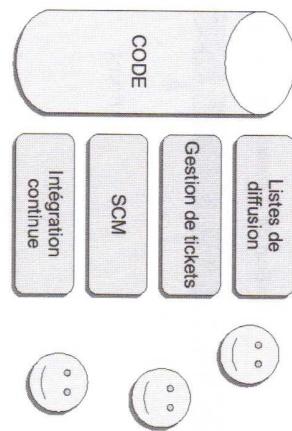


FIG. 8.3 – Serveur de développement

Gestionnaires centralisés

Les gestionnaires centralisés sont historiquement les plus répandus et héritent tous des fonctionnalités introduites par CVS (*Concurrent Versioning System*)³, l'outil le plus utilisé au monde il y a encore quelques années. CVS fournit un système central où le code source est conservé, et chaque modification est historisée, avec un auteur, une date et un message de log. Chaque auteur modifie localement sur sa machine de développement le code et le transmet par le biais d'une petite application cliente au serveur central. L'outil permet de :

- récupérer des versions antérieures;
- travailler de manière isolée dans des branches, qui sont des versions parallèles;
- gérer les conflits et les merges côté client ;
- gérer les tags, les copies de branches.

Depuis trois ans, Subversion (SVN) a supplplanté CVS. Il offre des fonctionnalités supplémentaires comme le renommage des fichiers sans perte d'historique ou des numéros de révisions globaux. Des logiciels majeurs utilisent Subversion :

- GCC ;
- KDE ;
- Python ;
- Connectiva Linux (repository de 30 giga-octet).

Gestionnaires distribués

Une alternative s'est développée pour résoudre cette difficulté et toutes celles inhérentes au télétravail, qui est de plus en plus fréquent dans les entreprises : la gestion distribuée. Dans ce mode, chaque poste de développement gère sa propre base de code source, dans laquelle il commit son travail, et se synchronise de temps à autre avec d'autres bases de code, que ce soit celle d'un autre développeur, ou celle du serveur en charge d'effectuer des releases.

Cette approche décentralisée résout certains problèmes mais change la façon de travailler. Le défaut principal de ce mode de fonctionnement est l'isolement : un développeur qui se détache trop longtemps du *mainstream* des modifications peut avoir beaucoup de mal à merger son travail.

Les outils distribués sont cependant de plus en plus adoptés par les grands projets, comme :

- Drupal ;
- ALSA ;
- Xen ;
- Linux Kernel (en secondaire).

Mise en place de Subversion

Outre l'aspect distribué, le choix d'un gestionnaire de version est aussi dicté par les outils annexes avec lequel il est compatible (bug

³<http://www.nongnu.org/cvs>

Le défaut principal des gestionnaires centralisés est l'obligation pour les développeurs d'être en liaison avec le serveur de source, pour récupérer et poster des modifications. En d'autres termes, un développeur isolé, qui travaille hors ligne un certain temps, ne pourra pas commettre ses modifications avec la granularité souhaitée : il posera en une seule grosse modification l'ensemble de son travail. Cette façon de travailler casse le principe des tickets. Des outils ont été développés pour pallier à ce problème, comme **SVK**⁴, qui propose un système de gestion de source local, avec une fonction de synchronisation qui permet de poster les modifications en *mitraille* lorsque le développeur a accès au code central. Mais cet outil tient plus de la rustine, et le concept même de serveur de code est remis en cause.

⁴<http://svk.bestpractical.com>

tracker, etc.), afin de former une solution de gestion de projet homogène. Parce qu'il est à l'heure actuelle un des plus répandus, un des plus riches fonctionnellement, et *Python friendly*, le choix pour ce livre est **Subversion**.

Choix d'une machine

Un système Subversion est en général installé sur une machine serveur de type Linux, même si dans l'absolu un Windows ou un Mac peut faire l'affaire. Cette préférence est historique car les applications de gestion de source n'existaient il y a quelques années que sur les plates-formes Unixes. Installer aujourd'hui un serveur de développement reste toutefois plus complexe sous Windows, qui nécessite l'utilisation de couches de compatibilités comme *Cygwin*, et Linux reste dans ce domaine le système roi. Le reste de cette section propose une installation sur ce système.

En termes de puissance, toutes les machines actuelles permettent de faire fonctionner correctement un serveur Subversion. Seule la capacité de stockage et la fiabilité importent. Pour les gros projets susceptibles de gérer plusieurs centaines de mégaoctets de données, et des dizaines de développeurs, l'idéal est de :

- dédier une partition voire un disque complet aux données ;
- mettre en place un système de redondance pour éviter les ruptures de service : *High Availability* (voir Linux HA⁵), disques RAID, etc. ;
- mettre en place un système de sauvegarde journalier.

La suite du chapitre a été construite et testée sur un serveur Debian Sarge, avec un compte disposant des droits *sudo*. L'installation de sudo se fait par un simple *apt-get install sudo*, et l'utilisateur est ajouté au fichier */etc/sudoers* de façon similaire à root.

Installation du serveur Subversion

L'application serveur de Subversion est constituée d'un ensemble de scripts dédiés à la gestion des données et d'un pont de communication qui lui permet d'échanger avec une machine client. Ce pont

est multiprotocolaire, et permet d'utiliser SSH (*Secure Shell*) pour sécuriser les transactions. Installer un serveur Subversion consiste à :

- Récupérer et installer les paquets Subversion.
- Créer un utilisateur dédié en charge de lancer les processus.
- Configurer et mettre en place le mode d'accès.
- Valider le fonctionnement depuis un compte client.

Récupérer et installer les paquets Subversion Il est recommandé d'installer Subversion depuis les sources, afin d'ajouter le support du connecteur Python, qui servira à la mise en place du gestionnaire de tickets à la section 8.2.2. Les derniers sources sont récupérés dans la section *downloads* du site de Subversion⁶ et installés avec le support de Python, via Swig :

```
$ cd /usr/local/src
$ wget http://subversion.tigris.org/downloads/
$ tar -xzvf subversion-1.4.2.tar.gz
$ cd subversion-1.4.2
$ ./configure
$ make
$ sudo make install
$ make swig-py
$ sudo make install-swig-py
```

Créer un utilisateur dédié en charge de lancer les processus Toutes les manipulations de fichiers effectuées par les programmes de subversion, doivent être faites par un utilisateur dédié, afin de sécuriser le serveur et d'en faciliter son administration. L'ajout d'un utilisateur sous Linux se fait par la commande suivante :

```
$ sudo adduser --no-create-home svn
```

Si cette commande échoue, il est probable que cet utilisateur existe déjà sur votre système, créé par une installation par paquets, ou par un paramétrage d'usine de la distribution.

⁵<http://linux-ha.org/>

⁶<http://subversion.tigris.org/project-packages.html>

Créer un repository pour le projet L’arborescence qui va contenir le code source géré par Subversion, doit être initialisée par la commande **svnadmin**. La meilleure localisation pour cette arborescence est un dossier dédié dans */var/svn*, et est créée avec l’utilisateur **svn**. Le projet créé pour l’exemple se nomme *atomisator*.

```
$ sudo su -
# mkdir /var/svn
# chown svn:svn /var/svn
# su - svn
$ cd /var/svn/
$ svnadmin create --fs-type=fsfs atomisator
$ ls
atomisator
```

Deux types de stockage sont possibles avec Subversion : un de type *base de données* basé sur la bibliothèque BerkeleyDB, et un de type *système de fichiers*, basé sur une arborescence de fichiers classique. La version fichier, appelée FSFS, est la plus sûre en cas de crash du serveur : les fichiers sont tous physiquement présents sur le système et ont plus de chances d’être récupérés qu’une base de données qui risque d’être corrompue.

Configurer et mettre en place le mode d'accès Le serveur Subversion fournit un programme appelé **svnservve**, qui permet aux clients d'accéder aux données par le biais du protocole *svn*, avec la possibilité d'utiliser une authentification gérée en interne ou une authentification SSH gérée par le système. Les URL pour accéder au repository auront dans le cas de l'authentification interne la forme :

```
svn://atomisator.org/var/svn/atomisator
```

Pour un accès direct, chaque utilisateur doit être déclaré dans le fichier *passwd* qui se trouve dans le répertoire conf du repository. Pour autoriser le user *tarek* à utiliser le repository, il suffit de l'ajouter :

```
[users]
# harry = harrysecret
# sally = sallysecret
tarek = coucou
```

Il faut ensuite décommenter dans le fichier *svnservve.conf* la ligne suivante pour lier le fichier *passwd* au système d'authentification :

password-db = passwd

Pour lancer le serveur Subversion, *svnservve* est testé une première fois avec l'option *foreground*, pour valider qu'aucune erreur n'apparaît, puis relancé en mode *daemon*. Un grep sur les processus actifs permet de valider que le serveur tourne bien :

```
$ svnservve --foreground -d
^C
$ svnservve -d
$ ps aux | grep svnservve
svn      2654  0.0  0,1    33340   512   ??  Ss
6:49  0:00.00 svnservve
```

Valider le fonctionnement depuis un compte client Accéder au repository depuis un poste client se fait par le biais de la commande **svn**, installée avec les mêmes bibliothèques que la partie serveur. Cette commande s'invoque suivie d'un paramètre, dont les principaux sont :

- **add** : ajoute un fichier au repository local;
- **checkout** (*co*) : crée un repository local, image du repository;
- **commit** (*ci*) : synchronise les modifications locales vers le repository;
- **diff** (*di*) : affiche les modifications locales sous forme de différence;
- **status** (*st*) : affiche la liste des fichiers modifiés localement;
- **import** : équivalent à add mais directement importé dans le repository;
- **update** (*up*) : met à jour le repository local.

Voici un exemple de session **svn** :

```
$ mkdir atomisator
$ cd atomisator/
$ svn co svn://localhost/var/svn/atomisator .
Révision 0 extraite.
$ touch test.txt
$ svn add test.txt
A          test.txt
$ svn ci -m "ajout_d'un_fichier"
Domaine d'authentification : \
```

<svn://localhost:3690>

Mot de passe pour 'tarek' : coucou

Ajout **test.txt**

Transmission des données .

Révision 1 propagée.

\$ svn ls svn://localhost/var/svn/atomisator

test.txt

\$ svn st

\$ echo 'modif' > **test.txt**

\$ svn di

Index: **test.txt**

— **test.txt** (révision 1)

(copie de travail)

+modif

\$ svn ci -m "modif"

Envoyer **test.txt**

Transmission des données .

Révision 2 propagée.

Sécuriser les accès Il est possible de sécuriser les accès en utilisant le protocole SSH. Pour mettre en place ce fonctionnement, il suffit de lancer un serveur SSH⁷ et de laisser les utilisateurs se connecter en utilisant des préfixes *svn+ssh*:

svn+ssh://atomisator.org/var/svn/atomisator

Si l'utilisateur dispose d'un compte sur le serveur, il pourra accéder au repository⁸. Cette technique d'authentification a cependant un défaut : elle nécessite de créer un compte utilisateur pour chaque profil qui a besoin d'un accès. Pour éviter cette contrainte lorsque le projet a besoin d'offrir des accès externes, Subversion peut être configuré pour fonctionner comme une extension Webdav d'Apache⁹. Dans ce cas, la gestion des accès est gérée par Apache, et peut être sécurisée sans avoir à créer des comptes Unix pour chaque utilisateur.

⁷Daemon sshd.

⁸À condition toutefois qu'il ait les droits système sur le dossier du repository.
⁹Cette procédure est expliquée en détail dans le livre en ligne de Subversion, à cette adresse : <http://svnbook.red-bean.com>.

Les URL d'accès deviennent par exemple¹⁰ :

<http://atomisator.org/svn>

<https://atomisator.org/svn> (en mode TLS)

Contrôler la qualité des commits entrants Subversion propose à l'administrateur d'exécuter des scripts à chaque fois qu'une modification est réalisée par un développeur. Ces programmes peuvent être appelés avant (*pre-commit hook*) ou après (*post-commit hook*) que la modification ait été appliquée. Le pre-commit-hook permet de protéger la base de code de modifications non souhaitées, comme :

- l'ajout de fin de lignes Windows (Carriage Return+Line Feed)
- au code;
- l'ajout de tabulations;
- l'oubli d'une insertion d'un Line Feed en fin de fichier.

Ces problèmes sont provoqués par un éditeur de code mal réglé et peuvent, dans le cas des tabulations ou des CR+LF, casser le code. Un script de pre-commit qui bloque les commits qui contiennent ces défauts est un moyen simple et automatique de protéger la base de code. Il est conseillé de ne pas nettoyer de manière transparente les fichiers entrants, car les développeurs responsables resteraient dans l'ignorance de l'existence du problème. La section *scripts* du site <http://programming-python.org> présente un exemple de script qui peut être utilisé en pre-commit. Ce script est appellé dans le script shell *pre-commit*¹¹ qui se trouve dans le repository dans le dossier *hooks*. L'appel prend cette forme :

```
/chemin/vers/script svn-check-source.py \
"$_REPOS" "$TXN" || exit 1
```

Les contrôles bloquants sont à utiliser à bon escient : trop restrictifs ils peuvent entraver le travail quotidien.

Similairement, une série de scripts peut être appelée à la suite

d'un commit, avec le post-commit hook. Les deux utilisations les

¹⁰Le chemin est défini dans la configuration et permet de masquer les chemins absolu du système.

¹¹Il faut renommer le fichier *pre-committmpl*.

plus fréquentes sont le déclenchement d'une compilation (voir 8.2.4) et d'une série de tests et la liaison éventuelle avec un gestionnaire de tickets (voir 8.2.2).

8.2.2 Gestionnaire de tickets

Le monde des gestionnaires de tickets, ou *issue trackers* est riche, et on peut dénombrer autant d'outils qu'il existe de langages de programmation. Outre le socle de base qui consiste à gérer la naissance, la vie et la mort d'un ticket et tout ce que ce cycle incombe, l'outil fournit bien souvent des fonctionnalités supplémentaires pour devenir l'élément central de la gestion de projet. Les plus notables sont :

- l'intégration du gestionnaire de sources ;
- l'intégration du système de tests ;
- la possibilité de modifier le cycle de vie des tickets, pour élaborer des scénarios de traitement ;
- la gestion de temps ;
- l'intégration d'un système de documentation, comme un wiki ou des rapports statistiques.

Dans le monde libre¹², les plus fréquemment utilisés dans les projets Python sont :

- Trac, qui a émergé ces deux dernières années (voir figure 8.4) ;
- Roundup, remarquable par sa modularité ;
- Bugzilla, de la fondation Apache.

FIG. 8.4 – Visualisation des tickets dans Trac

Installation de Trac

Trac est un programme développé en Python, et s'installe donc comme une simple extension. Une section très détaillée sur le site du projet donne les informations pour son installation : <http://trac.edgewall.org/wiki/TracInstall>.

Le tracker le plus simple d'usage pour un projet Python géré par subversion est **Trac**. Il n'a pas de système de gestion de temps mais propose un panel de fonctionnalités largement suffisantes pour la plupart des projets. Trac offre une interface web qui permet :

- de visualiser en temps réel les commits effectués (*timeline*) ;
- d'associer les tickets à des itérations, et d'afficher une barre de progression d'une incrémentation ;
- de naviguer dans le code, visualiser les diffs, les versions, les changesets ;
- de lier les bugs à des révisions, pour fournir un système de suivi ;
- de créer un site web basique, basé sur un système wiki.

¹²A noter une utilisation non négligeable de JIRA qui n'est pas libre mais gratuit pour les projets Open Source publiques.

Congratulations!

La dernière étape de configuration consiste à créer un fichier de

mots de passe avec l'utilitaire *htpasswd*, qui contiendra les profils générés par Trac :

```
$ cd /var/trac/
$ htpasswd -c trac.HTPAsswd admin
New password: admin
Re-type new password: admin
Adding password for user admin
$ htpasswd -c trac.HTPAsswd tarek
New password: tarek
Re-type new password: tarek
Adding password for user tarek
```

Le système peut ensuite être lancé par le biais du démon *tracd* :

```
$ tracd --port 8000 /var/trac/projet \
--basic-auth *,/var/trac/trac.HTPAsswd ,Trac
```

Le tracker du projet devient dès lors accessible par le biais d'un navigateur à l'adresse *http://localhost:8000/projet*. Trac dispose également d'un système de gestion de rôles intégré, accessible depuis la commande *trac-admin*¹³, qui permet de gérer les droits inhérents aux profils.

```
$ trac-admin /var/trac/projet permission list
User      Action
-----
```

```
anonymous    BROWSER_VIEW
anonymous    CHANGESET_VIEW
anonymous    FILE_VIEW
...
```

post-commit hook

Lorsque des tickets sont ajoutés à Trac, ils sont dotés, comme les révisions de Subversion, d'un numéro unique. Grâce à cet identifiant, il est possible de lier les commits au ticket, afin de :

- fermer automatiquement des tickets ;
- ajouter des commentaires à certains tickets.

Le script *trac-post-commit-hook*, fourni dans le répertoire *contrib* de Trac, peut être utilisé pour effectuer cette tâche. Il interprète le message de commit pour effectuer des opérations dans la base des tickets. Il est appelé dans le script *post-commit-hook* qui se trouve dans le repository dans le dossier *hooks*, de la manière suivante :

```
REPO="$1"
REV="$2"
LOG='svnlook log -r $REV $REPO'
AUTHOR='svnlook author -r $REV $REPO'
TRAC_ENV=/var/trac/projet/
python /usr/local/src/trac/contrib/trac-post-commit-hook \
-p "$TRAC_ENV" \
-r "$REV" \
-u "$AUTHOR" \
-m "$LOG"
```

À chaque modification, le script recherche dans les messages de commit les patterns suivants :

- (*closes ou fixes*) #1 (, ou & ou and) #2 : ferme les tickets 1 et 2;
- (*references ou refs ou adresses ou re*) #1 : ajoute un commentaire dans le ticket 1.

Exemples :

- svn ci -m "corrected label, fixes #1"
- svn ci -m "changed param 2, closes #1, #2"
- svn ci -m "corrected label, fixes #1, refs #3"

Trac fournit également un script de pre-commit-hook qui bloque tous les commits qui n'ont pas de référence de ticket, ouvert dans leurs commentaires. Ce contrôle n'est pas conseillé, car de nombreux commits ne concernent pas de tickets précis. C'est le cas des corrections de style, de typographie ou toute modification instinctive du code qui n'a pas été commanditée par un ticket. L'alternative peut être de réservé des tickets pour ces modifications d'ordre général pour lister les commits concernés, mais ils peuvent vite devenir un raccourci à chaque fois que le script bloque un commit.

¹³Voir <http://trac.edgewall.org/wiki/TracPermissions>

8.2.3 Listes de diffusion

- Les listes de diffusion d'un projet permettent :
 - de fournir un flux d'informations automatique sur l'évolution de la base de code;
 - de fournir un média d'échange entre les membres de la communauté.

Les quatre listes de diffusion à mettre en place sont :

- la *Checking list* ;
- la *Dev list* ;
- la *Quality list* ;
- la *Community list*.

Checking list

Cette liste reçoit, par le biais d'un hook Subversion, toutes les modifications du code, sous la forme de diff. Elle permet de pratiquer une revue de code continue (voir section 8.1.3) Les développeurs peuvent directement réagir dessus et initier un échange sur une modification qui a été effectuée.

Dev list

Cette liste permet aux développeurs de faire des échanges sur le design de l'application et de passer des annonces sur les événements liés au développement.

Quality list

Les rapports de qualité, comme ceux générés par PyLint (voir section 5.4.1), peuvent être déclenchés périodiquement sur la base de code par un script, puis envoyés sur cette liste dédiée. Ce flux d'informations spécialisé permet de sensibiliser les développeurs à la qualité du travail produit : un score public les incite à soigner le code et à être proactifs, en utilisant régulièrement PyLint ou un outil équivalent, lorsqu'ils développent.

Installation de Buildbot

Installer Buildbot consiste à mettre en place un serveur et un certain nombre d'esclaves. Chaque esclave est chargé de lancer des tests. Ils sont programmés via un script de lancement. La procédure

et les utilisateurs, une liste dédiée à l'utilisation du projet est créée, pour regrouper des messages moins techniques et venir en aide aux utilisateurs.

L'ensemble des listes présentées peuvent être mises en place avec *Mailman*¹⁴ ou *Sympa*¹⁵. Le choix le plus judicieux étant Mailman, car il est codé en Python et peut être si nécessaire manipulé nativement par des scripts Python.

Installation de Mailman

L'installation de Mailman consiste :

- à mettre en place l'application sur un serveur ;
- à la paramétriser pour qu'elle fonctionne avec le serveur de mails ;
- à créer les mailings lists ;
- à mettre en place un accès web, via Apache.

L'ensemble de la procédure est définie à cette adresse : <http://www.gnu.org/software/mailman/mailman-install/index.html>.

8.2.4 Intégration continue

Un projet développé par les tests (voir chapitres 6 et 7) doit contrôler systématiquement que chaque modification n'entraîne pas de régressions. Cette tâche incombe au développeur qui a modifié le code : il doit relancer les tests avant de committer. Mais il ne peut pas procéder à un test global et exhaustif de l'application à chaque fois qu'il change une portion de code, soit parce que la batterie de tests complète est trop longue à s'exécuter, soit parce qu'il est nécessaire de lancer les tests sur plusieurs plateformes. Le développeur effectue donc des tests ciblés et le projet doit automatiser le contrôle global de l'application. On parle d'*intégration continue* et ce rôle est tenu par *Buildbot*.

Community list

Pour les gros projets, pour lesquels la communauté est suffisamment importante pour qu'il y ait une distinction entre les développeurs

¹⁴<http://www.gnu.org/software/mailman>

¹⁵<http://www.sympa.org>

8.3. CE QU'IL FAUT RETENIR

pour l'installation est indiquée sur le site officiel :

<http://buildbot.sourceforge.net>.

Buildbot est paramétré par le biais du fichier *master.cfg*, qui permet de définir les actions à mener par les esclaves lorsque le système est invoqué. Chaque commit sur la base de code invoque Buildbot, qui lui-même réveille les esclaves et les pilote.

THE AMERICAN JOURNAL OF THEOLOGY AND PHILOSOPHY

- faire un checkout de la base de code;
 - lancer un script de tests sur cette base ;
 - récupérer la sortie, qui sera *valide* ou *invalidé* ;
 - envoyer par mail les résultats, lorsque la sortie est invalide.

Le checkout du code est dans l'idéal un *bundle* dédié aux tests qui contient un script capable de déclencher les tests, et de renvoyer

exemple de script de tests sont présentés dans la section *scripts* du site <http://programming-python.org>. Le script de tests donné en exemple est utilisé pour tester une application Zope, et se base sur le script de test de *zope.testing*. Les tests de chaque paquet présent dans l'application sont lancés et leurs sorties renvoyées dans la sortie standard. En cas d'erreur, un code 1 est renvoyé pour que buildbot déclenche une erreur à son tour et envoie un mail.

Buidbot est lié à l'application grâce à un appel dans le script de post-commit de Subversion :

svn-buildbot.py est un script par défaut fourni par le projet Buildbot.

Le principe d'intégration continue permet en outre de tester l'application sur une grande variété de plateformes : chaque esclave peut être placé sur une autre machine que le serveur buildbot. Python lui-même utilise ce système pour son développement, afin de valider que chaque modification laisse le langage fonctionnel sur chaque plateforme supportée. Pour Python 2.5, il peut être visualisé à l'adresse <http://www.python.org/dev/buildbot/2.5>. La figure 8.5 est un exemple de sortie des esclaves de ce Buildbot.

8.4.3 Packaging et distribution

Lorsque l'application doit étre déployée, il convient d'utiliser les fonctionnalités de tagging de Subversion, pour étiqueter précisément ce qui est délivré. Cette mise en bouteille s'accompagne en général de numéros de versions pour chaque paquet. La technique la plus aboutie et la plus souple actuellement pour diffuser son code est l'utilisation de *setuptools* qui facilite l'empaquetage des projets et leur publication sur des dépôts publics lorsqu'ils sont diffusés largement.

8.3 Ce qu'il faut retenir

La gestion de projet agile devient fondamentale pour les applications qui grossissent et surtout qui s'ouvrent à des développeurs extérieurs, qui ne sont pas forcément en contact directs les uns avec les autres. Elle a été très développée dans le monde de l'Open Source et les grands projets Python ont été pionniers dans ce domaine. C'est pourquoi la plupart des outils qui ont été présentées dans ce chapitre sont eux-mêmes codés en Python.

La gestion de projet agile devient fondamentale pour les applications qui grossissent et surtout qui s'ouvrent à des développeurs extérieurs, qui ne sont pas forcément en contact directs les uns avec les autres. Elle a été très développée dans le monde de l'Open Source et les grands projets Python ont été pionniers dans ce domaine. C'est pourquoi la plupart des outils qui ont été présentés dans ce chapitre sont eux-même codés en Python.

Pour les projets plus modestes, ou développés dans un environnement fermé, les méthodes présentées apportent aussi de la rigueur et une augmentation sensible de la qualité aussi bien en termes de communication au niveau de l'équipe qu'en termes de code.

FIG. 8.3 - De Buidert en zijn zonen

Index

- object, 53
- Observer, 71
- Opérateurs, 37
- Pair writing, 158
- Paquet, 48
- Paramétrabilité, 23
- Polymorphisme, 59
- Programmation générative, 24
- Programmation générique, 24
- Programmation orientée composants, 62
- Property, 56
- Proxy, 66
- PyCommunity, 151
- Pylint, 96
- Recette, 148
- reStructuredText, 118
- rst2html, 119
- Scalability, 22
- Simplicité, 23
- Singleton, 69
- Système d'exploitation, 10
- Team writing, 157
- Test fonctionnel, 103
- Test unitaire, 102
- Test-driven development, 100
- Transformée Schwartzienne, 84
- Tuple, 39
- Tutoriel, 145
- Types de base, 35
- Unitest, 104
- Visitor, 70
- None, 41
- Ellipsis, 41
- Encodage, 47
- Adaptabilité, 31
- Adapter, 68
- Bad smelling code, 29
- Borg, 69
- Cheesecake, 97
- Classe, 45
- Code pattern, 74
- contextlib, 90
- Cookbook, 151
- Développement dirigé par la documentation, 151
- Développement dirigé par les tests, 100
- DDD, 151
- Dictionnaires, 39
- docstring, 131
- doctest, 108, 130
- Documentation d'un paquet, 140
- Documentation d'un projet, 143
- Documentation unitaire, 136
- Dogfooding, 10
- Écriture technique, 110
- Éditeur de code, 13
- Ellipsis, 41
- Encodage, 47
- Environnement de développement intégré, 112
- Extensibilité, 22
- Fonctions, 44
- for...in, 42
- Generator, 77
- Generator expression, 79
- Glossaire, 150
- Héritage, 57
- Indentation, 32
- Interface, 63
- itertools, 79
- List comprehension, 75
- Liste, 39
- Liste compréhensive, 75
- Literate programming, 28
- Documentation d'un paquet, 140
- Documentation unitaire, 136
- Dogfooding, 10
- Écriture technique, 110
- Éditeur de code, 13
- Ellipsis, 41
- Encodage, 47

Bibliographie

- [1] S. W. Ambler. *Agile Modeling : Effective Practices for Extreme Programming and the Unified Process*. John Wiley and Sons, 2002.
- [2] D. T. Andrew Hunt. *The Pragmatic Programmer : From Journeyman to Master*. Addison Wesley, 1999.
- [3] B. D. Bloom. *Developing Talent in Young People*. Ballantine Books, 1985.
- [4] P. Elbow. *Writing with power*. Oxford University Press, 1998.
- [5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.
- [6] C. M. C. George L. Kelling. *Fixing Broken Windows : Restoring Order And Reducing Crime In Our Communities*. Free Press, 1998.
- [7] J. R. Hayes. *Complete Problem Solver*. Lawrence Erlbaum Associates, 1989.
- [8] D. E. Knuth. *Literate Programming*. Center for the Study of Language and Inf, 1992.
- [9] L. Marc-André. Developing large-scale applications. In *Libre Software Meeting*, 2005.
- [10] A. Martelli and D. Ascher. *Python Cookbook*. O'Reilly, 2002.
- [11] A. Rueping. *Agile Documentation : A Pattern Guide to Producing Lightweight Documents for Software Projects*. Wiley, 2003.
- [12] T. Ziadé. *Programmation Python*. Eyrolles, 2006.