INFORMATIQUE
TECHNIQUE

Expert

Python 3

Traitement de données et techniques de programmation

En téléchargement

Code source des exemples

Version en ligne
OFFERTE !
pendant 1 an

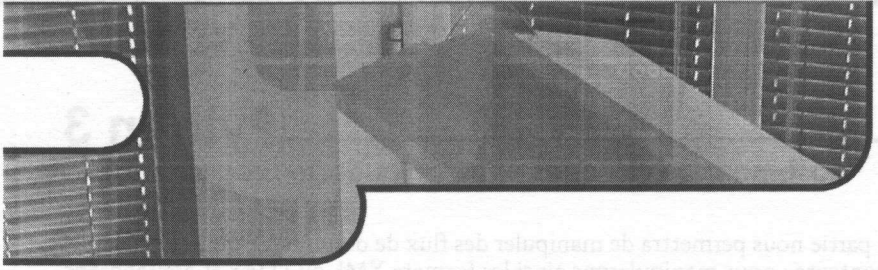
Sébastien CHAZALLET

eni

Expert IT

Python 3

Traitement de données et techniques de programmation



Avant-propos

Quelle serait l'utilité de disposer d'un langage de programmation, fût-il exceptionnel, si l'on ne pouvait pas l'utiliser avec d'autres outils ? Accepteriez-vous d'apprendre un langage qui ne peut pas ouvrir de fichiers ou se connecter à une base de données SQL ? Un tel langage n'aurait qu'une portée théorique.

Aujourd'hui, connaître un ou même plusieurs langages ne suffit pas à un développeur pour lui permettre de faire correctement son métier. Il faut être **polyvalent**, savoir maîtriser des bases de données, des flux XML, manipuler des images, générer des documents, connaître des langages tiers.

Par exemple, pour manipuler du XML, il faut savoir comment ce format fonctionne. Au-delà de ceci, si l'on échange avec un service tiers, il faut savoir comment ce service formate ses données et déterminer la meilleure manière de les exploiter.

Pour faire de la programmation web, il faudra maîtriser les langages HTML et CSS par exemple, tout comme il faudra avoir des notions de système, possiblement de bash pour faire de la programmation système ou connaître les protocoles TCP et UDP pour faire de la programmation réseau.

C'est donc, au-delà de la simple maîtrise d'un langage, une véritable culture technique qu'il faut cultiver pour répondre à des besoins qui sont aujourd'hui très communs.

On ne peut pas apprendre tout cela en un jour. Heureusement pour nous, pythonistes, notre langage Python permet d'abaisser le ticket d'entrée en offrant des modules et des techniques permettant de répondre simplement à de nombreux besoins.

Dans ce livre, nous allons vous proposer de monter en compétence dans plusieurs domaines, en expliquant les connaissances de base qu'il faut avoir et en vous montrant par l'exemple ce que l'on peut faire avec Python. Nous vous pointerons vers les modules les plus adaptés et présenterons des exemples qui sont complémentaires avec les diverses documentations officielles.

La première partie nous permettra de manipuler des flux de données, ce qui est utile quel que soit le contexte ; nous manipulerons ainsi les formats XML ou LDAP et apprendrons présenter des données en générant des documents dans différents formats.
La seconde partie se focalisera sur divers domaines de programmation : système, réseau, web et scientifique ; on s'attachera à présenter les technologies associées et à montrer comment on peut, avec Python, tirer parti de chaque domaine de compétence.
La troisième et dernière partie concerne plus spécifiquement les différentes manières d'optimiser son code en utilisant au mieux les ressources matérielles : programmation parallèle et distribuée, mais surtout, la programmation asynchrone qui est un des domaines dans lequel le langage a le plus évolué récemment.

© Editions ENI - All rights reserved

Table des matières

Les éléments à télécharger sont disponibles à l'adresse suivante :
http://www.editions-eni.fr
Saisissez la référence de l'ouvrage **EI3PYT** dans la zone de recherche et validez. Cliquez sur le titre du livre puis sur le bouton de téléchargement.

Avant-propos

Partie1
Traitement des données

Chapitre 1.1
Motifs de conception

1. Définition 15
   - 1.1 Positionnement par rapport à la notion d'objet 15
   - 1.2 Organisation du chapitre 16
   - 1.3 Positionnement par rapport à d'autres concepts 16
2. Motifs de création 17
   - 2.1 Singleton 17
   - 2.2 Fabrique 17
   - 2.3 Fabrique abstraite 20
   - 2.4 Monteur 21
   - 2.5 Prototype 23
3. Motifs de structuration 26
   - 3.1 Adaptateur 26
   - 3.2 Pont 29
   - 3.3 Composite 32
   - 3.4 Décorateur 34
   - 3.5 Façade 36
   - 3.6 Poids-mouche 38
   - 3.7 Proxy 40
4. Motifs de comportement 42
   - 4.1 Chaîne de responsabilité 42
   - 4.2 Commande 43
   - 4.3 Itérateur 45

4.4 Mémento 47
4.5 Visiteur 48
4.6 Observateur 49
4.7 Stratégie 51
4.8 Fonction de rappel 52
5. ZCA 53
5.1 Rappels 53
5.2 Adaptateur 53
5.3 Utilitaire 55
5.4 Fabrique 56
5.5 Pour aller plus loin 57

Chapitre 1.2
XML

1. XML et les technologies qui gravitent autour 59
1.1 Définition de XML, terminologie associée 59
1.2 Notion de schéma 60
1.3 Avantages et inconvénients de XML 61
1.4 Différentes manières de parcourir un fichier XML 62
1.5 Modules Python dédiés au XML 62
2. Valider un document XML 63
2.1 Document XML 63
2.2 Schéma DTD 64
2.3 Schéma XSD 64
2.4 Schéma RNG (RelaxNG) 65
2.5 Schematron 65
3. DOM 66
3.1 Lecture 66
3.2 Écriture 67
4. SAX 68
4.1 Support de SAX dans lxml 68
4.2 API SAX Allégée 69
5. XPath 70
6. XSLT 72

7. Cas spécifique des fichiers HTML 74
7.1 Problématique 74
7.2 Parser un fichier HTML à la façon DOM 74
7.3 Parser un fichier HTML à la façon SAX 76

Chapitre 1.3
Génération de contenu

1. PDF 79
1.1 Présentation 79
1.1.1 Format PDF 79
1.1.2 Avantages 79
1.1.3 Inconvénients 79
1.1.4 Présentation de la bibliothèque libre 80
1.2 Bas niveau 80
1.2.1 Bibliothèque de données 80
1.2.2 Canvas 82
1.3 Haut niveau 83
1.3.1 Styles 83
1.3.2 Flux de données 85
1.3.3 Création d'un visuel 87
1.3.4 Template de page 87
1.3.5 Page contenant plusieurs zones 88
2. OpenDocument 90
2.1 Présentation 90
2.2 ezodf2 91
2.2.1 Installation 91
2.2.2 OpenDocument Texte 91
2.2.3 OpenDocument Tableur 91
2.2.4 Aller plus loin 92
2.3 Alternatives 92
2.3.1 Ipod 92
2.3.2 Génération à partir de templates 92
3. Travailler avec des images 93
3.1 Représentation informatique d'une image 93
3.2 Présentation de Pillow 94

3.3 Formats d'images matricielles 95
3.4 Récupérer des informations d'une image 97
3.5 Opérations d'ensemble sur une image 98
3.6 Travailler avec les calques ou les pixels 100

Chapitre 1.4
Qualité

1. Programmation dirigée par les tests 103
1.1 Tests unitaires 103
1.1.1 Principes 103
1.1.2 Interprétation 104
1.1.3 Couverture 105
1.1.4 Outils 105
1.1.5 Autres outils 107
1.2 Tests de non-régression 107
1.2.1 Actions de développement 107
1.2.2 Gestion de la découverte d'une anomalie par une MOA 108
1.3 Tests fonctionnels 109
1.4 Tests de performance 110
1.5 Tests syntaxiques 112
1.6 Intégration continue 113
2. Programmation dirigée par la documentation 114
2.1 Documentation interne 114
2.1.1 À destination des développeurs 114
2.1.2 À destination des utilisateurs 115
2.2 Documentation externe 115
2.2.1 Présentation 115
2.2.2 Démarrage rapide 115
2.2.3 Résultat 118
3. Optimisation 118
3.1 Qualimétrie 118
3.2 Outils de débogage 120
3.3 Outils de profilage 120

3.4 Règles d'optimisation 122
3.4.1 Pourquoi optimiser ? 122
3.4.2 Règles générales 123
3.4.3 Profiler un algorithme 124
3.4.4 Optimiser l'utilisation de la mémoire 133

Partie 2
Programmation, système, réseau et scientifique

Chapitre 2.1
Programmation système

1. Présentation 135
1.1 Définition 135
1.2 Objectifs du chapitre 136
2. Appréhender son système d'exploitation 136
2.1 Avertissement 136
2.2 Système d'exploitation 136
2.3 Processus courant 137
2.4 Utilisateurs et groupes 138
2.5 Mots de passe et authentification 140
2.6 Constantes pour le système de fichiers 141
2.7 Gérer les chemins 142
2.8 Utilitaires 142
2.8.1 Comparer deux fichiers 142
2.8.2 Utilitaire de sauvegarde 144
2.8.3 Lire un fichier de configuration 145
2.8.4 Pickle 146
2.9 Compresser et décompresser un fichier 148
2.9.1 Tarfile 148
2.9.2 Gzip 150
2.9.3 Bz2 151
2.9.4 Zipfile 151
2.9.5 Interface de haut niveau 153
3. Gestion d'un fichier 154
3.1 Changer les droits d'un fichier 154
3.2 Changer de propriétaire ou de groupe 156

3.3 Récupérer des informations sur un fichier 157
3.4 Supprimer un fichier 157
4. Alternatives simples à des commandes bash usuelles 158
4.1 Répertoires 158
4.2 Fichiers 160
4.3 Module de haut niveau 161
4.4 Recherche d'un fichier 163
5. Exécuter des commandes externes 164
5.1 Exécuter et afficher le résultat 164
5.2 Exécuter et récupérer le résultat 165
5.3 Pour Python 3.5 et supérieur 166
6. IHM système 166
6.1 Présentation 166
6.2 Travailler avec des arguments 167
6.3 Fermeture du programme 171
6.4 Entrée standard 172
6.5 Sortie standard 173
6.6 Sortie d'erreur 174
6.7 Taille du terminal 175
7. Curses 176
7.1 Présentation 176
7.2 Gérer le démarrage et l'arrêt 176
7.3 Affichage de texte 177
7.4 Gestion des attributs 177
7.5 Gestion des couleurs 178
7.6 Gestion des événements 178
8. Accès aux périphériques 178
8.1 Introduction 179
8.2 Voir les partitions 179
8.3 Voir les clés USB 180
8.4 Liste des informations disponibles 181
8.5 Détecter le branchement d'une clé USB 181
8.6 Communiquer via un port série 182
8.7 Communiquer via un port USB 183

Chapitre 2.2
Programmation réseau

1. Écrire un serveur et un client 185
   1.1 Utilisation d'une socket TCP 185
   1.2 Utilisation d'une socket UDP 189
   1.3 Création d'un serveur TCP 191
   1.4 Création d'un serveur UDP 193
   1.5 Un peu plus loin sur le sujet 194
2. Utiliser un protocole standard 195
   2.1 Client HTTP 195
   2.2 Serveur HTTP 196
   2.3 Proxy 199
   2.4 Cookies 200
   2.5 FTP et SFTP 200
   2.6 SSH 202
   2.7 POP et POPS 204
   2.8 IMAP et IMAPS 205
   2.9 SMTP et SMPTS 206
   2.10 NNTP 211
   2.11 IRC 212
3. Wake-on-LAN 215
   3.1 Prérequis 215
   3.2 Mise en œuvre 215

Chapitre 2.3
Programmation web

1. Services web 217
   1.1 REST 217
   1.2 SOAP 218
   1.3 Pyro 220
2. Client web 221
3. Web scrapping 225

Chapitre 2.4
Programmation scientifique

1. Calcul scientifique 227
   1.1 Présentation 227
   1.2 Python, une alternative libre et crédible 227
   1.3 Vue d'ensemble de quelques bibliothèques 228
2. Tableaux multidimensionnels 229
   2.1 Création 229
   2.2 Déterminer la composition d'un tableau 230
   2.3 Générateurs de tableaux 231
   2.4 Opérations basiques 232
   2.5 Opérateur crochet 234
3. Matrices 236
4. Génération de graphiques 237
   4.1 Syntaxe MATLAB 238
   4.2 Syntaxe objet 239
   4.3 Mise en forme 239
   4.4 Graphiques 3D 244
5. Introduction à Pandas 247
   5.1 Présentation 247
   5.2 Séries 247
   5.3 Dataframes 251

Partie 3
Programmation concurrente

Chapitre 3.1
Initiation à la programmation concurrente

1. Notion de performance 259
   1.1 Programmation bloquante 259
   1.2 Utilisation de ressources CPU 260
   1.3 Organisation de l'application 261

2. Terminologie 262
2.1 Processus 262
2.2 Fil d'exécution 263
2.3 Programmation non bloquante 264
2.4 Notion de GIL 264
3. Présentation des paradigmes 265
3.1 Programmation asynchrone 265
3.2 Programmation parallèle 265
3.3 Programmation distribuée 266
3.4 Programmation concurrente 267

Chapitre 3.2
Programmation asynchrone : initiation

1. Utilité de la programmation asynchrone 269
2. Introduction à l'asynchrone 270
2.1 Notion de coroutine 270
2.2 Exécution d'une coroutine 271
2.3 Précisions sur le mot-clé await 272
2.4 Exécuter plusieurs coroutines en séries 273
2.5 Tâches asynchrones 274
2.6 Exécuter les tâches asynchrones de manière concurrente 276
2.7 Notion d'awaitable 276
2.8 Précisions sur asyncio.sleep 276
3. Éléments de grammaire 277
3.1 Gestionnaire de contexte asynchrone 277
3.2 Générateurs asynchrones 278
3.3 Compréhensions asynchrones 278
3.4 Itération asynchrone 278
4. Notions avancées 278
4.1 Introspection 278
4.2 Gestion du résultat ou de l'exception 282
4.3 Annuler une tâche 283
4.4 Se prémunir d'une annulation 285
4.5 Timeouts 286
4.6 Gestion globale fine de l'attente 286

4.7 Module contextvars 289
5. Boucle événementielle asynchrone 292
5.1 Gestion de la boucle 292
5.2 Débogage 293
5.3 Notion de future 295
5.4 Annulation 297
5.5 Gestion des exceptions 298
6. Utiliser la boucle événementielle 302
6.1 Utilisation des fonctions de retour (callbacks) 302
6.2 Planifier des appels à des fonctions classiques 305
6.3 Utiliser des signaux 306
6.4 Synchronisation 309
6.5 Communication entre coroutines 313
6.6 Exemple : lecture de l'entrée standard 317
6.7 Programmation parallèle et asynchrone 318
6.8 Exécuter du code bloquant 318
7. L'asynchrone suivant les versions de Python 319
7.1 Python 3.7 319
7.2 Python 3.6 320
7.3 Python 3.5 320
7.4 Python 3.4 321
7.5 Python 3.3 et inférieur 321
7.6 Python 2.7 322
8. Cas concret 322
8.1 Exemple de travail 322
9. Exemple retravaillé en utilisant des générateurs 325
9.1 Télécharger en asynchrone 326
9.2 Parser en asynchrone 327
9.3 Faire plusieurs traitements en asynchrone 328
9.4 Utiliser un exécuteur 329
9.5 Pour aller plus loin 329

Chapitre 3.3
Programmation asynchrone : avancée

1. Transports et protocoles 331
   1.1 Introduction 331
   1.2 Transports 331
   1.3 Protocoles 335
   1.4 Mise en œuvre 336
   1.5 Exemple pour le protocole TCP 338
   1.6 Exemple pour le protocole SSL 346
   1.7 Exemple pour le protocole UDP 349
   1.8 Traiter d'autres types de protocoles réseau 352
   1.9 Exemple pour Subprocess 353
2. Flux 359
   2.1 Introduction 359
   2.2 Exemple avec le protocole TCP 359

Chapitre 3.4
Programmation asynchrone : alternatives

1. Gevent 363
   1.1 Introduction 363
   1.2 DNS 363
   1.3 Appels système 364
   1.4 Programmation asynchrone 365
2. Twisted 366
   2.1 Introduction 366
   2.2 Client/serveur TCP 366
   2.3 Serveur/client UDP 368
   2.4 AMP 369
   2.5 Autres protocoles 372
3. Trollius 372
4. HTTP asynchrone avec aiohttp 374
   4.1 Côté serveur 374
   4.2 Côté client 376

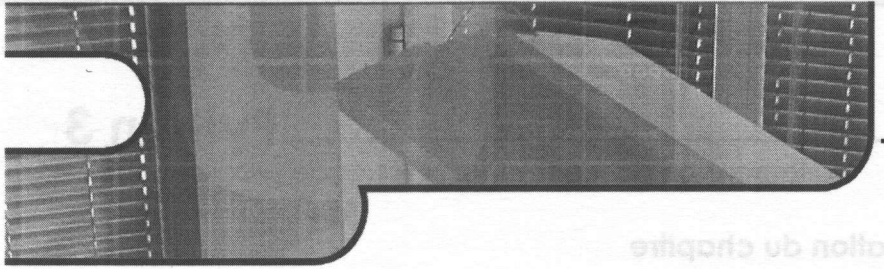
Chapitre 3.5
Programmation parallèle

1. Utilisation d'un fil d'exécution 377
1.1 Gestion d'un fil d'exécution 377
1.1.1 Présentation 377
1.1.2 Création 377
1.2 Gestion de plusieurs fils d'exécution 380
1.2.1 Lancement et contrôle 380
1.2.2 Opportunité d'utiliser un fil d'exécution 382
1.3 Résolution des problématiques liées 384
1.3.1 Synchronisation 384
1.3.2 Synchronisation conditionnelle 386
1.3.3 Sémaphore 388
2. Utilisation de processus 390
2.1 Gestion d'un processus 390
2.1.1 Présentation 390
2.1.2 Création 390
2.2 Gestion de plusieurs processus 393
2.2.1 Synchronisation 393
2.2.2 Paralléliser un travail 393
2.3 Résolution des problématiques liées 395
2.3.1 Communication interprocessus 395
2.3.2 Partage de données entre processus 397
2.4 Opportunité d'utiliser les processus 398
2.5 Démon 398
3. Exécution asynchrone 400
3.1 Introduction 400
3.2 Présentation 401

Chapitre 3.6
Programmation distribuée

1. Définitions 407
2. ØMQ 407
   2.1 Présentation générale 407
   2.2 Pair 409
   2.3 Client/serveur 410
   2.4 Publish/subscribe 411
   2.5 PUSH/PULL 415
   2.6 Patron pipeline 416
3. AMQP avec RabbitMQ 418
   3.1 Installation 418
   3.2 Configuration 418
   3.3 Introduction 419
   3.4 Notions importantes 421
   3.5 Publish/subscribe 422
   3.6 Routage et sujets 423
4. Celery 425
   4.1 Présentation 425
   4.2 Dans quel cas utiliser Celery ? 425
   4.3 Installation 425
   4.4 Configuration 426
   4.5 Utilisation 427
   4.6 Monitorer 429
5. Crossbar 429
   5.1 Présentation 429
   5.2 WebSocket 430
   5.3 Publish/subscribe 436

Index 439



Partie 1 : Traitement des données

Chapitre 1.1 Motifs de conception

1. Définition

1.1 Positionnement par rapport à la notion d'objet

Un objet ne fait pas l'application. Ce qui compte autant que les mécanismes permettant de travailler sur les objets, ce sont ceux qui permettent de gérer la manière dont ils interagissent en réponse à une problématique et à destination d'une fonctionnalité.

Pour ce faire, on utilise, consciemment ou non, des motifs de conception qui, chacun, correspondent à **une méthode pour faire interagir des objets entre eux**. Il y en a de plusieurs sortes, destinées à gérer des problématiques qui peuvent sembler parfois similaires mais qui ont toujours en réalité un contexte ou une destination particulière. Ils sont tellement nombreux qu'il faudrait un livre uniquement pour les présenter tous et cela n'aurait qu'un intérêt limité.

La plupart d'entre eux sont conçus pour répondre à une problématique précise. Ils sont la synthèse de retours d'expérience importants et significatifs et sont décrits avec précision par l'utilisation de plusieurs objets dont on décrit les rôles et les interactions.

Leur connaissance permet de standardiser la conception logicielle, de mettre en place des outils pour reproduire les bonnes solutions et d'améliorer projet après projet les processus même de conception logicielle par la maîtrise de solutions normalisées.

Cependant, la plupart des ouvrages de référence sur le sujet sont adaptés à des langages particuliers ou au mieux généralisés pour des types de langages particuliers comme les langages typés statiquement. Il convient donc de passer un peu de temps pour les analyser et les adapter à Python, ce qui peut produire des réponses originales.

1.2 Organisation du chapitre

L'objet de ce chapitre est de présenter une petite partie de ces motifs de conception (les plus connus et utilisés) de manière à ce que le développeur qui identifie une problématique puisse choisir d'utiliser l'un ou l'autre et partir directement vers la solution adaptée, ceci par le fruit d'une réflexion menée et de la connaissance des solutions les plus courantes. L'alternative étant la recherche par soi-même de la résolution du problème par lui-même, ce qui peut être stimulant pour des projets personnels, mais qui présente un risque pour des projets professionnels au pire d'un échec voire d'une réponse répondant fonctionnellement au problème mais peu performante ou au mieux une perte de temps.

Les motifs de conception abordés sont ceux qui sont le plus standardisés et ils se divisent en trois catégories structurantes :

- motifs de création ou de construction ;
- motifs de structuration ;
- motifs de comportement.

Les concepts seront présentés avec les problématiques concernées et les solutions proposées par Python et son modèle objet.

Les exemples choisis sont volontairement fonctionnellement simplistes de manière à ne pas être perturbé par une complexité fonctionnelle inutile et à ne pas regarder ailleurs que sur les éléments techniques purement liés au motif de conception, puisque c'est lui et lui seul qui importe.

De plus, outre le modèle objet classique de Python qui permet de créer ses propres solutions pour mettre en place un patron de conception, il existe la ZCA (*Zope Component Architecture*) qui offre des outils performants et puissants pour intégrer quelques patrons de conception par l'utilisation simple et efficace de fonctions qui réalisent le patron à la place du développeur. Ces solutions sont regroupées en fin de chapitre et ne sont donc pas abordées dans les sections intermédiaires.

1.3 Positionnement par rapport à d'autres concepts

Il est important de ne pas confondre les motifs de conceptions avec les motifs d'architecture. Ces derniers ne concernent pas les relations entre objets, mais entre composants et se positionnent donc un niveau au dessus. Il s'agit de définir une typologie d'objets. Ils peuvent cependant utiliser un à plusieurs motifs de conception.

Ainsi, par exemple, le motif d'architecture le plus connu est MCV et il utilise plusieurs motifs de conceptions qui sont Observateur, Stratégie et Composite.

D'autre part, il ne faut pas confondre les motifs de conception avec ce que l'on peut appeler les idiotismes de programmation, c'est-à-dire une construction syntaxique particulière pour répondre à un besoin particulier primitif comme poser une condition, itérer, valider une valeur, agréger une donnée, gérer une ressource...

© Editions ENI - All rights reserved

Par exemple, pour répéter x fois une même action, en C il faut écrire :

```
for(int i=0; i<100; i++) { ...
```

Alors qu'en Python, il faut écrire :

```
for i in range(100): ...
```

Ce qu'il faut retenir est que les motifs de conceptions se positionnent entre les deux concepts et concernent l'interaction entre les objets.

2. Motifs de création

2.1 Singleton

Un singleton est, en mathématiques, un ensemble ne contenant qu'un seul élément. En informatique, il s'agit d'une classe ne possédant qu'une seule instance.

```
>>> class Singleton:
...     instance = None
...     def __new__(cls):
...         if cls.instance is None:
...             cls.instance = object.__new__(cls)
...         return cls.instance
...
>>> object() is object(), Singleton() is Singleton()
(False, True)
```

Il faut noter que, la plupart du temps, les langages de programmation utilisent le singleton pour pallier le fait que leur modèle objet n'est pas suffisamment souple pour pouvoir gérer ce que Python fait déjà à l'aide de ses méthodes de classes.
Ainsi, l'utilisation d'un singleton en Python est extrêmement rare.

2.2 Fabrique

Présentation de la problématique

Lorsque l'on a une classe mère abstraite et plusieurs classes filles, la classe mère abstraite permet de capitaliser les comportements communs à toutes les filles. Pour les langages statiquement typés, une fonctionnalité utile est de potentiellement travailler avec des objets déclarés comme étant du type de la mère de manière à ce que les comportements puissent être homogénéisés. En clair, que l'on ne soit pas obligé de dupliquer du code autant de fois qu'il y ait de filles.
Pour cela, on utilise une fabrique qui va alors prendre en paramètre les données nécessaires à la construction de l'objet, déterminer à partir de critères déterministes la classe fille à instancier, créer cette instance, et la renvoyer. Cette instance portera alors le type de sa mère (voir notion de polymorphisme).

Solution

Python est un langage à typage dynamique. Il ne dispose pas de contraintes lui imposant de travailler avec un type de données particulier – à moins que le développeur l'ait lui-même spécifié dans son code – et il utilise naturellement la notion de « Duck Typing ».
En ce sens, les problématiques purement techniques réglées par la solution de la fabrique ne lui apportent rien, puisque celles-ci sont pour lui inexistantes.
Par contre, il peut toujours être utile de créer une fonction, ou une méthode de classe qui soit capable de renvoyer une instance d'un type précis en fonction de conditions déterministes :

```
>>> class A:
...     pass
...
>>> class B:
...     pass
...
>>> def fabrique(param):
...     if param % 2 == 0:
...         return A()
...     else:
...         return B()
...
```

Cet exemple est volontairement simpliste pour montrer l'absence de contraintes.
S'il n'y a pas de contraintes naturelles en Python, rien n'interdit par contre au développeur de poser un cadre rigide à ses développements, de manière à se créer un ensemble fortement structuré de classes et à créer par la suite une fabrique.
Il suffit de rajouter une simple fonction (telle que celle-ci dessus) pour avoir notre fabrique et effectuer un choix parmi les classes filles à utiliser. En effet, rien n'impose à Python de faire en sorte qu'une fabrique soit obligatoirement une classe.
Ceci dit, le modèle objet de Python étant extrêmement complet, il est possible d'aller beaucoup plus loin, c'est-à-dire d'intégrer la fabrique directement au cœur du processus de création de la classe mère.
Concrètement, on instancie un objet de la classe mère, et l'on se retrouve avec un objet de la bonne classe fille, choisie de manière déterministe. Cela peut être bien fait, puisque la méthode de discrimination entre les classes filles est portée par la mère, mais en fonction d'un paramètre que chaque fille précise. L'initialisation de la classe est également portée par la mère, car elle est commune.
Voici donc le code de la classe mère et de trois classes filles :

```
>>> import abc
>>> import os.path
>>> class Loader(metaclass=abc.ABCMeta):
...     def __new__(cls, filename):
...         ext = os.path.splitext(filename)[-1]
...         for sub in cls.__subclasses__():
...             if sub.isDesignedFor(ext):
```

© Editions ENI - All rights reserved

```
...                     o = object.__new__(sub)
...                     o.__init__(filename)
...                     return o
...         def __init__(self, filename):
...             self.filename = filename
...         @classmethod
...         def isDesignedFor(cls, ext):
...             if ext in cls.extensions:
...                 return True
...             return False
...         @abc.abstractmethod
...         def load(self):
...             return
...
>>> class TextLoader(Loader):
...     extensions = ['.txt']
...     def load(self):
...         print('Fichier textuel')
... #       with open(self.filename) as f:
... #           return f.readlines()
...
>>> import csv
>>> class CSVLoader(Loader):
...     extensions = ['.csv']
...     def load(self):
...         print('Fichier CSV')
... #       with open(self.filename) as f:
... #           return cvs.reader(f.read())
...
>>> import pickle
>>> class PickleLoader(Loader):
...     extensions = ['.pkl']
...     def load(self, filename):
...         print('Fichier Pickle')
... #       with open(self.filename) as f:
... #           return pickle.load(f)
...
```

Et voici ce qu'il se passe lorsque l'on instancie la classe mère avec des paramètres pris en compte par l'une des classes filles :

```
>>> loader = Loader('fichier.txt')
>>> type(loader)
<class '__main__.TextLoader'>
>>> loader = Loader('fichier.pkl')
>>> type(loader)
<class '__main__.PickleLoader'>
>>> loader = Loader('fichier.csv')
>>> type(loader)
```

```
<class '__main__.CSVLoader'>
```

Voici également ce qu'il se passe lorsqu'aucune fille ne peut traiter le paramètre passé au constructeur :

```
>>> loader = Loader('fichier.existepas')
>>> type(loader)
<class 'NoneType'>
```

Du coup, il suffit, après instanciation, de vérifier que l'on a bien quelque chose de différent de None et l'on peut alors utiliser l'instance à volonté.

Conséquences

Pour du développement simple, il est facile de créer un composant capable de retourner une instance, de quelque nature que ce soit. Il s'agit alors d'une fabrique sans en être vraiment une. Il est également possible de créer une fabrique dans les règles de l'art, mais qui n'est pas aussi indispensable qu'elle l'est pour les langages statiquement typés.
Par contre, lorsque l'on va aussi loin que le dernier exemple, ce concept prend une autre dimension et utilise alors à pleine capacité le modèle objet de Python.

2.3 Fabrique abstraite

Présentation de la problématique

La problématique de la fabrique abstraite est la problématique de la fabrique posée sur la notion même de fabrique. En clair, il s'agit de créer une fabrique sur un ensemble de fabriques, donc d'appliquer le motif de conception vu précédemment à la fabrique elle-même.
L'idée est de regrouper les fabriques relatives à un contexte dans une seule et même classe (en Python, ces fabriques peuvent être des méthodes de classe), puis d'homogénéiser les fabriques des différents contextes en les faisant hériter d'une seule classe mère abstraite.
Un exemple pourrait être une fabrique pour créer des composants graphiques pour une IHM (bouton, zone de saisie, tableau...). On aurait alors une fabrique pour chacun de ces éléments et toutes seraient regroupées au sein d'une seule et même classe. Et ce travail pourrait être réalisé pour chaque contexte graphique, c'est-à-dire une classe pour TkInter, une autre pour PyGTK, une autre pour WxPython...

Solution

Pour Python, cette fabrique abstraite est simplement une fabrique comme les autres. La seule différence est que les méthodes regroupées au sein de la classe sont des méthodes qui sont elles-mêmes des fabriques simples.

© Editions ENI - All rights reserved

2.4 Monteur

Présentation de la problématique

La problématique tourne autour du moyen de formaliser un certain nombre de méthodes d'initialisation d'un objet potentiellement complexe, l'idée étant de ne pas manipuler directement l'objet lui-même, mais que l'on passe par l'une de ces méthodes formalisées. Chacune de ces méthodes s'appelle un **monteur**, l'objet créé est nommé le **produit** et le ou les objets susceptibles d'utiliser un monteur pour récupérer le produit et l'utiliser sont nommés **directeur**.

Les différentes méthodes d'initialisation du produit peuvent potentiellement être rapprochées sous forme de classes ayant pour mère une classe abstraite. On parle alors de **monteur** pour la classe abstraite et de **monteur concret** pour chaque classe fille.

L'avantage de cette méthode est de séparer clairement – isoler – le produit du directeur, c'est-à-dire l'objet utilisé de celui qui l'utilise en obligeant à passer par une méthode prédéfinie pour l'initialisation du produit.

Solution

Le produit peut être créé simplement par une instanciation avec beaucoup de paramètres et étant donné les très grandes possibilités de Python pour le passage de paramètres, les solutions sont nombreuses.

Ce que propose ce motif de conception est d'externaliser les méthodes de créations et de les capitaliser au sein de classes qui sont des monteurs. Les possibilités de Python permettraient d'utiliser éventuellement autre chose que des classes.

La problématique consistant également en la maîtrise de l'isolation entre le produit et le directeur, le parti a été pris dans l'exemple qui suit d'utiliser les propriétés plutôt que des attributs et des fonctions, ce qui permet de mettre en place un moyen de contrôle sur les accès et les modifications, mais également de ne pas alourdir la verbosité en obligeant à utiliser des méthodes **get** ou **set**.

L'exemple colle au nom des concepts. Il y a une classe produit qui porte deux notions et des classes monteur qui les paramètrent. La manière d'utiliser chaque monteur est identique et le directeur ne voit donc pas de différences.

```
>>> import abc
>>> class Produit:
...     @property
...     def forme(self):
...         return self._forme
...     @forme.setter
...     def forme(self, forme):
...         self._forme = forme
...     @property
...     def couleur(self):
...         return self._couleur
...     @couleur.setter
...     def couleur(self, couleur):
...         self._couleur = couleur
```

```
...     def __str__(self):
...         return 'Produit forme=%s couleur=%s' %
(self.forme, self.couleur)
...
```

Voici la classe monteur abstraite qui se charge de créer le produit. Contrairement à la classe fabrique où il s'agit de créer des classes différentes en fonctions de paramètres, on crée ici une seule et même classe, mais en la paramétrant de manière différente. La classe abstraite porte donc logiquement elle-même la méthode de création du produit, mais délègue à ses classes filles son paramétrage :

```
>>> class Monteur:
...     @property
...     def produit(self):
...         return self._produit
...     @produit.setter
...     def produit(self, produit):
...         self._produit = produit
...     def creerProduit(self):
...         self.produit = Produit()
...     @abc.abstractmethod
...     def concevoirForme(self):
...         return
...     @abc.abstractmethod
...     def concevoirCouleur(self):
...         return
...
>>> class MonteurCubeBleu(Monteur):
...     def concevoirForme(self):
...         self.produit.forme = "Cube"
...     def concevoirCouleur(self):
...         self.produit.couleur = "Bleu"
...
>>> class MonteurSphereRouge(Monteur):
...     def concevoirForme(self):
...         self.produit.forme = "Sphere"
...     def concevoirCouleur(self):
...         self.produit.couleur = "Rouge"
...
>>> class MonteurPyramideJaune(Monteur):
...     def concevoirForme(self):
...         self.produit.forme = "Pyramide"
...     def concevoirCouleur(self):
...         self.produit.couleur = "Jaune"
...
```

Le directeur va donc être lié à un monteur qui est un attribut et porter une méthode pour prendre à son compte toute la procédure de création/paramétrage en une seule méthode.

© Editions ENI - All rights reserved

```
>>> class Directeur:
...     @property
...     def monteur(self):
...         return self._monteur
...     @monteur.setter
...     def monteur(self, monteur):
...         self._monteur = monteur
...     def concevoirProduit(self):
...         self.monteur.creerProduit()
...         self.monteur.concevoirForme()
...         self.monteur.concevoirCouleur()
...         return self.monteur.produit
...
```

Pour utiliser une telle classe, il faut donc instancier le directeur, lui adjoindre un monteur, puis lancer la méthode de création/paramétrage :

```
>>> directeur = Directeur()
>>> directeur.monteur = MonteurPyramideJaune()
>>> produit = directeur.concevoirProduit()
```

On retrouve ainsi notre produit :

```
>>> print(produit)
Produit forme=Pyramide couleur=Jaune
```

Conséquences

Ce motif de conception est très différent de la fabrique et en conséquence, il est employé pour des raisons différentes.
Il reste assez simple d'utilisation et peut être largement intégré au modèle objet de Python et utilisé dès lors que le besoin est de définir différentes méthodes pour créer et paramétrer un même objet.
Créer un directeur dont le monteur serait choisi lors de son initialisation n'est pas une bonne idée. Le choix du monteur est indépendant et doit pouvoir être modifié à volonté.

2.5 Prototype

Présentation de la problématique

Il se peut que la création d'une instance soit complexe (fasse intervenir un certain nombre de calculs, de création d'autres objets...) ou qu'elle soit consommatrice de temps.
Pour pallier ce problème, le motif de conception par prototype permet de remplacer le processus d'instanciation par la création d'une première instance et un clonage de celle-ci pour toutes les autres créations d'instance.
Il faut avoir à l'esprit que la notion de clonage est propre à chaque langage et est à l'origine prévue pour résoudre des problématiques particulières.

Solution

La solution s'opère en deux temps. Le premier temps consiste à créer la première instance (et donc de détecter le fait que la première instance ait été créée ou non) et le second temps consiste à prévoir une méthode de clonage efficace et sans perte de données.
Voici un exemple simple avec plusieurs types d'attributs pour tester le clonage :

```
>>> class A:
...     pass
...
>>> class NonPrototype:
...     def __init__(self):
...         # Méthode complexe de création
...         self.a = 42
...         self.b = 'Complexe'
...         self.c = A()
...         self.c.a = [1, 2, 3]
...         self.c.b = A()
...         self.c.b.a = (1, 2, 3)
...     def __str__(self):
...         return 'Prototype {self.a}, {self.b}, {self.c.a},
{self.c.b.a}'.format(self=self)
...
>>> a = NonPrototype()
>>> print(a)
Prototype 42, Complexe, [1, 2, 3], (1, 2, 3)
```

Voici comment mettre en place le motif de conception prototype sur cet objet :

```
>>> from copy import deepcopy
>>> class Prototype:
...     _instance_reference = None
...     def __new__(cls):
...         if cls._instance_reference is not None:
...             print('Clonage')
...             result = object.__new__(cls)
...             result.__dict__ =
deepcopy(cls._instance_reference.__dict__)
...             return result
...         result = object.__new__(cls)
...         cls._instance_reference = result
...         return result
...     def __init__(self):
...         if self._instance_reference is None:
...             return
...         self._instance_reference = None
...         print('Initialisation')
...         # Méthode complexe de création
...         self.a = 42
...         self.b = 'Complexe'
```

© Editions ENI - All rights reserved

```
...             self.c = A()
...             self.c.a = [1, 2, 3]
...             self.c.b = A()
...             self.c.b.a = (1, 2, 3)
...         def __str__(self):
...             return '{self.__class__.__name__} {0}, {self.a},
{self.b}, {self.c.a}, {self.c.b.a}'.format(id(self), self=self)
...
```

L'idée principale du moyen technique mis en œuvre est de ne garder l'instance de référence que dans l'attribut de classe, mais que cet attribut soit vide pour chaque instance de manière que celles-ci ne référencent pas la première d'entre elles.
La méthode de création de la classe va alors la première fois qu'elle est utilisée suivre son processus de création classique, mais garder une référence vers l'instance créée.
Pour toutes les autres fois, elle va créer un objet, mais dupliquer son contenu à partir de celui de la toute première instance.
Lorsque l'on n'a jamais encore créé d'instances :

```
>>> print(Prototype._instance_reference)
None
```

On crée la première instance :

```
>>> b = Prototype()
Initialisation
```

Les valeurs sont correctement initialisées :

```
>>> print(b)
Prototype 20707280, 42, Complexe, [1, 2, 3], (1, 2, 3)
```

La classe possède bien une référence vers cette instance, mais l'instance elle-même n'a pas de référence vers elle :

```
>>> print(Prototype._instance_reference, b._instance_reference)
Prototype 20707280, 42, Complexe, [1, 2, 3], (1, 2, 3) None
```

On crée un second objet :

```
>>> c = Prototype()
Clonage
```

Celui-ci a été cloné ; il est passé par la méthode d'initialisation (**`__init__`**) car celle-ci est automatiquement appelée si la méthode de construction (**`__new__`**) renvoie un objet du bon type, mais la toute première ligne de cette méthode fait en sorte que l'on quitte tout de suite. Au final, nos valeurs sont bien présentes :

```
>>> print(c)
Prototype 20707856, 42, Complexe, [1, 2, 3], (1, 2, 3)
```

Et la nouvelle instance ne contient toujours pas de références vers l'instance première :

```
>>> print(Prototype._instance_reference, c._instance_reference)
Prototype 20707280, 42, Complexe, [1, 2, 3], (1, 2, 3) None
```

Conséquences

La solution proposée réalise tout le processus au sein de la même classe. La solution habituellement présentée fait intervenir un objet tiers pour la construction des classes et cet objet doit appeler une méthode clone. L'avantage de la solution présentée ici est de toujours appeler le constructeur et de le laisser gérer le fait qu'il initialise l'objet ou le copie par prototype. Cela est transparent pour celui qui instancie l'objet.

3. Motifs de structuration

3.1 Adaptateur

Présentation de la problématique

Pour avoir des traitements génériques, lorsque l'on conçoit une architecture, la solution permettant de disposer d'une interface commune et de créer les objets qui vont fournir cette interface est l'idéal.
Seulement, on travaille rarement uniquement avec des objets que l'on a conçus, on travaille également avec des bibliothèques tierces, ou des objets conçus préalablement qui sont adaptés à une problématique autre que la nôtre.
Dans tous les cas, il n'est pas possible de reprendre ces objets pour les faire entrer dans un moule qui satisfait parfaitement nos besoins.
Dans ce cadre-là, une solution largement diffusée est de créer des adaptateurs qui vont adapter le comportement des objets que l'on a à une interface unique.

Solution

Voici un exemple de classes qui sont parfaitement adaptées à une certaine utilisation et que nous souhaitons reprendre, mais utiliser d'une manière générique :

```
>>> class Chien:
...     def aboyer(self):
...         print('Ouaff')
...
>>> class Chat:
...     def miauler(self):
...         print('Miaou')
...
>>> class Cheval:
...     def hennir(self):
...         print('Hiiii')
...
>>> class Cochon:
...     def grogner(self):
...         print('Gruik')
...
```

Notre souhait est de faire « parler » ces animaux d'une manière générique.

© Editions ENI - All rights reserved

Voici une classe, qui correspond à l'interface souhaitée :

```
>>> import abc
>>> class Animal(metaclass=abc.ABCMeta):
...     @abc.abstractmethod
...     def faireDuBruit(self):
...         return
...
```

On pourrait alors reprendre les quatre classes précédentes et les réécrire avec la même méthode, mais cela entraînerait une perte au niveau sémantique alors que c'est potentiellement utile pour d'autres utilisations.
Python propose alors plusieurs solutions. L'une des plus communes consiste simplement en l'utilisation de l'héritage multiple. Il offre là une réponse simple et efficace en surchargeant la méthode abstraite pour la rediriger vers la bonne méthode :

```
>>> class ChienAlternative(Animal, Chien):
...     def faireDuBruit(self):
...         return self.aboyer()
...
```

Ce qui peut également s'effectuer plus simplement (une méthode étant un attribut) :

```
>>> class ChatAlternative(Animal, Chat):
...     faireDuBruit = Chat.miauler
...
```

Ceci fonctionne, mais ne correspond pas au motif de conception Adaptateur dont on peut se rapprocher ainsi :

```
>>> class ChevalAlternative(Animal):
...     def __init__(self, cheval):
...         self.cheval = cheval
...     def faireDuBruit(self):
...         return self.cheval.hennir()
...     def __getattr__(self, attr):
...         return self.cheval.__getattr__(attr)
...
```

Voici l'adaptateur qui n'hérite de rien et ne fait que de la redirection de méthode, ce qui peut s'écrire en Python uniquement dans la méthode **`__getattr__`** :

```
>>> class CochonAdaptateur:
...     def __init__(self, cochon):
...         self.cochon = cochon
...     def __getattr__(self, attr):
...         if attr == 'faireDuBruit':
...             return self.cochon.grogner
...         return getattr(self.cochon, attr)
...
```

Voici donc l'utilisation successive de ces classes, avec les deux alternatives et les deux adaptateurs (il faut noter leur différence dans le processus d'instanciation) :

```
>>> for animal in [ChienAlternative(), ChatAlternative(),
ChevalAlternative(Cheval()), CochonAdaptateur(Cochon())]:
...     animal.faireDuBruit()
...
Ouaff
Miaou
Hiiii
Gruik
```

Conséquences

L'adaptateur peut être vu comme un composant permettant, comme son nom l'indique, d'adapter un composant existant à une interface imposée qui diffère. Cependant, au niveau purement technique, il est surtout utile lorsqu'il est utilisé en collaboration avec une fabrique, puisque cette dernière peut alors choisir de quelle manière adapter une classe. Ainsi, le processus d'adaptation peut s'opérer non pas directement lors de l'instanciation, mais à la demande, lorsque l'on en a besoin.

```
>>> def animal_adapterFactory(context):
...     if isinstance(context, Chien):
...         return ChienAdaptateur(context)
...     elif isinstance(context, Chat):
...         return ChatAdaptateur(context)
...     elif isinstance(context, Cheval):
...         return ChevalAdaptateur(context)
...     elif isinstance(context, Cochon):
...         return CochonAdaptateur(context)
...     else:
...         raise Exception('Adaptateur non trouvé')
...
>>> for animal in [Chien(), Chat(), Cheval(), Cochon()]:
...     animal_adapterFactory(animal).faireDuBruit()
...
Ouaff
Miaou
Hiiii
Gruik
```

Ceci peut également vouloir dire qu'un objet peut être adapté de plusieurs manières à des moments différents, pour répondre à des problématiques différentes.

L'adaptateur n'est donc plus uniquement un motif de conception à utiliser a posteriori. Dès la phase de conception, il peut être prévu de créer des objets au fort sens sémantique et de les adapter en fonction des besoins.

Dernier point, une adaptation ne veut pas forcément dire que chaque méthode à adapter équivaut à une méthode adaptée. Il se peut qu'une adaptation demande un travail plus conséquent.

© Editions ENI - All rights reserved

3.2 Pont

Présentation de la problématique

Le pont est un motif de conception ayant pour destination le découplage entre l'interface et son implémentation, ce qui permet de fusionner les fonctionnalités de deux types de classes ayant des hiérarchies orthogonales.

Par exemple, soit trois types de données :

- ville ;
- département ;
- région.

Avec deux possibilités de charger et/ou stocker ces données :

- CSV ;
- Pickle.

Il est alors possible de se baser sur l'un de ces concepts puis de créer une classe pour chaque cas d'utilisation nécessitant l'autre concept.

On a donc six classes :

- csvVilles ;
- csvDepartements ;
- csvRegions ;
- pickleVilles ;
- pickleDepartements ;
- pickleRegions.

Bien évidemment, une partie du code de chaque classe est redondante par rapport à l'un ou l'autre des concepts.

Python propose éventuellement des solutions grâce à l'héritage multiple, permettant de définir trois classes pour gérer la partie données pour les villes, les départements et les régions et deux autres classes pour gérer la partie chargement/enregistrement pour le CSV et le Pickle. Il ne reste plus qu'à créer les six classes héritant chacune des combinaisons des deux concepts.

Cette manière de faire reste simple, très basique, mais pas forcément très lisible ou même évolutive, parce que si l'on rajoute un concept ou une nouvelle classe pour un concept, il faut alors créer toutes les classes nécessaires pour prendre en compte ce changement, ce qui peut être une opération fastidieuse.

La solution consiste donc à utiliser le pont qui, par la syntaxe lors de son utilisation, peut vaguement ressembler à l'adaptateur, mais qui est vraiment différent. Car il ne s'agit pas de pointer une sémantique vers une autre, mais de permettre le découplage de plusieurs notions au sein d'une même classe.

Solution

Voici un exemple avec deux concepts à utiliser et deux notions chacun. Le premier concept concerne la nature de la donnée à charger :

```
>>> import abc
>>> class Loader(metaclass=abc.ABCMeta):
...     @abc.abstractmethod
...     def load(self):
...         return
...
>>> #import csv
... class CSVLoader(Loader):
...     def load(self, filename):
...         print('Fichier CSV')
... #       with open(filename) as f:
... #           return cvs.reader(f.read())
...
>>> #import pickle
... class PickleLoader(Loader):
...     def load(self, filename):
...         print('Fichier Pickle')
... #       with open(filename) as f:
... #           return pickle.load(f)
...
```

Cette première série de classes présente une relation de mère à fille. La classe concrète est l'implémentation abstraite de l'interfaçage entre la donnée stockée sous forme persistante et celle sous forme d'objet manipulable. Les deux objets sont les dérivées concrètes.
La seconde série de classes sont les ponts, puisqu'ils permettent de traiter les données en faisant abstraction de leur provenance, mais en y appliquant la même transformation :

```
>>> class Transformer(metaclass=abc.ABCMeta):
...     @abc.abstractmethod
...     def transform(self):
...         return
...
```

La méthode loadDatas est donc une méthode dépendante de l'implémentation, c'est-à-dire des trois classes Loader, on dit aussi qu'elle est de bas niveau. La méthode est dépendante uniquement de sa classe abstraite, on dit qu'elle est de haut niveau :

```
>>> class UpperTransformer(Transformer):
...     def __init__(self, filename, *args, loader):
...         self.filename = filename
...         self.loader = loader
...     def loadDatas(self):
...         self.content = self.loader.load(self.filename)
...         # Au cas où il y ait des commentaires dans le loader
...         if self.content is None:
...             self.content = [
```

© Editions ENI - All rights reserved

```
...                 ['Truc', 'machin'],
...                 ['cHoSe', 'BIBULE']]
...     def transform(self):
...         for i, l in enumerate(self.content):
...             for j, d in enumerate(l):
...                 self.content[i][j] = d.upper()
...
>>> class LowerTransformer(Transformer):
...     def __init__(self, filename, *args, loader):
...         self.filename = filename
...         self.loader = loader
...     def loadDatas(self):
...         self.content = self.loader.load(self.filename)
...         # Au cas où il y ait des commentaires dans le loader
...         if self.content is None:
...             self.content = [
...                 ['Truc', 'machin'],
...                 ['cHoSe', 'BIBULE']]
...     def transform(self):
...         for i, l in enumerate(self.content):
...             for j, d in enumerate(l):
...                 self.content[i][j] = d.lower()
...
```

Voici comment utiliser un tel pont :

```
>>> test1 = UpperTransformer('test.csv', loader=CSVLoader())
```

Le composant d'implémentation est passé en paramètres.
Il ne reste plus qu'à utiliser les méthodes. Celle de bas niveau :

```
>>> test1.loadDatas()
Fichier CSV
```

Et celle de haut niveau :

```
>>> test1.transform()
```

On peut également voir à quoi ressemblent les données ainsi traitées :

```
>>> test1.content
[['TRUC', 'MACHIN'], ['CHOSE', 'BIDULE']]
```

Voici l'utilisation de ces composants pour un tout autre contexte :

```
>>> test2 = LowerTransformer('test.pkl', loader=PickleLoader())
>>> test2.loadDatas()
Fichier Pickle
>>> test2.transform()
>>> test2.content
[['truc', 'machin'], ['chose', 'bidule']]
```

Conséquences

Étant donné qu'en Python, tout est objet, qu'une classe ou une fonction sont elles-mêmes des objets, il existe des solutions plus simples qui consistent à passer la classe elle-même en paramètre d'une méthode, par exemple. Les choix d'architecture sont relativement nombreux. On préfère ainsi créer des composants autonomes, parfaitement découplés et les tisser entre eux dans un second temps, plutôt que d'en créer un et d'introduire une notion de bas niveau et haut niveau, l'un utilisant l'autre.
Les solutions utilisant l'héritage multiple peuvent être dans certains cas avantageuses, mais ce ne sont pas forcément celles à privilégier.

3.3 Composite

Présentation de la problématique

L'objet composite est un motif de conception ayant pour destination la constitution d'un tronc commun à plusieurs objets similaires pour permettre une manipulation générique de ces objets. Il est souvent utilisé pour concevoir une structure arborescente.
Le composant est la classe abstraite de tout composant, le composite est un composant qui peut en contenir d'autres au contraire de la feuille qui est dit terminal.

Solution

La solution, avec Python, est d'utiliser simplement une classe abstraite contenant les méthodes communes et de surcharger ces méthodes dans le composite et dans la feuille. Ces deux objets seuls permettent de représenter l'arbre.
Voici un composant muni seulement d'une méthode lui permettant de se décrire :

```
>>> import abc
>>> class Composant(metaclass=abc.ABCMeta):
...     def __init__(self, name):
...         self.name = name
...     @abc.abstractmethod
...     def verbose(self, level=0):
...         return
...
```

La feuille surcharge la méthode abstraite :

```
>>> class Feuille(Composant):
...     def verbose(self, level=0):
...         return '%sFeuille %s' % ('\t' * level, self.name)
...
```

Le composite aussi, mais il rajoute des comportements supplémentaires :

```
>>> class Composite(Composant):
...     def __init__(self, name):
...         Composant.__init__(self, name)
...         self.contenu = []
...     def add(self, composant):
...         self.contenu.append(composant)
```

© Editions ENI - All rights reserved

```
...     def verbose(self, level=0):
...         feuilles = [f.verbose(level+1) for f in self.contenu]
...         feuilles.insert(0, '%sComposite %s' % ('\t' *
level, self.name))
...         return '\n'.join(feuilles)
...
```

Voici la partie cliente, qui utilise notre motif de conception.
On commence par créer deux feuilles :

```
>>> c1 = Feuille('F1')
>>> c2 = Feuille('F2')
```

Puis un composite :

```
>>> c3 = Composite('C1')
```

Auquel il est possible de rajouter des feuilles :

```
>>> c3.add(Feuille('F4'))
>>> c3.add(Feuille('F5'))
>>> c3.add(Feuille('F6'))
```

Il est également possible de créer un composite auquel on ajoute d'autres composites :

```
>>> c4 = Composite('C2')
>>> c41 = Composite('C3')
```

Afin d'aller plus vite, on ajoute directement ces composites via la modification de l'attribut qui les contient :

```
>>> c41.contenu = [Feuille('F7'), Feuille('F8'), Feuille('F9')]
>>> c4.contenu = [Composite('C4'), c41, Feuille('FA')]
```

On peut à chaque stade de la création vérifier ce que répond la méthode de description.
Mais il est également possible de tout regrouper sous une même racine :

```
>>> main = Composite('Test')
>>> main.contenu.extend([c1, c2, c3, c4])
```

Ce qui donne :

```
>>> print(main.verbose())
Composite Test
   Feuille F1
   Feuille F2
   Composite C1
     Feuille F4
     Feuille F5
     Feuille F6
   Composite C2
     Composite C4
     Composite C3
        Feuille F7
        Feuille F8
        Feuille F9
     Feuille FA
```

Conséquences

Il est possible d'être plus rigoureux dans l'utilisation des attributs, par exemple, mais il est aussi possible de l'être moins, en n'utilisant pas abc par exemple.
Quoi qu'il en soit, cette façon de faire, plus ou moins formalisée, est en Python quasi naturelle et très simple à mettre en place.

3.4 Décorateur

Présentation de la problématique

La destination d'un décorateur est de rajouter dynamiquement des fonctionnalités et de le faire par la composition plutôt que par l'héritage.
Le concept s'exprime mathématiquement par « rond ». Ainsi, l'expression mathématique **(décorateur o fonction) (params)** peut aussi s'exprimer en informatique par l'expression **decorateur(fonction)(params)**.
Un décorateur est donc une fonction qui transforme une fonction en une autre fonction, voire une fonction qui transforme une classe en une autre classe.
La problématique consiste à gérer la manière de composer les fonctionnalités et elle est non triviale à résoudre pour un langage classique. Mais pas pour Python grâce au fait que tout est objet, y compris les classes et les fonctions.
Les décorateurs sont donc une alternative qui est certes complexe à maîtriser, mais extrêmement séduisante.
Pour cette raison, ils sont devenus un élément essentiel du langage et ont même une syntaxe appropriée pour les appliquer.

Solution

Voici l'exemple le plus simple avec un décorateur identité (il renvoie la fonction qu'il reçoit en paramètre) et un décoré :

```
>>> def decorator(func):
...     return func
...
>>> @decorator
... def decorated(param):
...     pass
...
```

Ceci est fonctionnellement et techniquement équivalent à :

```
>>> def to_decorate(param):
...     pass
...
>>> decorated = decorator(to_decorate)
```

Ainsi, lorsqu'un appel avec la fonction décorée telle que déclarée précédemment est effectué, l'équivalent de ce qui suit :

```
>>> result = decorated(value)
```

© Editions ENI - All rights reserved

N'est absolument pas ceci :

```
>>> result = decorator(to_decorate(value))
```

Mais plutôt :

```
>>> result = decorator(to_decorate)(value)
```

La différence entre les deux est fondamentale et il faut revenir à la définition.
Voici un exemple plus complet où l'on voit à la fois comment passer un paramètre à un décorateur (**param**) et comment gérer ceux de la fonction originelle (**arg**) :

```
>>> def decorator(param):
...     def wrapper(func):
...         def wrapped(arg):
...             result = func(arg)
...             return result > param and result or param
...         return wrapped
...     return wrapper
...
```

Pour l'appliquer, il suffit de procéder ainsi :

```
>>> @decorator(20)
... def calcul(arg):
...     return arg
...
```

Ce décorateur fait en sorte de poser une barrière minimale à un calcul, celle-ci étant le paramètre passé au décorateur :

```
>>> calcul(40)
40
>>> calcul(10)
20
```

La décoration d'une fonction est une opération qui modifie en profondeur la fonction, y compris ses métadonnées. Normalement, lorsque l'on a une fonction, on a ceci :

```
>>> def example():
...     """Example docstring"""
...
>>> example.__name__
'example'
>>> example.__doc__
'Example docstring'
```

Voici ce qu'il se passe lorsque l'on décore la fonction :

```
>>> def my_decorator(f):
...     """Decorator docstring"""
...     def wrapper(*args, **kwargs):
...         """Wrapper docstring"""
...         return f(*args, **kwargs)
...     return wrapper
```

```
...
>>> @my_decorator
... def example():
...     """Example docstring"""
...
>>> example.__name__
'wrapper'
>>> example.__doc__
'Wrapper docstring'
```

Voici donc une solution pour que la fonction décorée ressemble à l'originale :

```
>>> def my_decorator(f):
...     """Decorator docstring"""
...     @functools.wraps(f)
...     def wrapper(*args, **kwargs):
...         """Wrapper docstring"""
...         return f(*args, **kwargs)
...     return wrapper
...
>>> @my_decorator
... def example():
...     """Example docstring"""
...
>>> example.__doc__
'Example docstring'
>>> example.__name__
'example'
```

Conséquences

Maîtriser la création de décorateur demande de parfaitement maîtriser le modèle objet de Python et de réaliser ses propres expériences pour cerner vraiment toute l'étendue du concept. Par contre, l'utilisation d'un décorateur est, en Python, la solution privilégiée pour répondre élégamment à énormément de cas d'utilisation car il est extrêmement efficace. Il est d'ailleurs utilisé pour des fonctionnalités aussi importantes que la transformation de méthodes pour les rendre statiques ou de classe.

Dans cet ouvrage, plusieurs décorateurs sont écrits pour répondre à des problématiques et de nombreux autres sont appliqués dans des exemples portant sur tous les sujets.

3.5 Façade

Présentation de la problématique

La destination du motif de conception façade est de masquer la complexité d'un système en offrant un objet simple permettant de répondre aux problématiques nécessaires à la majorité des utilisateurs.

La façade réalise en quelque sorte l'interface entre le programme principal et un module très complexe, par exemple. Ses fonctionnalités peuvent croiser celles de plusieurs composants pour les réunir dans la même façade.

© Editions ENI - All rights reserved

Solution

Pour commencer, il faut créer quelques classes qui interagissent entre elles pour répondre à une fonctionnalité. De plus, on complexifie un peu les choses de manière à mieux apprécier notre façade :

```
>>> class Word:
...     def hello(self):
...         return 'Hello, I\'m'
...     def goodbye(self):
...         return 'GoodBye, I\'m'
...
>>> class Speaker:
...     def __init__(self, name):
...         self.name = name
...     @classmethod
...     def say(cls, what, to):
...         word = Word()
...         methode = getattr(word, what)
...         if methode is None:
...             return ''
...         return ' '.join([methode(), to])
...     def speak(self, what):
...         return Speaker.say(what, self.name)
...     def who(self):
...         return self.name
...
>>> class Dialog:
...     def __init__(self, speaker1, speaker2):
...         self.speaker1 = Speaker(speaker1)
...         self.speaker2 = Speaker(speaker2)
...         self.sentences = []
...     def __call__(self):
...         sentences = []
...         sentences.append(self.speaker1.speak('hello'))
...         sentences.append(self.speaker2.speak('hello'))
...         sentences.extend(self.sentences)
...         sentences.append(self.speaker1.speak('goodbye'))
...         sentences.append(self.speaker2.speak('goodbye'))
...         return '\n'.join(['- %s' % s for s in sentences])
...
```

On a donc trois classes et l'on veut proposer une interface simple à un développeur qui utiliserait nos classes, les deux fonctionnalités essentielles étant « faire dire quelque chose à quelqu'un », puis pour « initier un dialogue » :
Voici donc une façade appropriée :

```
>>> class Facade:
...     @classmethod
...     def say(cls, what, to):
```

```
...         print(Speaker.say(what, to))
...     def dialog(self, speaker1, speaker2, sentences):
...         dialog = Dialog(speaker1, speaker2)
...         dialog.sentences = sentences
...         print(dialog())
...
```

La façade respecte les classes qu'elle utilise, et propose deux méthodes aux signatures simples. La première n'est qu'une redirection vers une méthode d'une autre classe, la seconde prend à sa charge du travail supplémentaire afin que ce ne soit pas l'utilisateur qui ait à le faire.
Voici comment utiliser la première.

```
>>> Facade.say('hello', 'World')
Hello, I'm World
```

Voici comment utiliser la seconde :

```
>>> facade = Facade()
>>> facade.dialog('Plic', 'Ploc', ['Nice factory', 'It works!'])
- Hello, I'm Plic
- Hello, I'm Ploc
- Nice factory
- It works!
- GoodBye, I'm Plic
- GoodBye, I'm Ploc
```

Conséquences

La façade est un des motifs de conceptions les plus utilisés. Beaucoup de modules Python disposent de sous-modules, de classes complexes, de fonctions complexes, toutes préfixées par un caractère souligné pour montrer que ceux-ci sont de la mécanique interne et ne doivent pas être utilisés directement.
Le nom est représentatif de ce que voit celui qui l'utilise, mais il n'a pas besoin d'aller au-delà de la façade à part s'il a besoin d'une fonctionnalité non présente parmi celles mises en façade.

3.6 Poids-mouche

Présentation de la problématique

La problématique est liée au fait que le coût d'une instanciation (création d'une instance à partir d'une classe) est relativement élevé et que très souvent, est pratiquée la politique du tout objet qui veut que l'on crée une instance pour tous les besoins ou presque.
Ce motif de conception vise donc à remplacer le fait de créer des instances par une utilisation de méthodes plus génériques permettant de travailler sur des paramètres plutôt que sur des attributs d'instance.

© Editions ENI - All rights reserved

Solution

Voici une classe simple, petite, dont on a potentiellement beaucoup d'instances :

```
>>> class A:
...     def __init__(self, name):
...         self.name = name
...     def sayHello(self):
...         return 'Hello %s' % self.name
...
```

Voici une autre classe qui est plus générale et dont il n'est pas besoin d'avoir beaucoup d'instances (en l'occurrence, une seule suffit) :

```
>>> class B:
...     def sayHello(self, name):
...         return 'Hello %s' % name
...
```

Dans ce genre de cas spécifique, il est également possible de rendre ce genre de méthode plus cohérente, puisque si celle-ci n'est pas directement en rapport avec l'instance, il faut la rattacher à la classe :

```
>>> class C:
...     @classmethod
...     def sayHello(cls, name):
...         return 'Hello %s' % name
...
```

Et si elle n'est pas en rapport avec la classe non plus, il ne faut pas hésiter à en faire une méthode statique :

```
>>> class D:
...     @staticmethod
...     def sayHello(name):
...         return 'Hello %s' % name
...
```

Le critère déterminant pour savoir ce que l'on fait de la méthode à ce niveau est de déterminer quelle est sa sémantique.

```
>>> a = A('World')
>>> a.sayHello()
'Hello World'
>>> b = B()
>>> b.sayHello('World')
'Hello World'
>>> C.sayHello('World')
'Hello World'
>>> D.sayHello('World')
'Hello World'
```

Conséquences

La sémantique fait que la notion qui était portée par un attribut, donc quelque part par l'instance, est maintenant dissociée, puisque portée par un paramètre.
Il faut donc évaluer la perte de sémantique éventuelle par rapport à l'économie réalisée au niveau des performances. Mais, a contrario, cela peut être explicitement utilisé pour gérer à part une sémantique qui n'est clairement pas rattachée à l'objet.

3.7 Proxy

Présentation de la problématique

Le motif de conception proxy ou délégation est une classe se substituant à une autre en présentant exactement les mêmes caractéristiques externes qu'elle ou seulement une partie et redirigeant ses méthodes vers elle où modifiant le résultat.

Solution

La solution pour créer un proxy identité est :

```
>>> class IdentityProxy:
...     def __init__(self, context):
...         self.context = context
...     def __getattr__(self, name):
...         return getattr(self.context, name)
...
```

Ce proxy peut être utilisé pour projeter un point de l'espace sur le plan horizontal. Voici le point :

```
>>> class Point:
...     def __init__(self, x, y ,z):
...         self._x, self._y, self._z = x, y, z
...     def x(self):
...         return str(self._x)
...     def y(self):
...         return str(self._y)
...     def z(self):
...         return str(self._z)
...
```

Voici la projection (héritant du proxy, le contexte est alors le point de l'espace) :

```
>>> class Projection(IdentityProxy):
...     def z(self):
...         return '0'
...
```

Voici une fonction pour visualiser le résultat sous forme de 3-uplets :

```
>>> def formatter(point):
...     return '(%s)' % ', '.join([point.x(), point.y(), point.z()])
...
```

© Editions ENI - All rights reserved

Et voilà comment construire nos points :

```
>>> point = Point(1, 2, 3)
>>> projection = Projection(point)
>>>
```

Et les afficher :

```
>>> print(formatter(point))
(1, 2, 3)
>>> print(formatter(projection))
(1, 2, 0)
```

Au-delà de ce contexte générique, il est possible de créer un proxy sur mesure pour ne présenter que la ou les méthode(s) que l'on veut offrir, en supprimant les autres. Pour cela, il y a plusieurs moyens, le plus simple étant celui qui suit.
En premier lieu, la classe de base :

```
>>> class A:
...     def m1(self):
...         pass
...     def m2(self):
...         pass
...     def m3(self):
...         pass
...
```

Puis le proxy, qui définit lui-même son contexte et qui opère la redirection de méthodes vers le contexte :

```
>>> class ProxyDeA:
...     def __init__(self):
...         self.context = A()
...     def m1(self):
...         return self.context.m1(self)
...     def m3(self):
...         return self.context.m3(self)
...
```

Voici les différences entre les deux :

```
>>> a1 = A()
>>> 'm1' in dir(a1), 'm2' in dir(a1)
(True, True)
>>> a2 = ProxyDeA()
>>> 'm1' in dir(a2), 'm2' in dir(a2)
(True, False)
```

Voici une classe proxy plus générique :

```
>>> class ProxySelectif:
...     redirected = ['m1', 'm3']
...     def __init__(self, context):
...         self.context = context
```

```
...     def __getattr__(self, name):
...         if name in self.redirected:
...             return getattr(self.context, name)
...
```

Les méthodes redirigées ne sont pas visibles par **dir**, mais bien présentes :

```
>>> a3 = ProxySelectif(A())
>>> 'm1' in dir(a3), 'm2' in dir(a3)
(False, False)
>>> a3.m1, a3.m2
(<bound method A.m1 of <__main__.A object at 0x1eaa090>>, None)
```

Conséquences

Le proxy est très simple à réaliser et il permet de simplifier l'apparence d'un objet. Au contraire, il est possible de concevoir des objets très complexes permettant de gérer plusieurs cas d'utilisation, puis d'en faire un proxy par cas d'utilisation.

4. Motifs de comportement

4.1 Chaîne de responsabilité

Présentation de la problématique

Le motif de conception nommé chaîne de responsabilité permet de créer une chaîne entre différents composants qui traitent une donnée. Ainsi chaque composant reçoit une donnée, la traite s'il le peut, et la transmet au composant suivant dans la chaîne, le tout sans se préoccuper de savoir si le message va intéresser son successeur ou pas.

Solution

Voici un composant autonome qui gère le traitement ou non d'une donnée en fonction de conditions qui lui sont passées à l'initialisation :

```
>>> class Composant:
...     def __init__(self, name, conditions):
...         self.name = name
...         self.conditions = conditions
...         self.next = None
...     def setNext(self, next):
...         self.next = next
...     def traitement(self, condition, message):
...         if condition in self.conditions:
...             print('Traitement du message %s par %s' %
(message, self.name))
...         if self.next is not None:
...             self.next.traitement(condition, message)
...
```

© Editions ENI - All rights reserved

Voici comment créer trois composants :

```
>>> c0 = Composant('c0', [1, 2])
>>> c1 = Composant('c1', [1])
>>> c2 = Composant('c2', [2])
```

Comment créer la chaîne de dépendance :

```
>>> c0.setNext(c1)
>>> c1.setNext(c2)
```

Et le résultat lorsque l'on donne une condition et un message :

```
>>> c0.traitement(1, 'Test 1')
Traitement du message Test 1 par c0
Traitement du message Test 1 par c1
>>> c0.traitement(2, 'Test 2')
Traitement du message Test 2 par c0
Traitement du message Test 2 par c2
```

Conséquences

Cette méthodologie est un moyen simple de créer un découplage entre fonctionnalités séquentiellement exécutées.

4.2 Commande

Présentation de la problématique

Le modèle objet présente des facilités pour produire des objets qui chacun portent les méthodes permettant de se gérer. C'est ce que l'on réalise pour concevoir un modèle de données, par exemple.
Or, le fonctionnement des IHM n'est pas spécifiquement conçu pour utiliser directement les méthodes de ces objets. Une des techniques utilisées est la création de commandes indépendantes qui ont une action sur un objet.
La commande peut éventuellement être complexe et réaliser plusieurs actions, mais l'intérêt est qu'au niveau de l'IHM, on se contente de lancer la commande sans plus de détails.
La commande communique donc à un ou plusieurs objets les actions à effectuer et communique les paramètres nécessaires.
Le traitement de plusieurs commandes peut être dévolu à un objet particulier qui a éventuellement la main sur les commandes et peut les modifier si besoin.
Ceci est un moyen de faire abstraction d'un modèle objet de manière à n'avoir qu'à traiter avec des fonctions ou à la limite assimilables à des fonctions.

Solution

Voici un exemple avec un mobile autonome qui dispose des méthodes pour se gérer lui-même :

```
>>> class Mobile:
...     def deplacerGauche(self):
...         print('Le mobile se déplace à gauche')
...     def deplacerDroite(self):
```

```
...         print('Le mobile se déplace à droite')
...
```

Voici une commande abstraite et les implémentations pour chacune des méthodes :

```
>>> import abc
>>> class Commande(metaclass=abc.ABCMeta):
...     @abc.abstractmethod
...     def executer(self):
...         return
...
>>> class GaucheCommande(Commande):
...     def __init__(self, mobile):
...         self.mobile = mobile
...     def executer(self):
...         self.mobile.deplacerGauche()
...
>>> class DroiteCommande(Commande):
...     def __init__(self, mobile):
...         self.mobile = mobile
...     def executer(self):
...         self.mobile.deplacerDroite()
...
```

Voici maintenant le pilote qui, en fonction des ordres reçus, crée les commandes nécessaires :

```
>>> class Pilote:
...     def __init__(self, cG, cD):
...         self.commandeGauche = cG
...         self.commandeDroite = cD
...     def ordreGauche(self):
...         self.commandeGauche.executer()
...     def ordreDroite(self):
...         self.commandeDroite.executer()
...
```

Rien n'empêche, à ce niveau, d'avoir d'autres méthodes permettant de modifier les commandes, voire de les remplacer.
Voici comment créer le mobile :

```
>>> mobile = Mobile()
```

Comment créer les commandes associées :

```
>>> commande_gauche = GaucheCommande(mobile)
>>> commande_droite = DroiteCommande(mobile)
```

Et le pilote qui reçoit chaque commande en commentaire :

```
>>> pilote = Pilote(commande_gauche, commande_droite)
```

© Editions ENI - All rights reserved

Voici maintenant comment le pilote fonctionne et le résultat :

```
>>> pilote.ordreGauche()
Le mobile se déplace à gauche
>>> pilote.ordreDroite()
Le mobile se déplace à droite
```

Conséquences

Une fois que le modèle objet a été conçu pour obtenir des objets conformes au paradigme, ces commandes sont un moyen de créer un canal direct entre une demande de l'utilisateur, réalisée à travers une IHM, par exemple et une action à réaliser sur ces objets.
En pratique, il est assez peu utilisé, sauf cas particulier.

4.3 Itérateur

Présentation de la problématique

Un itérateur est un motif de conception qui offre un moyen efficace permettant de parcourir un objet présentant un contenu, que celui-ci soit une séquence, un dictionnaire, un arbre ou autre.
Il peut éventuellement être nommé curseur, car il est vu comme un pointeur vers l'élément courant du contenant.
En Python, ce motif de conception est également un type d'objet particulier et suit certaines règles.

Solution

La clé pour réussir un bon itérateur est de parvenir à développer une méthodologie pour trouver l'élément suivant quelle que soit la complexité de la structure de données. Le second point est de le réaliser avec une performance acceptable.
Pour cet exemple, nous reprenons ce qui a été fait pour le motif de conception composite et rajoutons ce qu'il faut pour lui adjoindre un itérateur.
Concevoir un itérateur qui n'ait aucun impact sur l'objet qui lui est associé est idéal, mais ce n'est pas toujours possible. Dans cet exemple, la solution est trop complexe. Voici donc une solution a minima, non élégante, mais qui répond au besoin :

```
>>> class Iterateur:
...     def __init__(self, context):
...         self.context = context.childs()
...     def __iter__(self):
...         return self
...     def __next__(self):
...         if len(self.context) == 0:
...             raise StopIteration
...         return self.context.pop(0)
...
>>> class Composant(metaclass=abc.ABCMeta):
...     def __init__(self, name):
...         self.name = name
```

```
...     @abc.abstractmethod
...     def verbose(self, level=0):
...         return
...     def __iter__(self):
...         return Iterateur(self)
...
>>> class Feuille(Composant):
...     def verbose(self, level=0):
...         return '%sFeuille %s' % ('\t' * level, self.name)
...     def childs(self):
...         return [self]
...
>>> class Composite(Composant):
...     def __init__(self, name):
...         Composant.__init__(self, name)
...         self.contenu = []
...     def add(self, composant):
...         self.contenu.append(composant)
...     def verbose(self, level=0):
...         feuilles = [f.verbose(level+1) for f in self.contenu]
...         feuilles.insert(0, '%sComposite %s' % ('\t' *
level, self.name))
...         return '\n'.join(feuilles)
...     def childs(self):
...         result = [self]
...         for f in self.contenu:
...             result.extend(f.childs())
...         return result
...
```

L'itération a en réalité lieu sur une liste d'enfants, celle-ci étant construite par le composite lui-même. À noter que l'on rajoute aussi la méthode pour relier l'objet conteneur à l'itérateur.

On recrée l'arbre :

```
>>> c1 = Feuille('F1')
>>> c2 = Feuille('F2')
>>> c3 = Composite('C1')
>>> c3.add(Feuille('F4'))
>>> c3.add(Feuille('F5'))
>>> c3.add(Feuille('F6'))
>>> c4 = Composite('C2')
>>> c41 = Composite('C3')
>>> c41.contenu = [Feuille('F7'), Feuille('F8'), Feuille('F9')]
>>> c4.contenu = [Composite('C4'), c41, Feuille('FA')]
>>> main = Composite('Test')
>>> main.contenu.extend([c1, c2, c3, c4])
```

© Editions ENI - All rights reserved

Il ne reste plus qu'à tester l'itérateur :

```
>>> for a in main:
...     print(a.name)
...
Test
F1
F2
C1
F4
F5
F6
C2
C4
C3
F7
F8
F9
FA
```

Conséquences

Les itérateurs sont devenus pour Python (et encore plus dans la branche 3.x) une arme essentielle et sont utilisés abondamment.
Cette astuce sur l'utilisation des méthodes spaciales **`__iter__`** dans le contenu comme dans le conteneur permet d'assurer une syntaxe très minimaliste lors de l'utilisation d'un itérateur, puisqu'on l'utilise naturellement, sans même s'en rendre compte.
Il est bien entendu que les itérateurs sont, pour les types de Python, particulièrement travaillés et très performants.
Tout objet permettant de travailler sur les éléments d'une liste, d'un n-uplet, d'un ensemble, des clés, valeurs ou items d'un dictionnaire utilisent les itérateurs de manière systématique, ceux-ci étant bien entendu aisément transformables en un autre type de donnée (liste, par exemple).

4.4 Mémento

Présentation de la problématique

Il s'agit de permettre à un objet de garder à disposition une représentation d'un état précédent et d'être capable d'y revenir.
Ceci est utilisé par exemple par une fonctionnalité de « annuler » ou « Undo ».

Solution

Le modèle de Python permet de ne pas laisser des objets qui sont utilisés dans un seul contexte dans le flux global. Le mieux est alors de créer la classe Memento directement au moment où l'on s'en sert, ce qui est utile lorsque l'on utilise que peu d'instances de notre objet.

```
>>> class Current:
...     def __init__(self, state):
...         class Memento:
...             state = None
...         self.state = state
...         self.memento = Memento()
...     def setState(self, state):
...         self.memento.state, self.state = self.state, state
...     def resetState(self):
...         state = self.memento.state
...         if state is None:
...             print("Il n'est pas possible d'aller en arrière")
...             self.memento.state, self.state = None, self.memento.state
...
```

Voici comment initialiser notre objet, et l'utiliser afin de voir l'état changer au fur et à mesure des instructions :

```
>>> c = Current('1')
>>> print(c.state)
1
>>> c.setState('2')
>>> print(c.state)
2
>>> c.resetState()
>>> print(c.state)
1
>>> c.setState('3')
>>> print(c.state)
3
>>> c.resetState()
>>> print(c.state)
1
>>> c.resetState()
Il n'est pas possible d'aller en arrière
```

Conséquences

Ce motif n'est utilisé que dans des cas très particuliers.

4.5 Visiteur

Présentation de la problématique

La problématique consiste à détacher la fonctionnalité de ce sur quoi elle s'applique ou dit autrement, l'algorithme du type de données.

Pour cela, on a d'un côté des types de données qui sont capables de se décrire et contiennent les méthodes pour se gérer, et de l'autre, on a des visiteurs qui proposent une fonctionnalité et autant de méthodes que nécessaire pour gérer le même traitement pour chaque type de données.

© Editions ENI - All rights reserved

Solution

Voici un visiteur qui se contente d'effectuer un affichage lorsqu'il est sollicité :

```
>>> class Visiteur1:
...     def visiterCarre(self, carre):
...         print('Visite du carré')
...     def visiterCercle(self, cercle):
...         print('Visite du cercle')
...
```

En voici un autre qui va chercher une donnée dans l'objet qui les contient :

```
>>> class Visiteur2:
...     def visiterCarre(self, carre):
...         print(carre.mesure)
...     def visiterCercle(self, cercle):
...         print(cercle.mesure)
...
```

Voici les types de données utilisés :

```
>>> class Carre:
...     mesure = 'longueur du côté'
...     def accept(self, visiteur):
...         visiteur.visiterCarre(self)
...
>>> class Cercle:
...     mesure = 'rayon'
...     def accept(self, visiteur):
...         visiteur.visiterCercle(self)
...
```

Comme on le voit, la clé c'est que le visiteur passe de paramètre à objet dont on appelle une méthode alors que le visité passe d'objet appelé à paramètre.

```
>>> Carre().accept(Visiteur1())
Visite du carré
>>> Cercle().accept(Visiteur2())
rayon
```

Conséquences

Le visiteur est en Python assez peu utilisé, car il oblige à être trop verbeux et que la nature de l'objet selon Python implique que le décorateur lui est préférable.

4.6 Observateur

Présentation de la problématique

Ce motif de conception est utilisé lorsque des signaux doivent être échangés entre un composant et un autre et qu'il y a une relation de dépendance de l'un à l'autre, c'est-à-dire que c'est un des deux composants qui attend que l'autre le sollicite.

Ainsi, l'observable enregistre la liste de ses observateurs, c'est-à-dire des composants qui sont à l'écoute de lui et va pouvoir subir des modifications. Lorsque celles-ci sont réalisées, elles doivent se terminer par l'appel de la méthode de notification qui seule se charge de demander à tous les observateurs de se mettre à jour.
Elle peut juste demander de se mettre à jour et les observateurs sauront quoi faire pour aller chercher les données, elle peut transmettre directement les données, ou elle peut envoyer un événement qui va les contenir ou donner les informations pour les récupérer.

Solution

Voici une solution minimaliste et fonctionnelle :

```
>>> class Observable:
...     def __init__(self):
...         self.observers = set()
...     def addObserver(self, observer):
...         self.observers.add(observer)
...     def removeObserver(self, observer):
...         self.observers.remove(observer)
...     def notify(self, datas):
...         for o in self.observers:
...             o.update(datas)
...
>>> class Observer:
...     def __init__(self, name):
...         self.name = name
...         self.listeners = []
...     def update(self, datas):
...         print('Mise à jour de %s avec %s' % (self.name, datas))
...
>>> observable = Observable()
>>> observer1 = Observer('Observer 1')
>>> observable.addObserver(observer1)
>>> observer2 = Observer('Observer 2')
>>> observable.addObserver(observer2)
>>>
>>> observable.notify('des données')
Mise à jour de Observer 2 avec des données
Mise à jour de Observer 1 avec des données
```

Conséquences

Le motif est simple sur les principes de base, mais il peut très rapidement se complexifier, à commencer par le fait de rajouter une hiérarchie à ces classes.

© Editions ENI - All rights reserved

4.7 Stratégie

Présentation de la problématique

Le motif de conception stratégie vise à permettre de sélectionner un algorithme à utiliser et de faire en sorte que cet algorithme soit interchangeable, potentiellement à la volée, comme devrait l'être un changement de stratégie pendant une opération.
Le modèle objet particulièrement permissif de Python permet de faire cela très simplement et la problématique principale étant résolue sans efforts, le sujet central porte sur la manière de procéder à ce changement de stratégie, vu les choix multiples.

Solution

Commençons par écrire deux stratégies distinctes :

```
>>> strategy1 = lambda x: x.lower()
>>> strategy2 = lambda x: x.upper()
```

Puis par écrire un composant utilisant le motif de conception :

```
>>> class StrategyManager:
...     def bind(self, func):
...         self.execute = func
...
```

Voici comment ce dernier s'utilise :

```
>>> manager = StrategyManager()
>>> manager.bind(strategy1)
>>> manager.execute('Donnée')
'donnée'
>>> manager.bind(strategy2)
>>> manager.execute('Donnée')
'DONNÉE'
```

On est donc capable d'appliquer différentes fonctionnalités à une donnée tout en gardant un même objet et en utilisant une même fonction.
Idéalement, la fonctionnalité est utilisée dans une boucle ou à la demande et le changement de stratégie s'opère par un événement particulier.

Conséquences

Ce motif de conception bénéficie grandement de la simplicité du modèle objet de Python et du fait qu'un algorithme puisse être écrit très simplement.
On peut combiner ce motif de conception avec celui de la commande pour créer une boucle infinie qui lance une commande à intervalles réguliers et un manager qui change la nature de la commande en fonction de paramètre tierce.
Le mobile bouge donc tout le temps, mais son mouvement est affecté par tous les changements de stratégie.
Ceci peut servir de base pour le développement de jeux vidéos, par exemple.

4.8 Fonction de rappel

Présentation de la problématique

Le principe de fonctionnement de la fonction de rappel est d'être passée en argument d'une autre fonction pour que cette dernière en fasse usage dans certaines conditions.
Derrière ce principe très simple se cachent deux utilisations courantes. La première consiste à se passer soi-même pour que la fonction que l'on utilise nous rende la main d'une manière ou d'une autre. L'autre utilisation consiste à réaliser une première action, laisser la fonction appelée réaliser une seconde action et déclencher une troisième action par l'exécution de la fonction de rappel sans que les actions aient forcément de corrélation entre elles.

Solution

La solution est assez triviale en ce qui concerne la mise en place de la fonction de rappel. À noter qu'il est préféré un paramètre nommé :

```
>>> def callback():
...     print('Fonction de rappel')
...
>>> def do(value, *args, callback):
...     print('Action')
...     if value > 0:
...         callback()
...
```

Ainsi, la fonction de rappel peut être ou ne pas être appelée :

```
>>> do(0, callback=callback)
Action
>>> do(1, callback=callback)
Action
Fonction de rappel
```

La partie complexe n'est pas tant la mise en place de cette fonction que la manière dont les données qu'elles partagent vont être utilisées en son sein.

```
>>> class A:
...     def __init__(self, name):
...         self.name = name
...     def do(self, *args, callback):
...         callback(self.name)
...
>>> class B:
...     def print(self, name):
...         print(name)
...
>>> a = A('Test')
>>> a.do(callback=B().print)
Test
```

© Editions ENI - All rights reserved

Conséquences

Cette fonctionnalité est utilisée pour les IHM et est simple à mettre en œuvre.

5. ZCA

5.1 Rappels

La ZCA est la *Zope Component Architecture* et est un ensemble de bibliothèques indépendantes qui permettent de créer une architecture entre composants.

5.2 Adaptateur

Déclaration

Voici, déclarés conformément aux usages de la ZCA, deux interfaces et deux objets :

```
>>> from zope.interface import Interface
>>> from zope.interface import Attribute
>>> from zope.interface import implements
>>> class Ichien(Interface):
...     nom = Attribute("""Nom du chien""")
...     def aboyer(filename)
...         """Méthode permettant de le faire aboyer"""
...
>>> class Chien(object):
...     implements(IChien)
...     nom = u''
...     def __init__(self, nom):
...         self.nom = nom
...     def aboyer(self):
...         """Méthode permettant de le faire aboyer"""
...         print('Ouaff')
...
>>> class Ichat(Interface):
...     nom = Attribute("""Nom du chat""")
...     def miauler(filename):
...         """Méthode permettant de le faire miauler"""
...
>>> class Chat(object):
...     implements(IChat)
...     nom = u''
...     def __init__(self, nom):
...         self.nom = nom
...     def miauler(self):
...         """Méthode permettant de le faire miauler"""
...         print('Miaou')
...
```

L'idée de cet exemple est d'adapter ces deux objets au sein d'une seule et même classe dont on va commencer par créer une interface. Cette adaptation est simplement réalisée par l'utilisation de la fonction **adapts**.
Voici ce composant :

```
>>> from zope.component import adapts
>>> class Ianimal(Interface):
...     def exprimer(self):
...         """Methode permettant à un animal de s'exprimer"""
...
>>> class Animal(object):
...     implements(IAnimal)
...     adapts(Chien, Chat)
...     def __init__(self, animal):
...         self.animal = animal
...     def exprimer(self):
...         """Methode permettant à un animal de s'exprimer"""
...         if isinstance(self.animal, Chien):
...             self.animal.aboyer()
...         elif isinstance(self.animal, Chat):
...             self.animal.miauler()
...         else:
...             raise Exception("Cet animal ne sait pas s'exprimer")
...
```

Utilisation

À partir de maintenant, lorsque l'on a un chien et un chat, il est possible de les utiliser via l'adaptateur :

```
>>> bambou = Chien('Bambou')
>>> Animal(bambou).exprimer()
Ouaff
```

Le principe de l'adaptation est très utilisé dans la ZCA et c'est un des motifs de conception les plus fréquents. Pour se faire une idée de tous les cas possibles de son utilisation, il est important de multiplier les sources de documentation, l'exemple présenté ici n'ayant qu'une valeur théorique, mais étant dégagé de toute notion liée à son utilisation dans Zope.
La ZCA propose beaucoup d'autres outils utilisant ces adaptateurs dans différents contextes.

© Editions ENI - All rights reserved

5.3 Utilitaire

Déclaration

Le principe de l'utilitaire est assez simple, puisqu'il s'agit d'un composant fournissant une fonctionnalité qui est indépendante. On crée alors une classe qui a les méthodes nécessaires, une seule instance de cette classe et c'est cette instance qui va être utilisée par les composants ayant besoin des services offerts par l'utilitaire.
Là encore, au-delà de cet aspect, ce que la ZCA propose, c'est une méthodologie qui repose sur l'interface. On cherche un utilitaire qui va pouvoir être capable de fournir un service identifié par cette interface. Derrière cela, il y a également beaucoup d'autres fonctionnalités, à commencer par le fait de pouvoir enregistrer les utilitaires et les appeler depuis une autre portion de code.
On déclare donc l'interface, la classe et l'instance :

```
>>> from zope.interface import Interface
>>> from zope.interface import implements
>>> class IidGenerator(Interface):
...     def get(self):
...         """Fournit un id unique"""
...     def getIdFor(self, category):
...         """Fournit un id unique pour chaque catégorie"""
...
>>> class IdGenerator(object):
...     implements(IIdGenerator)
...     def __init__(self):
...         self.id = 0
...         self.ids = {}
...     def get(self):
...         """Fournit un id unique"""
...         self.id += 1
...         return self.id
...     def getIdFor(self, category):
...         """Fournit un id unique pour chaque catégorie"""
...         if category not in self.ids.keys():
...             self.ids[category] = 0
...         self.ids[category] += 1
...         return self.ids[category]
...
>>> id_generator = IdGenerator()
```

Utilisation

Voici comment utiliser cet objet :

```
>>> id_generator.get()
1
>>> id_generator.get()
2
>>> id_generator.getIdFor('moi')
```

```
1
>>> id_generator.getIdFor('moi')
2
>>> id_generator.getIdFor('toi')
1
>>> id_generator.getIdFor('moi')
3
```

La ZCA offre donc des outils supplémentaires pour retrouver cette instance à partir de son interface en demandant quelle instance fournit l'interface.

5.4 Fabrique

Déclaration

On a vu ce qu'était la notion de fabrique et comment mettre en œuvre ce motif de conception simplement avec Python. La ZCA l'utilise abondamment et en conséquence, il a fait l'objet de beaucoup d'attention pour permettre une utilisation facile et très efficace. Une fabrique est un utilitaire qui implémente l'interface IFactory et il est proposé un objet générique Factory.
Voici un exemple simple qui reprend ce que l'on a déjà vu :

```
>>> from zope.interface import Interface
>>> from zope.interface import Attribute
>>> from zope.interface import implements
>>> class Ichien(Interface):
...     nom = Attribute("""Nom du chien""")
...     def aboyer(filename):
...         """Méthode permettant de le faire aboyer"""
...
>>> class Chien(object):
...     implements(IChien)
...     nom = u''
...     def __init__(self, nom):
...         self.nom = nom
...     def aboyer(self):
...         """Méthode permettant de le faire aboyer"""
...         print('Ouaff')
...
```

Voici comment créer un motif de conception fabrique :

```
>>> from zope.component.factory import Factory
>>> factory = Factory(Chien, 'Chien')
```

© Editions ENI - All rights reserved

Utilisation

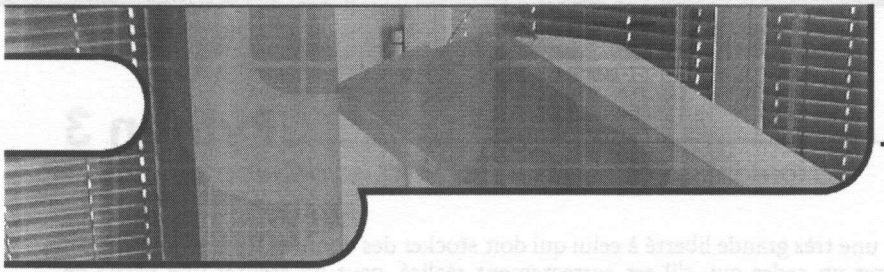
À partir de là, la fabrique est utilisable.

```
>>> bambou = factory('Bambou')
```

L'élément important, c'est la forte intégration de ce motif dans la ZCA qui peut être utilisée conjointement avec une autre fonctionnalité createObject de manière à créer des objets dans des endroits du code séparés de là où la fabrique est créée.

5.5 Pour aller plus loin

La ZCA est un ensemble particulièrement vaste.
Les interfaces, leurs implémentations, les adaptateurs, les utilitaires et les fabriques sont au cœur de cet ensemble permettant la création de composants.
La ZCA est un moyen efficace de traiter les problématiques en remplaçant les solutions qui consistent à utiliser abondamment l'héritage multiple par d'autres proches des motifs de conceptions qui sont des notions transversales aux langages de programmation et donc susceptibles d'être plus partagées par des développeurs venant d'horizons différents.
La ZCA offre également des règles particulières concernant l'organisation de code (quelle classe mettre dans quel fichier, à quel moment créer un répertoire...) et le principal avantage de cette architecture est de permettre de partager les informations d'un module à un autre, de partager les objets.
Ainsi, on crée des composants qui peuvent être configurables et réutilisables.
L'objet central devient alors le registre de composants, également nommé manager du site (notion sémantiquement liée à la destination première de la ZCA).



Chapitre 1.2
XML

1. XML et les technologies qui gravitent autour

1.1 Définition de XML, terminologie associée

XML est le sigle de *eXtensible Markup Language*, soit en français le langage de balisage générique extensible. Il a été conçu pour permettre le stockage d'informations de toute nature dans des fichiers structurés en arborescence via l'utilisation de balises au nom devant être représentatif et devant elles-mêmes suivre des règles inhérentes au format XML et d'autres décrites dans des schémas associés au fichier XML afin de permettre leur validation.

Par nature, un fichier XML contient des balises qui portent un nom et doivent être bien fermées (**</balise></balise>**, **<balise />**) et bien parenthésées. Par exemple, **<a>** est valide, mais pas **<a>**. Chaque balise de son début à sa fin forme ainsi un nœud.

C'est ce principe qui permet de construire l'arborescence et les relations entre nœuds. Par exemple dans **<a><c></c>**, on peut dire que les nœuds **b** et **c** sont frères, qu'elles sont les fils de **a** et que **a** est le père de **b** et de **c**. On peut également dire que **b** est le frère gauche de **c** et que **c** est le frère droit de **b**. On peut par ailleurs dire que **b** et **c** sont des feuilles, car il s'agit de nœuds ne contenant pas d'autres nœuds.

Une balise peut également contenir un ou plusieurs attributs : **<balise attr="valeur" />**.

Au-delà de ces règles communes à tous les fichiers XML et en formant le socle, le format est dit « extensible » parce que le vocabulaire et la grammaire du langage XML ne sont pas figés, mais sont définis pour chaque document XML par une référence à ce schéma qui est un élément central.

Un tel schéma indique quelle balise peut être contenue dans quelle autre et également combien, quels sont les attributs que peuvent utiliser telle ou telle balise et quelle est la nature des données d'un attribut ou d'une feuille.

Ceci donne une très grande liberté à celui qui doit stocker des données tout en lui permettant d'assurer un cadre qui, s'il est correctement réalisé, peut lui assurer une partie de l'intégrité des données. Le schéma peut être vu comme une spécification à laquelle doivent se plier les données et par conséquent les algorithmes qui génèrent et lisent les documents XML associés.

XML est générique car d'une part, il n'est pas dépendant d'un langage ou d'une technologie particulière, même si certaines de ces technologies ont mis XML au cœur de leur manière de travailler et d'autre part il peut être adapté à tout type de données d'une manière plus souple que ce que proposent les bases de données SQL et néanmoins cadrée à l'aide des schémas.

On peut donner deux exemples illustratifs de technologies basées sur XML, à savoir XHTML qui décrit la structure et le contenu d'une page web et SVG qui permet de stocker des images vectorielles.

1.2 Notion de schéma

DSDL est le sigle de *Document Schema Definition Languages* et est une norme qui décrit les différents moyens que l'on peut mettre en œuvre pour valider un document XML ainsi que les impératifs liés au travail de validation.

XML est un langage déclaratif de balisage générique et extensible dont le vocabulaire et la grammaire ne sont pas définis a priori. DSDL apporte simplement cette définition et est une norme (ISO/CEI 19757).

Parmi les schémas les plus connus, il y a DTD (*Document Type Definition*) mais en réalité, ce n'est pas vraiment un schéma, plutôt un modèle de document.

Sa caractéristique est qu'il est spécifié par le W3C et qu'il reste à ce jour le seul moyen officiel reconnu par le W3C de validation d'un document XML. Ce point sera potentiellement amené à changer lorsque les autres méthodes de validation seront plus éprouvées et répandues.

Pour rester simple, un fichier DTD indique pour chaque nœud qui peut exister dans un fichier XML lié les noms des éventuels attributs assortis de leurs caractéristiques ainsi que leurs éventuels contenus (#PCDATA pour du texte ou la liste des nœuds fils assortis de leur cardinalité (?, *, + pour 0 ou 1, 0 à plusieurs ou un à plusieurs)).

L'autre schéma qui est une recommandation du W3C est le « Schéma XML » ou XSD pour *XML Schema Definition*. Il s'agit d'un fichier XML qui est un schéma décrivant d'autres fichiers XML. Bien entendu, lui-même est décrit par un schéma XML. À noter que ce dernier s'autodéfinit. Pour la plupart des besoins, il s'agit, à mon sens, de la façon la plus complète, la plus précise et la plus pertinente de gérer un schéma, bien que ce ne soit qu'une recommandation.

XML Schéma est utilisé pour implémenter le standard Dublin Core de métadonnées, très important en Python puisque le pivot de nombreuses technologies.

Il existe également le Schéma RNG (Relax NG pour *REgular LAnguage for XML Next Generation*). Il s'agit d'un DSDL qui peut revêtir une forme XML ou une forme compacte non-XML.

La spécification de Relax NG est gérée par l'OASIS et elle est utilisée pour spécifier OpenDocument, Atom et le futur DocBook.

© Editions ENI - All rights reserved

Le dernier élément n'est pas vraiment un schéma. Il s'agit de Schematron qui est un DSDL simple et puissant reposant sur l'utilisation d'un nombre réduit d'éléments et Xpath.
En réalité, il est un élément permettant de générer des rapports en décrivant au sein d'un document XML des conditions qui génèrent des phrases. Cela peut être utile pour générer automatiquement un document qui soit une recommandation d'amélioration du fichier XML soumis à validation. On peut considérer que l'émission de zéro recommandation équivaut à une validation réussie mais il est également possible d'afficher des messages pour des conditions exprimant une réussite, auquel cas le fait d'afficher un message n'a pas valeur de validation ratée.
Cet outil est souvent utilisé en tant qu'outil complémentaire aux autres.

1.3 Avantages et inconvénients de XML

Le premier avantage de XML est l'interopérabilité et c'était d'ailleurs l'un des buts de ce langage. XML n'étant pas lié à une technologie, il est utilisable par toutes et du moment que l'on se met d'accord sur le format des données à échanger via l'écriture d'un schéma, il est alors facile de réaliser des algorithmes pour lire ou écrire des fichiers compatibles. Le fichier XML permet entre autres de décrire des objets et est flexible.
Il est donc à la base de certains protocoles comme XML-RPC qui est un protocole RPC (*Remote Procedure Call*) qui effectue des appels de méthodes à travers un réseau, le transport des données s'effectuant via HTTP et le format des données étant en XML et donc hautement structurables tout en restant exploitables quelle que soit la technologie appelante. Aujourd'hui, ce protocole a tendance à être remplacé par SOAP, lui aussi bâti autour de XML.
La technologie Ajax (*Asynchronous Javascript and XML*) est elle aussi utilisatrice de XML , même si ce dernier à tendance à être remplacé par Json (d'un point de vue Pythonique, les deux technologies sont identiques en termes de quantité et de complexité de code à produire).
Les autres avantages sont les possibilités de structuration des données de manière descriptive, la modularité, l'extensibilité et la généricité, en fait, toutes les raisons pour lesquelles XML a été conçu.
Les inconvénients sont la haute verbosité nécessaire pour décrire les données, qui va de pair avec l'éventail de possibilités offertes et le poids du fichier généré pour transporter les données. Il est vrai que si l'on ne garde que les données sans les balises et le formalisme, pour de petites données hautement structurées, il ne reste plus grand-chose.
Pour le premier point, la réponse est qu'il faut savoir utiliser le bon format de données pour les bonnes données, la comparaison entre XML et CSV étant à ce titre révélatrice de cet enjeu puisque les deux formats ne sont pas du tout faits pour les mêmes besoins. D'autre part, vouloir à tout prix résoudre tous les problèmes de la terre avec une et une seule technologie quitte à la tordre dans tous les sens est une hérésie, il faut savoir utiliser la bonne technologie à bon escient.
Pour le second point, il est nécessaire de se poser la question de l'excès d'architecture car très souvent, pour des raisons techniques, on voit des logiciels qui sont réalisés en 7 ou 8 couches alors que Python dispose de tous les outils nécessaires pour éviter cette complexité. Cela permet de se concentrer sur les exigences fonctionnelles et non fonctionnelles qui sont l'essentiel du logiciel plutôt que sur un purisme théorique qui n'apporte rien de concret.

En ce sens, l'écriture d'une application en utilisant plusieurs couches métiers autonomes et cloisonnées n'est pas, pour Python, une bonne pratique.
Par rapport au temps de traitement pour l'analyse ou la génération de fichiers XML, les performances sont bonnes, que ce soit en Python, en C, en C++ ou en Java et ne posent pas vraiment de difficultés. La seule difficulté réelle étant plutôt la formation des développeurs qui doivent utiliser XML et la complexité du traitement des données. En Python, c'est une chance, les choses restent simples.

1.4 Différentes manières de parcourir un fichier XML

Il existe deux principales manières de gérer un fichier XML : SAX et DOM.
SAX est l'acronyme de *Simple API for XML* et a pour principe de parcourir les balises dans l'ordre de lecture et de proposer des hooks au développeur qu'il doit utiliser de manière à réaliser l'action qu'il désire. L'inconvénient d'une telle méthode est que le développeur ne dispose pas de l'ensemble de la structure de données dans laquelle il peut naviguer simplement et à volonté, mais le gros avantage est qu'il est extrêmement léger, puisqu'il ne charge quasiment rien en mémoire, seulement l'élément courant.
DOM est l'acronyme de *Document Object Model* et a pour principe de charger l'intégralité du document en mémoire et de fournir tous les outils nécessaires pour permettre au développeur de naviguer dans l'arborescence à la demande et la modifier autant qu'il le désire sans avoir à suivre de règles spécifiques. Les avantages de DOM sont les inconvénients de SAX et réciproquement.
Python dispose d'une bibliothèque particulière, qui est ElementTree et qui ressemble à DOM sans en être une API, puisqu'elle est conçue pour être pythonique et permettre aux développeurs Python de profiter au maximum des avantages de Python. Il existe par ailleurs BeautifulSoup qui, s'il est prévu plutôt pour (X)HTML, permet de travailler avec des flux XML cassés, ce que ne permettent ni SAX ni DOM qui requièrent un fichier XML bien formé.

1.5 Modules Python dédiés au XML

Il existe de nombreux modules Python dédiés au XML et chacun possède ses propres qualités, cependant le module faisant référence et étant le plus complet est ElementTree. C'est donc sur ce dernier que va se concentrer tout ce chapitre, étant donné les règles pythoniques relatives au ZEN et particulièrement celle qui dit qu'il ne devrait y avoir qu'une seule manière de faire les choses.
Ceci dit, comme Python est bien fait, l'utilisation d'un module différent ne changera rien aux grands principes qui seront exposés ici et ne changera pas grand-chose à la sémantique et donc aux noms de classes ou de méthodes à utiliser.
Contrairement à d'autres technologies, Python intègre les technologies connexes aux siennes pour rendre leur utilisation Pythonique et en cela, la complexité du XML est sérieusement masquée pour le développeur, ce qui est une chose appréciable.
Le module en question est **`xml.etree`** et a été porté sur la branche 3.x de Python.
Les autres modules sont :
- xml.dom (API DOM complète proche de celle de Java) ;
- xml.dom.minidom (API DOM plus légère et pythonique) ;

© Editions ENI - All rights reserved

- xml.dom.pulldom (croisement entre Sax et DOM, permet de ne construire qu'une partie du document, au lieu de devoir tout charger) ;
- xml.sax (API complète SAX2) ;
- xml.parses.expat (API SAX plus légère et pythonique).

2. Valider un document XML

2.1 Document XML

Pour toute cette partie et pour la suite, nous avons besoin de la bibliothèque lxml que l'on peut installer ainsi :

```
$ sudo pip-3.2 install lxml
```

Voici comment importer la brique de base :

```
from lxml import etree
```

Pour valider un document XML, il faut commencer par le charger. Pour ce faire, il y a plusieurs méthodes. La méthode suivante pose un problème lorsqu'est présente la première ligne des fichiers XML contenant un encoding, car Python considère que cette information peut être fausse. L'erreur peut être mise en évidence ainsi :

```
>>> with open('document.xml') as f:
...     f.encoding
...     f.read()
...     f.seek(0)
...     etree.XML(f.read())
...
'UTF-8'
'<?xml version="1.0" encoding="UTF-8"?>\n'
0
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
  File "lxml.etree.pyx", line 2723, in lxml.etree.XML
 (src/lxml/lxml.etree.c:52448)
  File "parser.pxi", line 1564, in lxml.etree._parseMemoryDocument
(src/lxml/lxml.etree.c:79843)
ValueError: Unicode strings with encoding declaration are not supported.
```

Le fichier est déclaré comme étant en UTF-8 dans la toute première ligne du fichier et l'est réellement comme le prouve le premier résultat, mais la présence de cette première ligne crée un problème qui est explicitement décrit dans le message généré par l'exception.
Voici donc comment contourner ce problème :

```
>>> with open('document.xml') as f:
...     if f.read(6) == '<?xml ':
...         f.readline()
...     else:
...         f.seek(0)
...     tree = etree.XML(f.read())
```

```
...
'version="1.0" encoding="UTF-8"?>\n'
>>> tree
<Element liste at 0x2cce8c0>
```

Il existe un autre moyen plus simple et plus court :

```
>>> with open('document.xml') as f:
...     t = etree.parse(f)
...
>>> t.getroot()
<Element liste at 0x2cce870>
```

On a maintenant notre fichier XML qui est bien exploitable et on peut le valider.

2.2 Schéma DTD

En prérequis, il est considéré que le lecteur sait ce qu'est un DTD. Celui qui est utilisé est disponible en téléchargement sur le site des Éditions ENI.
Il traite deux types de données différents (entier et chaîne de caractères), un nœud et une feuille, et deux attributs, dont un obligatoire et un facultatif afin de traiter de manière minimaliste des cas différents.
On a besoin pour cette partie là de StringIO :

```
>>> from io import StringIO
```

Il faut dans un premier temps charger le fichier DTD :

```
>>> with open('document.dtd', 'r') as f:
...     dtd = etree.DTD(StringIO(f.read()))
...
```

Puis dans un second temps réaliser la validation avec l'un ou l'autre arbre :

```
>>> dtd.validate(root)
True
>>> dtd.validate(t.getroot())
True
```

Si une erreur se produit, il est possible d'obtenir de plus amples informations que le simple False que renvoie la méthode précédente :

```
>>> print(dtd.error_log.filter_from_errors())
```

2.3 Schéma XSD

En prérequis, il est considéré que le lecteur sait ce qu'est un XSD. Celui qui est utilisé est disponible en téléchargement sur le site des Édtions ENI.
Comme pour le cas précédent, il traite deux types de données différentes (entier et chaîne de caractères), un nœud et une feuille, et deux attributs dont un obligatoire et un facultatif afin de traiter de manière minimaliste des cas différents.

© Editions ENI - All rights reserved

Pour valider selon un schéma XSD, la procédure est relativement similaire, au détail près qu'un fichier XSD est également un fichier XML. Il faut donc parser le fichier XML pour ensuite donner le résultat (racine exploitable) au constructeur XMLSchema :

```
>>> with open('document.xsd', 'r') as f:
...     xsd = etree.XMLSchema(etree.parse(f))
...
```

Il ne reste plus qu'à valider :

```
>>> xsd.validate(tree)
True
>>> xsd.validate(t.getroot())
True
```

Pour voir les éventuelles erreurs, on garde le même principe :

```
>>> print(schema.error_log.filter_from_errors())
```

2.4 Schéma RNG (RelaxNG)

En prérequis, il est considéré que le lecteur sait ce qu'est un RNG (http://relaxng.org/tutorial-20011203.html). Celui qui est utilisé est disponible en téléchargement sur le site des Éditions ENI.

Comme pour le cas précédent, il traite deux types de données différentes (entier et chaîne de caractères), un nœud et une feuille, et deux attributs, dont un obligatoire et un facultatif afin de traiter de manière minimaliste des cas différents.

Encore une fois, la validation suivant RNG est similaire à ce que l'on a déjà vu :

```
>>> with open('document.rng', 'r') as f:
...     rng = etree.RelaxNG(etree.parse(f))
...
>>> rng.validate(tree)
True
>>> rng.validate(t.getroot())
True
```

Et pour les éventuelles erreurs :

```
>>> rng.error_log.filter_from_errors()
```

Mais il existe également une version compacte de ces schémas, mais elle n'est pas encore nativement supportée par lxml, ni de façon éprouvée par d'autres bibliothèques Python 3.x. Il faut donc la convertir en un document RNG pour pouvoir l'utiliser à l'aide d'outils externes.

2.5 Schematron

En prérequis, il est considéré que le lecteur sait ce qu'est un Schematron (http://www.schematron.com/). Celui qui est utilisé est disponible en téléchargement sur le site des Éditions ENI.

La spécificité de ce format est décrite dans la section précédente, mais la méthode employée est similaire à ce que l'on a vu précédemment :

```
>>> with open('document.sch', 'r') as f:
...     sch = etree.Schematron(etree.parse(StringIO(f.read())))
...
>>> sch.validate(tree)
True
>>> sch.validate(t.getroot())
True
```

Et pour lire les éventuelles erreurs :

```
>>> sch.error_log.filter_from_errors()
```

Ce schéma est très utile pour une utilisation sur du XHTML, ou du HTML valide au niveau XML (pour que le parsing réussisse afin que le flux puisse être validé). Ainsi, il est possible d'avoir des rapports donnant des conseils du type « utilisez des feuilles de styles externes plutôt que des styles embarqués » lorsqu'il est détecté la présence d'une balise **<style>** dans l'en-tête.

3. DOM

3.1 Lecture

On a vu la manière de parser un fichier pour le valider. En réalité, il s'agit d'un arbre DOM que l'on valide. Cet arbre est extrêmement important pour beaucoup de technologies, comme par exemple la technologie JavaScript qui utilise l'arbre DOM du document HTML comme base de travail (on peut le voir grâce à firebug). Le code le plus court est rappelé ici :

```
>>> with open('document.xml') as f:
...     t = etree.parse(f)
...
>>> root = t.getroot()
```

Ainsi, il est possible de réaliser à peu près toutes les opérations que l'on souhaite. Voici un pot-pourri :

```
>>> root.getchildren()
[<Element personne at 0x2cce910>, <Element personne at 0x2d750f0>,
<Element personne at 0x2dc5320>]
>>> for c in root.iterchildren():
...     c.getchildren(), c.text, c.values(), c.items()
...
([], None, ['1', 'Paul Personne'])
([], None, ['2', 'Joe Satriani'])
([], None, ['3'])
>>> root.getchildren()[0].keys()
['id', 'nom']
>>> root.getchildren()[0].items()
[('id', '1'), ('nom', 'Paul Personne')]
```

© Editions ENI - All rights reserved

```
>>> root.getchildren()[0].get('id')
'1'
```

Voici un exemple d'extraction de données :

```
>>> d = {n.get('id'): n.get('nom') for n in root.getchildren()}
>>> d
{'1': 'Paul Personne', '3': None, '2': 'Joe Satriani'}
```

Comme cela a été signalé, il existe xml.dom.minidom, qui est une implémentation allégée de DOM, plus proche de ce que l'on rencontre dans les autres langages de programmation et qui est donc potentiellement plus facile à prendre en main pour ceux venant du monde de Java ou de PHP, par exemple, alors que lxml est idéale pour ceux qui recherchent efficacité et performance.

```
>>> from xml.dom.minidom import parse, Node
>>> xml = parse('document.xml')
>>> d = [dict(n.attributes.items()) for n in
xml.documentElement.childNodes if n.nodeType == Node.ELEMENT_NODE]
>>> d
[{'nom': 'Paul Personne', 'id': '1'}, {'nom': 'Joe Satriani',
'id': '2'}, {'id': '3'}]
```

La bibliothèque lxml est plus performante, plus pythonique et beaucoup moins verbeuse, une notion importante lorsque l'on touche à XML. Cependant, le choix entre ces librairies, dans les cas où la performance n'est pas le critère principal, est uniquement fonction de l'expérience et des habitudes du développeur.

3.2 Écriture

Écrire un document XML nécessite une bonne vision de l'architecture utilisée pour représenter les données. Il est vraiment recommandé de se baser sur un schéma afin de toujours terminer l'écriture d'un document par une validation et ainsi s'assurer d'avoir réalisé un travail conforme.
Pour le reste, au plan technique, l'écriture d'un document est relativement simple :

```
>>> root = etree.Element('liste')
>>> p = etree.Element('personne', id='4', nom='Brian May')
>>> root.append(p)
```

On a construit deux éléments, dont un avec des attributs et on a créé une hiérarchie entre eux. Vérifions ce que l'on a effectué par des mécanismes divers :

```
>>> root.tag
'liste'
>>> root.getchildren()
[<Element personne at 0x2ccec80>]
>>> root.getchildren()[0].getroottree().getroot() is root
True
```

Un point très important est ce que l'on appelle la copie profonde, c'est-à-dire la copie d'un nœud avec tous ses fils qui se réalise comme suit :

```
>>> from copy import deepcopy
>>> root.append(deepcopy(root[0]))
```

Il est également possible de supprimer un nœud :

```
>>> root.remove(root[1])
```

Et d'écrire le résultat :

```
>>> print(etree.tostring(root, pretty_print=True))
b'<liste>\n  <personne nom="Brian May" id="4"/>\n</liste>\n'
```

Il n'y a plus (après l'avoir validé) qu'à ouvrir un fichier en écriture pour y insérer cette séquence d'octets et on a un nouveau fichier XML.
Il existe également un autre module permettant de réaliser ce genre d'opération. Les manipulations avec l'un ou l'autre se ressemblent avec les mêmes restrictions que celles décrites pour la lecture. Voici un algorithme réalisant la même opération :

```
>>> from xml.dom.minidom import Document
>>> document = Document()
>>> root = document.createElement("liste")
>>> child = document.createElement('personne')
>>> child.setAttribute('id', '4')
>>> child.setAttribute('nom', 'Brian May')
>>> root.appendChild(child)
>>> document.appendChild(root)
>>> print(document.toprettyxml())
<?xml version="1.0" ?>
<liste>
    <personne id="4" nom="Brian May"/>
</liste>
```

4. SAX

4.1 Support de SAX dans lxml

La bibliothèque lxml est, avant toute chose, faite pour parser des fichiers XML et construire des arbres DOM. Cependant, elle dispose des outils nécessaires pour appliquer les principes de SAX non pas directement sur le fichier XML, mais sur l'arbre DOM.
Du coup, on perd l'avantage de SAX qui est de n'utiliser que peu de mémoire, puisqu'il ne charge que la balise courante, mais on profite de la méthodologie SAX, le tout étant vu comme de la programmation événementielle.
Le principe est de définir un handler qui offre des hooks permettant au développeur de déclencher une action lorsqu'un certain événement se produit. Cet événement peut être le début ou la fin du document, l'entrée ou la sortie d'une balise, etc. La difficulté pour le développeur est de gérer correctement l'événement en s'assurant qu'il se trouve bien là où il pense être (surtout lorsqu'une balise de même nom peut se trouver à plusieurs niveaux de profondeur), et à gérer correctement le fait que l'arborescence est lue à plat et non pas en profondeur.

© Editions ENI - All rights reserved

Ce handler s'appuie sur l'API complète de SAX et se définit ainsi :

```
>>> from xml.sax.handler import ContentHandler
>>> class MyHandler(ContentHandler):
...     def __init__(self):
...         self.nb = 0
...         self.personnes = []
...     def startElementNS(self, name, qname, attributes):
...         print('Noeud trouvé: %s' % qname)
...         if qname == 'personne':
...             self.nb += 1
...
self.personnes.append(dict(attributes.items()))
...     def characters(self, data):
...         pass
...
>>> handler = MyHandler()
```

Il faut utiliser une instance distincte pour chaque document XML parsé.
Pour plus d'information sur les autres hooks, il faut exécuter :

```
>>> help(ContentHandler)
```

Et voici comment parser effectivement le fichier et récupérer les données :

```
>>> from lxml.sax import saxify
>>> saxify(tree, handler)
Noeud trouvé: liste
Noeud trouvé: personne
Noeud trouvé: personne
Noeud trouvé: personne
>>> handler.nb
3
>>> handler.personnes
[{(None, 'nom'): 'Paul Personne', (None, 'id'): '1'}, {(None,
'nom'): 'Joe Satriani', (None, 'id'): '2'}, {(None, 'id'): '3'}]
```

4.2 API SAX Allégée

La bibliothèque xml.sax est une API complète de SAX et est très proche en termes de syntaxe et de façon de faire de ce que l'on peut trouver en Java ou en PHP, donc idéale pour les développeurs venant de cet horizon et qui aiment retrouver les similarités avec ce qu'ils connaissent déjà. Pour les autres besoins, la bibliothèque expat est une API plus légère qui tire parti du modèle objet de Python pour permettre la construction plus modulaire et moins monolithique d'un parser.
Parser un fichier s'opère en quatre lignes, import compris :

```
>>> from xml.parsers.expat import ParserCreate
>>> parser = ParserCreate()
>>> with open('document.xml', 'rb') as f:
...     parser.ParseFile(f)
...
1
```

Le parser ne produit rien, mais il a parsé le fichier, ce qui signifie déjà qu'il n'est pas mal formé. Il faut maintenant lui donner les bonnes méthodes pour les bons hooks, mais de manière originale.
La première chose à faire est de concevoir un objet destiné à recevoir les données que l'on compte extraire du document XML.

```
>>> class Donnees(object):
...     nb = 0
...     personnes = []
...
>>> donnees = Donnees()
```

Maintenant, il faut définir le ou les hooks à utiliser pour récupérer les données, à l'aide de simples fonctions :

```
>>> def start_element(name, attributes):
...     if name == 'personne':
...         donnees.nb += 1
...         donnees.personnes.append(dict(attributes.items()))
...
```

Maintenant, il faut créer le parser et lui adjoindre la fonction :

```
>>> parser = ParserCreate()
>>> parser.StartElementHandler = start_element
```

Il ne reste plus qu'à parser le fichier et afficher les résultats :

```
>>> with open('document.xml', 'rb') as f:
...     parser.ParseFile(f)
...
1
>>> donnees.nb
3
>>> donnees.personnes
[{'nom': 'Paul Personne', 'id': '1'}, {'nom': 'Joe Satriani',
'id': '2'}, {'id': '3'}]
```

L'avantage de cette méthode – et il est de taille – est de permettre la séparation entre le handler et les données elles-mêmes.

5. XPath

XPath est un langage permettant une interrogation simplifiée d'un document XML.
Toute la puissance de XPath se révèle lorsque le développeur maîtrise bien la formulation de ses interrogations et recense tous les cas possibles décrits par son expression.
Pour le reste, la partie Python est triviale. Pour commencer, Python permet à l'objet document XML issu de **`etree.parse`** de donner le chemin d'un de ses nœuds :

```
>>> xml.getpath(xml.getroot().getchildren()[1])
'/liste/personne[2]'
```

© Editions ENI - All rights reserved

Voici un court rappel de la manière dont la variable xml a été générée :

```
>>> with open('document.xml') as f:
...     f.readline()# suppression de la première ligne
...     xml = etree.parse(StringIO(f.read()))
...
```

Voici un exemple basique d'interrogation et d'utilisation de la réponse :

```
>>> element = xml.xpath('/liste/personne')
>>> len(element)
3
>>> element[0].tag
'personne'
```

Un exemple des fonctions utilisables dans les expressions XPath :

```
>>> xml.xpath('count(/*/personne)')
3.0
```

Un exemple d'itération à travers des nœuds (qui commencent à 1, et non à 0) :

```
>>> for i in range(5):
...     expr = '/*/personne[%s]' % i
...     print('%s: %s' % (expr, xml.xpath(expr)))
...
/*/personne[0]: []
/*/personne[1]: [<Element personne at 0x1240050>]
/*/personne[2]: [<Element personne at 0x12400a0>]
/*/personne[3]: [<Element personne at 0x12400f0>]
/*/personne[4]: []
```

Des exemples travaillant sur la présence ou la valeur d'un attribut :

```
>>> xml.xpath('/*/personne[@id]')
[<Element personne at 0x1240050>, <Element personne at 0x12400a0>,
<Element personne at 0x12400f0>]
>>> xml.xpath('/*/personne[@nom]')
[<Element personne at 0x1240050>, <Element personne at 0x12400a0>]
>>> xml.xpath('/*/personne[@id=2]')
[<Element personne at 0x12400a0>]
```

Et pour terminer, un exemple de variabilisation d'un paramètre de la recherche :

```
>>> xml.xpath('/*/personne[@id=$id]', id=2)
[<Element personne at 0x12400a0>]
```

Voici maintenant un exemple plus concret et plus complet.
Un document OpenDocument est un fichier zip contenant une structure de fichiers que nous présenterons ultérieurement, dans le chapitre Génération de contenu.
Les fichiers contenus sont des fichiers XML et parmi eux, le fichier **`content.xml`** contenant la structure du document.

Il est donc possible, et facile, d'accéder au contenu d'un document OpenDocument par de simples scripts Python. Pour cela, il faut procéder en deux étapes :
D'abord récupérer le contenu du fichier qui nous intéresse :

```
>>> with zipfile.ZipFile('technologies.odt') as f:
...     content = f.read('content.xml')
...
```

Ensuite, créer l'arbre DOM de ce fichier :

```
>>> tree = etree.fromstring(content)
```

À partir de là, pour utiliser XPath, il faut avoir la liste des namespaces dont on a besoin (les namespaces servant de préfixe aux noms de balises). La terminologie utilisée est relativement longue, mais n'est pas complexe pour autant côté Python.

```
>>> namespaces = {
...     "table": "urn:oasis:names:tc:opendocument:xmlns:table:1.0",
...     "fo": "urn:oasis:names:tc:opendocument:xmlns:xsl-fo-compatible:1.0",
...     "manifest": "urn:oasis:names:tc:opendocument:xmlns:manifest:1.0",
...     "presentation":
 "urn:oasis:names:tc:opendocument:xmlns:presentation:1.0",
...     "meta": "urn:oasis:names:tc:opendocument:xmlns:meta:1.0",
...     "style": "urn:oasis:names:tc:opendocument:xmlns:style:1.0",
...     "draw": "urn:oasis:names:tc:opendocument:xmlns:drawing:1.0",
...     "text": "urn:oasis:names:tc:opendocument:xmlns:text:1.0",
...     "office": "urn:oasis:names:tc:opendocument:xmlns:office:1.0",
... }
```

Maintenant, il est facile de récupérer, par exemple, les titres du document :

```
>>> titres =
[(n.get('{urn:oasis:names:tc:opendocument:xmlns:text:1.0}outline-level'),
n.text) for n in tree.xpath('//text:h', namespaces=namespaces)]
>>> titres
[('1', 'Linux'), ('2', 'Debian'), ('2', 'Ubuntu'), ('1', 'Serveurs'),
('2', 'Apache'), ('2', 'OpenLDAP'), ('1', 'Langages'), ('2', 'Python'),
('2', 'XML')]
```

En clair, le plus gros travail est d'avoir sous la main les schémas correspondant aux fichiers XML sur lesquels on travaille ou à défaut une bonne documentation afin de savoir comment aborder un document XML.

6. XSLT

XSLT est l'acronyme de *eXtensible Stylesheet Language Transformations* et, comme son nom le laisse supposer, il s'agit d'un langage XML qui permet d'opérer des transformations de style afin de traduire un document XML respectant un certain schéma en un autre document XML respectant un autre schéma.
Le document XML d'entrée ou de sortie peut être des utilisations particulières de langages, comme (X)HTML ou SVG, mais dans tous les cas ces documents doivent être bien formés, sans quoi la transformation est impossible.

© Editions ENI - All rights reserved

Bien que cela ne soit pas prévu par la recommandation XSLT, il est envisageable que le format de sortie ne soit pas un format XML, ce qui permet de mettre en place des mécanismes de transformation vers un fichier texte, par exemple.
Pour commencer, il faut charger le document XSLT qui est du XML :

```
>>> with open('document.xslt', 'r') as f:
...     xslt = etree.XSLT(etree.parse(f))
...
```

Puis, il suffit de se servir du résultat pour transformer notre document XML :

```
>>> xml2 = xslt(xml, **{'date': '20110901'})
```

Le résultat de l'opération peut être affiché en console :

```
>>> etree.tostring(xml2, pretty_print=True)
b'<root name="liste de personnes" date="20110901">\n  <liste>\n
<personne>\n      <nom>Paul Personne</nom>\n    </personne>\n
<personne>\n      <nom>Joe Satriani</nom>\n    </personne>\n
<personne>\n      <nom/>\n    </personne>\n  </liste>\n</root>\n'
```

ou écrit dans un fichier à l'aide du code suivant :

```
>>> with open('transforme.xml', 'wb') as f:
...     f.write(etree.tostring(xml2, pretty_print=True))
...
245
```

Ce qui donne le résultat suivant :

```
<root name="liste de personnes" date="20110901">
  <liste>
    <personne>
      <nom>Paul Personne</nom>
    </personne>
    <personne>
      <nom>Joe Satriani</nom>
    </personne>
    <personne>
      <nom/>
    </personne>
  </liste>
</root>
```

Là encore, il faut se reporter aux fichiers joints à cet ouvrage, les lire, les tester afin de bien prendre en main les outils Python, mais le principal effort à fournir est de maîtriser les concepts de XSLT, de préférence en utilisant les syntaxes courtes, plutôt que celles plus verbeuses.

7. Cas spécifique des fichiers HTML

7.1 Problématique

L'histoire de l'Internet, du web et de l'évolution du langage HTML est pavée de rebondissements et même s'il est maintenant normé (HTML 4, XHTML, XML5), il est rare que cette norme soit réellement suivie.

Pour pallier cette difficulté et pour permettre un traitement professionnel et efficace des flux XML, le langage XHTML a été conçu, rapprochant le HTML du XML et permettant d'en faire un fichier respectant parfaitement les normes XML et donc pouvant être parsé sans difficultés, se débarrassant au passage de résidus indésirables.

Il reste que les fichiers HTML mal formés ne peuvent être parsés à l'aide des parsers XML classiques. **`BeautifulSoup`** et le module **`html`** sont deux réponses.

Il est à noter que le nouveau standard HTML5 est une nouvelle norme supprimant la plupart des résidus de HTML4, mais en gardant quelques-uns, rajoutant de nouvelles balises destinées à améliorer la sémantique (web sémantique) et surtout, définissant des heuristiques permettant d'interpréter de manière déterministe le flux de données quelles que soient ses malformations.

7.2 Parser un fichier HTML à la façon DOM

La bibliothèque BeautifulSoup est portée sous Python 3, voici comment l'installer pour la branche 2.x :

```
$ sudo easy_install3 BeautifulSoup4
```

Parser un fichier HTML reste trivial :

```
>>> from bs4 import BeautifulSoup
>>> with open('document.html') as f:
...     soup = BeautifulSoup(f.read())
...
```

Et en plus, la représentation de l'objet est le code parsé lisible et corrigé dans la mesure des moyens des heuristiques de BeautifulSoup :

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//FR"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="EN" lang="fr">
<head>
<title>Page XHTML d'exemple</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
</head>
<body>
<p><b>Ceci est un document mal formé</b></p>
<p id="verdict">Il y a deux erreurs, une dans chaque ligne du
corps. </p></body>
</html>
```

© Editions ENI - All rights reserved

La correction sur les balises dont la fermeture est dans le mauvais ordre est réalisée, mais celle où la fin de la balise est oubliée est corrigée en positionnant la fin de balise fermante manquante juste avant la balise fermante du père.
La correction peut alors ne pas correspondre à la volonté initiale du développeur, en particulier si l'élément suivant l'emplacement réel de la balise fermante est un frère :

Code voulu :

```
<a>
  <b></b>
  <c></c>
</a>
```

Code écrit :

```
<a>
  <b>
  <c></c>
</a>
```

Code corrigé :

```
<a>
  <b>
  <c></c>
</b></a>
```

Il n'y a donc pas de solution miracle et l'importance de réaliser un code HTML propre à la source est réelle. Cependant, ce parseur permet au moins d'avoir un résultat.
BeautifulSoup4 permet par la suite de rendre ce code mieux parenthésé et donc plus facilement lisible :

```
>>> for line in soup.prettify().splitlines():
...     print(line)
...
```

Voici le résultat :

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//FR"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="EN"
lang="fr">
 <head>
  <title>
   Page XHTML d'exemple
   <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  </title>
 </head>
 <body>
  <p>
   <b>
    Ceci est un document mal formé
   </b>
  </p>
  <p id="verdict">
   Il y a deux erreurs, une dans chaque ligne du corps.
  </p>
 </body>
</html>
```

En quatre lignes (import compris), on a donc un objet qui lit un fichier HTML en le corrigeant et construit un arbre DOM qui est exploitable à la manière du DOM ou ainsi :

```
>>> soup.html.body.p.text
u'Ceci est un document mal form\xe9'
>>> soup.html.body.p.findNextSibling().attrs
[(u'id', u'verdict')]
```

7.3 Parser un fichier HTML à la façon SAX

Le module html.parser contient un parseur HTML permettant de gérer les flux HTML en gérant l'exigence de validité, mais les critères d'invalidité sont faibles pour des raisons de compatibilité avec les anciennes versions.
Le fichier document.html fourni dans le code source livré avec cet ouvrage est ainsi considéré comme invalide, mais ne lève pas d'exceptions. Pour en avoir une, il faut produire des erreurs de nature différentes, telles que celle-ci:

```
>>> with open('document.html') as f:
...     parser = HTMLParser()
...     parser.feed(f.read())
...
Traceback (most recent call last):[...]
html.parser.HTMLParseError: bad end tag: '</body\n</html>'[...]
```

À la fin de la balise body, il manque le signe supérieur. Il est possible de faire fonctionner un tel code par le passage d'un paramètre nommé **strict** :

```
>>> with open('document.html') as f:
...     parser = HTMLParser(False)
...     parser.feed(f.read())
...
```

Pour que le parseur fasse quelque chose, il faut remplir des hooks :
- handle_starttag(tag, attrs) : hook utilisé lorsqu'une balise est ouverte ;
- handle_startendtag(tag, attrs) : hook utilisé lorsqu'une balise est fermée ;
- handle_data(data) : hook utilisé lorsqu'un contenu au sens XML est détecté ;
- handle_entityref(name) : hook utilisé lorsqu'une entité (sens XML) est détectée ;
- handle_comment(data) : hook utilisé lorsqu'un commentaire HTML est détecté ;
- handle_decl(decl) : hook utilisé pour une déclaration **DOCTYPE**.

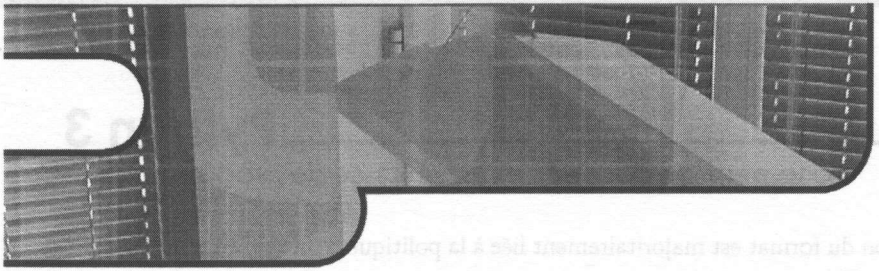
Voici un court exemple récupérant le titre du document :

```
>>> class TitleHTMLParser(HTMLParser):
...     capture = False
...     def handle_starttag(self, tag, attrs):
...         if tag == 'title':
...             self.capture = True
...     def handle_endtag(self, tag):
...         self.capture = False
...     def handle_data(self, data):
...         if self.capture == True:
...             print(data)
```

© Editions ENI - All rights reserved

```
...
>>> with open('document.html') as f:
...     parser = TitleHTMLParser()
...     parser.feed(f.read())
...
Page XHTML d'exemple
```

Le module **html** contient par ailleurs une fonction utile **escape** permettant d'échapper les séquences HTML d'un texte pour produire un contenu.



Chapitre 1.3
Génération de contenu

1. PDF

1.1 Présentation

1.1.1 Format PDF

PDF est le sigle de *Portable Document Format* qui est un format de document utilisant un langage de description de page qui est PDL (sigle de *Page Document Language*) et un protocole d'impression non dépendant d'un constructeur qui est une évolution de Postscript. Il est devenu progressivement le standard pour l'impression des documents et est devenu une norme ISO.

Aujourd'hui, de très nombreux formats de données (textes, dessins, images...) proposent des fonctionnalités d'export vers PDF.

1.1.2 Avantages

Ses qualités principales sont :

- la concordance entre l'affichage à l'écran et ce qui sera réellement imprimé ;
- son indépendance par rapport au système d'exploitation ;
- son indépendance par rapport au matériel ;
- la présence de logiciels ou bibliothèques libres pour ce format ;
- la très grande diffusion de lecteurs de documents PDF, dont certains sont libres.

1.1.3 Inconvénients

Les inconvénients sont de plusieurs ordres :

- Les droits liés à chaque document dépendent à la fois de ceux liés au format, mais également à ceux liés à tout ce qui est contenu dans le document, à savoir les droits d'auteurs sur les textes, les images embarquées, mais également les polices embarquées.

– L'évolution du format est majoritairement liée à la politique d'un seul éditeur.

1.1.4 Présentation de la bibliothèque libre

ReportLab est une bibliothèque externe écrite en Python qui offre des outils simples et performants pour générer des documents PDF. Cette bibliothèque est maintenue par un éditeur qui propose deux branches, dont une à destination de la communauté qui est celle que nous allons utiliser.
Elle est portée sous Python 3, plus précisément à partir de Python 3.3, et voici le dépôt de code : https://github.com/nakagami/reportlab

1.2 Bas niveau

1.2.1 Bibliothèque de données

La première chose pour créer un document est d'avoir des données préfabriquées. En effet, il est agréable d'avoir des formats tout prêts correspondant à du A4, A3 ou tout autre format usuel, des couleurs prédéfinies prêtes à l'emploi et un système permettant de convertir les mesures que l'on connaît avec celles utilisées par ReportLab, à savoir en **point** car c'est l'unité de mesure des écrans et des imprimantes.
Voici ce que l'on peut trouver en fouillant un peu dans cette bibliothèque de données :

```
>>> from reportlab import lib
>>> dir(lib)
['RL_DEBUG', '__builtins__', '__doc__', '__file__', '__name__',
'__package__', '__path__', '__version__', 'boxstuff', 'colors',
'logger', 'os', 'pagesizes', 'rltempfile', 'units', 'utils']
```

Voici pour commencer le module dédié aux formats standards des imprimantes courantes :

```
>>> from reportlab.lib import pagesizes
>>> dir(pagesizes)
['A0', 'A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'B0', 'B1', 'B2', 'B3',
'B4', 'B5', 'B6', 'ELEVENSEVENTEEN', 'LEGAL', 'LETTER', '_BH',
'_BW', '_H', '_W', '__builtins__', '__doc__', '__file__',
'__name__', '__package__', '__version__', 'cm', 'elevenSeventeen',
'inch', 'landscape', 'legal', 'letter', 'portrait']
```

Voici quelques-unes de ces unités :

```
>>> units.inch
72.0
>>> units.cm
28.346456692913385
```

Il y a donc 72 points par pouce et 28 par centimètre. On retrouve facilement le fait qu'un pouce fait 2,54 centimètres. On préférera parler dans une unité qui est homogène à celle du système international, mais la bibliothèque elle-même utilise les unités anglaises :

```
>>> units.inch / units.cm
2.54
```

© Editions ENI - All rights reserved

On peut donc ainsi convertir les mesures de pages en centimètres :

```
>>> [s / cm for s in pagesizes.A4]
[21.0, 29.7]
```

Il y a un dernier module particulièrement important, permettant de ne pas se compliquer la vie pour créer des couleurs, en disposant des plus courantes d'entre elles :

```
>>> from reportlab.lib import colors
>>> dir(colors)
```

Le résultat de cette commande est relativement long, signe du nombre important de couleurs. Il y a fort à parier que LA couleur que vous voulez en fasse partie.
Voici comment se présentent ces couleurs :

```
>>> colors.yellow
Color(1,1,0,1)
>>> colors.yellowgreen
Color(.603922,.803922,.196078,1)
>>> colors.turquoise
Color(.25098,.878431,.815686,1)
>>> colors.olive
Color(.501961,.501961,0,1)
```

Il s'agit en réalité d'une sorte de 4-uplet qui ne présente pas une couleur au format CMJN (Cyan, Magenta, Jaune, Noir), contrairement à ce que l'on pourrait penser de prime abord, étant donné que c'est le format pour les imprimantes, mais un format RVB.
Il est possible d'obtenir des informations sur une couleur selon beaucoup de méthodes différentes :

```
>>> c = colors.turquoise
>>> c.red, c.green, c.blue, c.alpha
(0.25098039215686274, 0.8784313725490196, 0.8156862745098039, 1)
>>> c.hexval(), c.hexvala()
('0x40e0d0', '0x40e0d0ff')
>>> c.rgb(), c.rgba()
((0.25098039215686274, 0.8784313725490196, 0.8156862745098039),
(0.25098039215686274, 0.8784313725490196, 0.8156862745098039, 1))
>>> c.bitmap_rgb(), c.bitmap_rgba()
((64, 224, 208), (64, 224, 208, 255))
```

Et si aucune couleur ne convenait, il est toujours possible d'en créer. Voici par exemple un « Azur clair », dont les valeurs proviennent de Wikipédia (http://fr.wikipedia.org/wiki/Liste_de_couleurs) :

```
>>> c = colors
>>> c = colors.Color(116./255, 208./255, 241./255, 1)
>>> c
Color(.454902,.815686,.945098,1)
```

De la même manière, il est possible de se fabriquer un format d'impression, puisque ce n'est finalement rien d'autre qu'un 2-uplet, mais cela est plus rare, ReportLab ayant tout ce qu'il faut à ce niveau-là.

Un dernier point extrêmement important est la gestion des polices. Par convention, 14 d'entre elles sont standardisées et disponibles donc sans efforts particuliers, il suffit d'utiliser leur nom. Voici ces noms :

```
>>> from reportlab.pdfbase import pdfmetrics
>>> pdfmetrics.standardFonts
('Courier', 'Courier-Bold', 'Courier-Oblique', 'Courier-BoldOblique',
'Helvetica', 'Helvetica-Bold', 'Helvetica-Oblique', 'Helvetica-BoldOblique',
'Times-Roman', 'Times-Bold', 'Times-Italic', 'Times-BoldItalic', 'Symbol',
'ZapfDingbats')
```

Leur utilisation est libre. Il est toujours possible d'utiliser d'autres polices d'écritures, qui peuvent potentiellement être soumises à licence, mais il faut alors les charger et les embarquer dans le document.

1.2.2 Canvas

Voici un petit script, disponible parmi ceux fournis avec cet ouvrage, qui permet d'utiliser le canvas. L'objectif est de montrer ce qu'il est possible de faire à bas niveau. D'abord les imports nécessaires :

```
>>> from reportlab.pdfgen.canvas import Canvas
>>> from reportlab.lib.units import cm
```

Voici comment créer un canvas et rajouter des métadonnées qui sont visibles par les systèmes d'exploitation et les lecteurs PDF :

```
>>> canvas = Canvas("hello.pdf")
>>> canvas.setTitle("Premier document")
>>> canvas.setSubject("Creation de document PDF avec ReportLab")
>>> canvas.setAuthor('SCH')
>>> canvas.setKeywords(['PDF', 'ReportLab', 'Python'])
>>> canvas.setCreator('sch')
```

Voici comment positionner un texte en précisant sa police d'écriture et sa taille :

```
>>> canvas.setFont("Helvetica", 36)
>>> canvas.drawCentredString(12.0 * cm, 18.0 * cm, "Hello world")
```

Voici un autre exemple intéressant :

```
>>> canvas.setFont("Times-Roman", 12)
>>> canvas.drawString(1.0 * cm, 1.0 * cm, "O")
>>> canvas.drawString(2.0 * cm, 1.0 * cm, "X")
>>> canvas.drawString(1.0 * cm, 2.0 * cm, "Y")
```

On a positionné ainsi un repère 0, X, Y qui permet de voir dans quel sens sont prises les mesures. Comme on peut le voir en lisant le document ainsi généré, le point 0 est en bas à gauche, l'axe X est vers la droite et l'axe Y vers le haut.
Tant que l'on ne change pas de police d'écriture, on continue de l'utiliser. On n'a pas besoin de le préciser à chaque écriture.

© Editions ENI - All rights reserved

Voici maintenant la notion d'alignement qui est mise en évidence :

```
>>> canvas.drawString(8.5 * cm, 16.0 * cm, "G")
>>> canvas.drawRightString(8.5 * cm, 15.0 * cm, "D")
>>> canvas.drawCentredString(8.5 * cm, 14.0 * cm, "C")
>>> canvas.drawString(12.5 * cm, 16.0 * cm, "Aligné à gauche")
>>> canvas.drawRightString(12.5 * cm, 15.0 * cm, "Aligné à droite")
>>> canvas.drawCentredString(12.5 * cm, 14.0 * cm, "Aligné au centre")
```

Enfin, est mis en évidence le fait que le changement de ligne ne peut se faire simplement et que les textes ne passent pas à la ligne automatiquement :

```
>>> canvas.drawCentredString(10.5 * cm, 10.0 * cm,
'\n'.join(["Aligné plein centre"] * 5))
>>> canvas.drawString(18.5 * cm, 12.0 * cm, "Mal Aligné" * 5)
>>> canvas.drawRightString(2.5 * cm, 12.0 * cm, "Mal Aligné" * 5)
>>> canvas.save()
```

Finalement, on enregistre et on crée ainsi le fichier.

1.3 Haut niveau

1.3.1 Styles

Un élément clé de la génération de document avec des outils de haut niveau est la manipulation de styles.
Voici pour commencer le moyen de récupérer une feuille de style standard :

```
>>> from reportlab.lib.styles import getSampleStyleSheet
>>> styles = getSampleStyleSheet()
>>> styles.list ()
```

Voici ce que cette commande donne pour chaque style :

```
BodyText None
     name = BodyText
     parent = <ParagraphStyle 'Normal'>
     alignment = 0
     allowOrphans = 0
     allowWidows = 1
     backColor = None
     borderColor = None
     borderPadding = 0
     borderRadius = None
     borderWidth = 0
     bulletFontName = Helvetica
     bulletFontSize = 10
     bulletIndent = 0
     firstLineIndent = 0
     fontName = Helvetica
     fontSize = 10
     leading = 12
     leftIndent = 0
```

```
    rightIndent = 0
    spaceAfter = 0
    spaceBefore = 6
    textColor = Color(0,0,0,1)
    textTransform = None
    wordWrap = None
```

Et voici la liste des autres styles standards :

- Bullet
- Code
- Definition
- Heading1
- Heading2
- Heading3
- Heading4
- Heading5
- Heading6
- Italic
- Normal
- Title

Voici comment créer un nouveau style de paragraphe :

```
>>> from reportlab.lib.styles import ParagraphStyle
>>> from reportlab.lib.enums import TA_JUSTIFY
>>> mon_style = ParagraphStyle(name='mon_style',
alignment=TA_JUSTIFY, fontName = "Helvetica", fontSize = 14)
```

Et voici comment le rajouter à la liste ci-dessus :

```
>>> styles.add(mon_style)
```

Ceci fait également partie du module vu en introduction.
L'utilisation de paramètres nommés est la règle dès lors que l'on en a beaucoup, ce qui permet une relecture de code plus claire, y compris lorsque l'on ne connaît pas par cœur les signatures des méthodes :

```
>>> help(ParagraphStyle)
```

On a donc les moyens de créer des styles spécifiques à la demande avant de commencer à créer le contenu. Tous sont applicables à des textes.
Cependant, il y a une particularité pour laquelle l'application de styles est particulière, il s'agit des tableaux :

```
>>> style_tableau = [
...     ('ALIGN',         (0,0),   (-1,-1), "LEFT"),
...     ('VALIGN',        (0,0),   (-1,-1), "TOP"),
...     ('LEFTPADDING',   (0,0),   (-1,-1), 0*cm),
...     ('RIGHTPADDING',  (0,0),   (-1,-1), 0*cm),
...     ('TOPPADDING',    (0,0),   (-1,-1), 0*cm),
...     ('BOTTOMPADDING', (0,0),   (-1,-1), 0*cm),
... ]
```

Cette feuille de style est une liste de 4-uplets qui désignent chacun respectivement le nom du style, la case en haut à gauche et la case en bas à droite qui englobent les cases du tableau concernées et la valeur associée au style.
Il est possible de créer un autre style à partir de celui-ci, pour éviter une double saisie :

```
>>> style_tableau1 = style_tableau[:]
>>> style_tableau1.append(('LINEABOVE', (0,0), (-1, 0), 1,
colors.turquoise))
```

© Editions ENI - All rights reserved

```
>>> style_tableau1.append(('LINEABOVE', (0,1), (-1,-1), 0.5,
colors.darkturquoise))
```

Il est très important de ne pas oublier les **[:]** pour faire une copie, sans quoi on ne fait que créer un pointeur vers la même liste.

1.3.2 Flux de données

Pour un document textuel qui est une suite de paragraphes, voire d'objets comme des images ou des tableaux, on écrit alors un flux de donnée.
Voici les éléments vus précédemment nécessaires :

```
>>> from reportlab.lib import colors
>>> from reportlab.lib.styles import getSampleStyleSheet
>>> from reportlab.lib.styles import ParagraphStyle
>>> from reportlab.lib.enums import TA_JUSTIFY
>>> from reportlab.lib.pagesizes import A4
>>> from reportlab.lib.units import cm
```

Le module utilisé est platypus et l'élément central est :

```
>>> from reportlab.platypus import SimpleDocTemplate
```

Il faut créer une liste d'autres objets de ce même module :

```
>>> flowables = []
```

Une fois que l'on a défini les styles comme présenté à la section précédente, un paragraphe est rajouté ainsi :

```
>>> from reportlab.platypus import Paragraph
>>> flowables.append(Paragraph("Fichier PDF Généré", styles["Heading1"]))
>>> flowables.append(Paragraph("Sébastien CHAZALLET", styles["Normal"]))
>>> flowables.append(Paragraph("http://www.inspyration.com",
styles["Code"]))
```

Pour chaque paragraphe, le style est précisé. Il est possible d'insérer un contenu sur plusieurs lignes, mais celles-ci ne formeront qu'une seule ligne dans le résultat final :

```
>>> content = """Ce document est généré par le script 02_flux.py.
... Ce script est livré avec le présent ouvrage.
... Vous pouvez le modifier à souhait pour faire vos propres expériences"""
>>> flowables.append(Paragraph(content, styles["Normal"]))
```

Il existe également un objet particulier pour effectuer un saut de ligne :

```
>>> from reportlab.platypus import  Spacer
>>> flowables.append(Spacer (0, 0.2*cm))
```

Un autre pour gérer le saut de page :

```
>>> from reportlab.platypus import   PageBreak
>>> flowables.append(PageBreak())
```

Voici comment écrire un tableau. Il s'agit d'une liste de listes de paragraphes qui chacun portent un style (en plus du style propre au tableau) :

```
>>> data, line = [], []
>>> line.append( Paragraph ("Technologie", styles["Normal"]) )
>>> line.append( Paragraph ("Logiciel", styles["Normal"]) )
>>> line.append( Paragraph ("Alternatives", styles["Normal"]) )
>>> data.append(line)
>>> line = []
>>> line.append( Paragraph ("OS", styles["Normal"]) )
>>> line.append( Paragraph ("Debian", styles["Normal"]) )
>>> line.append( Paragraph ("Ubuntu, Fedora", styles["Normal"]) )
>>> data.append(line)
```

Pour rajouter un tel tableau dans le flux principal, il faut procéder ainsi :

```
>>> from reportlab.platypus import Table
>>> flowables.append(Table(data, colWidths=[5*cm, 5*cm, 8*cm],
style=style_tableau1))
```

Voici comment rajouter une image :

```
>>> from reportlab.platypus import   Image
>>> flowables.append(Image('hello.pdf.png', height = 5 * cm, width
= 8 * cm))
```

Voici enfin comment créer le document final à partir du travail effectué :

```
>>> pdf = SimpleDocTemplate('test.pdf', pagesize = A4, title =
'Premier test', author = 'SCH')
>>> pdf.build(flowables)
```

La complexité de la création d'un fichier PDF est extrêmement basse, puisque l'instanciation de quelques objets et l'utilisation de listes et de n-uplets permettent de créer très simplement des documents contenant du texte sur lequel on peut appliquer des styles, des sauts de lignes, des sauts de page, des tableaux et des images comme montré ici.

Cette bibliothèque externe est relativement bien documentée et il suffit d'utiliser help sur différents objets ou méthodes pour retrouver des informations sur la manière de les utiliser. Il existe une documentation qui apporte notamment quelques exemples à reproduire.

Une fois que l'on maîtrise un peu la bibliothèque, réaliser des composants permettant d'automatiser la création de documents selon des règles métiers précises est relativement aisé.

Il est nécessaire d'aller un peu plus loin pour voir les possibilités offertes, comme par exemple la possibilité de créer des visuels, comme cela sera présenté dans la section qui suit, ou encore la possibilité de créer un traitement spécifique pour la première page via la signature de la méthode build, ou enfin créer des templates de pages permettant d'automatiser un numéro de page, ce qui sera présenté dans la section d'après.

Il est également à noter que la classe Image de la bibliothèque PIL et celle de la bibliothèque ReportLab sont compatibles et il est possible d'ouvrir une image, d'effectuer des traitements dessus (modifier les dimensions...) et la transformer en image ReportLab.

© Editions ENI - All rights reserved

1.3.3 Création d'un visuel

ReportLab permet de créer des visuels qui peuvent être intégrés facilement. Comme ce que l'on a vu précédemment, les données à produire sont décrites par l'utilisation de listes et n-uplets. Le paramétrage est à réaliser par le positionnement de différents attributs. Voici un exemple complet :

```
>>> from reportlab.graphics.shapes import Drawing
>>> from reportlab.graphics.charts.linecharts import HorizontalLineChart
>>> drawing = Drawing(10 * cm, 5 * cm)
>>> lc = HorizontalLineChart()
>>> lc.data = [
...     (0, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1000),
...     (0, 64, 128, 192, 256, 320, 384, 448, 512, 576, 640, 704),
... ]
>>> legend = ["Jan", "Fev", "Mar", "Avr", "Mai", "Jun", "Jul",
"Aou", "Sep", "Oct", "Nov", "Dec"]
>>> lc.categoryAxis.categoryNames = legend
>>> lc.valueAxis.valueMin = 0
>>> lc.valueAxis.valueMax = 1000
>>> lc.valueAxis.valueStep = 200
>>> lc.lines[0].strokeWidth = 2
>>> lc.lines[1].strokeWidth = 1.5
>>> drawing.add(lc)
>>> flowables.append(drawing)
```

L'exemple reste simple, surtout si l'on s'amuse à comparer avec d'autres modules qui sont dédiés à cette problématique.

1.3.4 Template de page

Le principe d'un template de page est de prendre à sa charge tous les travaux concernant la mise en forme d'une page indépendamment du flux.
Cela peut donc être les en-têtes et les pieds de page et un exemple classique consiste à calculer le numéro de la page courante.
Pour cela, il faut utiliser les composants suivants :

```
>>> from reportlab.platypus.doctemplate import PageTemplate
>>> from reportlab.platypus.frames import Frame
>>> from reportlab.lib.units import cm
```

Et procéder ainsi :

```
>>> class FlowTemplate (PageTemplate):
...     """Template for a pdf with datas in a flow."""
...     def __init__ (self, parent):
...         """Initialization of Template : llc = lower left corner"""
...         self.parent = parent
...         self.largeur = self.parent.document.pagesize[0]
...         self.hauteur = self.parent.document.pagesize[1]
...         self.marginx, self.marginy = 0.7 * cm, 1.4 * cm
...         self.llcx = self.largeur - self.marginx
```

```
...         self.llcy = 1.0 * cm
...         self.page = 0
...         content = Frame (self.marginx, self.marginy,
self.largeur - 2 * self.marginx, self.hauteur - 2 * self.marginy)
...         PageTemplate.__init__ (self, "Content", [content])
...     def beforeDrawPage (self, canvas, doc):
...         """before Drawing Page, we draw elements of the template"""
...         canvas.saveState ()
...         try:
...             self.drawTemplate(canvas, doc)
...         finally:
...             canvas.restoreState()
...     def drawTemplate(self, canvas, doc):
...         """Can be overridden"""
...         self.page += 1
...         #Dessin d'un carré noir
...         canvas.setFillColorCMYK( 0, 0, 0, 1 )
...         #canvas.setStrokeColorCMYK( 0, 0, 0, 1 )
...         canvas.rect(self.llcx, self.llcy, 0.4 * cm, 0.4 *
cm, stroke=0, fill=1 )
...         #Ajout du numéro de la page dans le carré noir.
...         canvas.setFont ('Helvetica', 8)
...         canvas.setFillColorCMYK( 0, 0, 0, 0 )
...         if self.page >= 10:
...             canvas.drawRightString (self.llcx + 0.35 *
cm, self.llcy + 0.1 * cm, "%d" % self.page)
...         else:
...             canvas.drawRightString (self.llcx + 0.3 *
cm, self.llcy + 0.1 * cm, "%d" % self.page)
...
```

À noter l'utilisation de la méthode – qui est en réalité un hook – beforeDrawPage qui contient le code à exécuter avant le traitement de la page. Il y a d'autres hooks.

1.3.5 Page contenant plusieurs zones

La manière la plus simple de créer un document est de faire se succéder plusieurs objets au sein d'un seul et même flux. L'utilisation de templates permet de créer des zones qui vont être rajoutées pour chaque page.

Un autre document type que l'on peut avoir à créer est une plaquette PDF, au sens d'un document d'une seule page pouvant contenir plusieurs zones qui peuvent être du texte, des images ou autre.

On sait déjà comment créer du contenu différent, ce qu'il manque est de créer plusieurs zones sur une seule et même page et de les remplir en parallèle.

Pour réaliser cela, on peut également créer un template qui contiendra toutes ces zones. Ainsi, créer un tel PDF n'est rien d'autre que de créer un flux par zone.

© Editions ENI - All rights reserved

Voici ce dont on a besoin :

```
>>> from reportlab.platypus.doctemplate import PageTemplate
>>> from reportlab.platypus.doctemplate import BaseDocTemplate
>>> from reportlab.lib import colors
```

Voici un template qui s'attache à un contexte (qui va contenir les flux) et crée lors de son initialisation trois zones de texte :

```
>>> class PlaquetteTemplate (PageTemplate):
...     """Modèle de Pages PDF pour une plaquette commerciale"""
...     def __init__ (self, context):
...         self.context = context
...         self.largeur = self.context.document.pagesize[0]
...         self.hauteur = self.context.document.pagesize[1]
...         self.zone1 = Frame (0.7*cm, 13*cm, self.largeur - 0.7*cm, 10*cm)
...         self.zone2 = Frame (0.7*cm, 9*cm, self.largeur - 0.7*cm, 6*cm)
...         self.zone3 = Frame (0.7*cm, 5*cm, self.largeur - 0.7*cm, 2*cm)
...         PageTemplate.__init__ (self, id="Tiers",
frames=[self.zone1, self.zone2, self.zone3], pagesize=A4)
```

Le template peut être vu comme un adaptateur du flux de données. En effet, chaque zone du template est directement reliée à un flux de données.
La classe utilise deux hooks. Le premier d'entre eux, déjà utilisé précédemment, permet simplement de remplir les zones créées avec le flux de données provenant du contexte :

```
...     def beforeDrawPage (self, canvas, doc):
...         canvas.saveState ()
...         try:
...             self.zone1.addFromList(
                        self.context.flowables_zone1, canvas)
...             self.zone2.addFromList(
                        self.context.flowables_zone2, canvas)
...             self.zone3.addFromList(
                        self.context.flowables_zone3, canvas)
...         finally:
...             canvas.restoreState()
```

On voit ici le lien entre zone du template et flux de données.
Le second hook (qui n'est pas obligatoire, présenté juste à titre d'exemple) permet d'effectuer des opérations une fois la page créée et les données du flux positionnées.
Ici, on crée un dessin minimaliste :

```
...     def afterDrawPage (self, canvas, doc):
...         canvas.saveState()
...         try:
...             canvas.setFillColorRGB(*colors.deeppink.rgba())
...             canvas.setStrokeColorRGB(*colors.lightcyan.rgba())
...             canvas.rect(0.7*cm, self.hauteur - 2*0.7*cm,
0.7*cm, 0.7*cm, fill=1)
...         finally:
...             canvas.restoreState ()
...
```

C'est l'endroit idéal pour ajouter des logos ou tout type de contenu qui ne serait pas lié au flux. Ces données seraient alors présentes quel que soit le fichier généré à l'aide du template.
La plaquette est l'adaptée (au sens du patron de conception « adaptateur ») du template. Elle reçoit des données et construit ses flux à partir d'elles (texte, images...). Pour cet exemple, on a simplifié au maximum, la problématique étant d'illustrer seulement le zonage :

```
>>> class PlaquettePDF:
...     def __init__ (self, datas):
...         self.datas = datas
...         self.built = 0
...         self.objects = [Spacer (0, 0.5*cm)]
...         self.styles = getSampleStyleSheet()
...         self.flowables_zone1=[Paragraph("ZONE 1", self.styles['Normal'])]
...         self.flowables_zone2=[Paragraph("ZONE 2", self.styles['Normal'])]
...         self.flowables_zone3=[Paragraph("ZONE 3", self.styles['Normal'])]
...         self.document = BaseDocTemplate ("plaquette.pdf", leftMargin=0.7*cm,
rightMargin=0.7*cm, topMargin=0.7*cm, bottomMargin=0.7*cm, pagesize=A4)
...         self.document.addPageTemplates ( PlaquetteTemplate(self))
...         self.document.build (self.objects)
...         self.built = 1
...
```

Il est à noter les trois dernières lignes qui font la construction effective du document ainsi que la ligne précédente qui lie le template au flux.
Il suffit alors d'instancier l'objet pour créer le document :

```
>>> PlaquettePDF('')
<__main__.PlaquettePDF instance at 0x302ccf8>
```

Le résultat est alors visible et il est d'ailleurs joint à cet ouvrage, avec les codes sources des exemples présentés dans ces chapitres.
Une fois ces bases techniques acquises, il ne reste plus qu'à laisser exprimer la créativité pour éditer des documents efficaces et professionnels.

2. OpenDocument

2.1 Présentation

Soyons clairs dès le départ. OpenDocument est un format de document, partagé par tous les éditeurs de logiciels de bureautique dignes de ce nom. Ce format a été élaboré minutieusement, négocié par les éditeurs qui ont été partie prenante de ce processus. OpenDocument est donc un format qui est censé être partagé par tous les logiciels de bureautique. On ne parle donc pas dans cette section de LibreOffice (la suite bureautique de référence), mais bien d'un format de document universel.
Et à ce niveau-là, le format est assez simple à comprendre. Il s'agit, concrètement, d'un simple fichier compressé au format ZIP et contenant de gros fichiers XML parmi lesquels on trouve **`content.xml`**, qui comprend en son sein tout le contenu du document et le format utilisé, incluant le style). Un autre fichier, **`style.xml`**, détaille chacun des styles personnalisés du document.

© Editions ENI - All rights reserved

On peut donc accéder au contenu d'un fichier OpenDocument simplement en le dézippant et en utilisant XPath pour le traverser.
Nous allons présenter ici des méthodes de haut niveau pour générer des documents de type OpenDocument, et pour ce faire la référence est ce site :
http://www.opendocumentformat.org/developers/

2.2 ezodf2

2.2.1 Installation

La bibliothèque idéale pour générer un document au format OpenDocument est ezodf2 (https://pypi.python.org/pypi/ezodf2). Il faut l'installer de la manière suivante :

```
$ sudo pip-3.2 install ezodf2
```

Ouvrir une console Python et importer le module devrait alors fonctionner.

```
>>> import ezodf2
```

2.2.2 OpenDocument Texte

Voici un exemple minimaliste tiré de la documentation sur PyPI :

```
>>> from ezodf2 import newdoc, Paragraph, Heading
>>> odt = newdoc(doctype='odt', filename='text.odt')
>>> odt.body += Heading("Chapitre 1")
>>> odt.body += Paragraph("C'est le début de l'histoire.")
>>> odt.save()
```

La première ligne est l'import des éléments nécessaires pour générer un document. La fabrique newdoc permet de créer n'importe quel type de document au format OpenDocument. Lorsqu'il s'agit d'un document texte, on dispose alors d'un attribut body auquel il suffit d'ajouter les paragraphes.

2.2.3 OpenDocument Tableur

La création d'un tableur est une problématique usuelle. Avec ezodf2, il suffit d'utiliser la fabrique puis de créer les feuilles et de les rajouter au document.
Voici un exemple tiré de la documentation que l'on peut trouver sur PyPI :

```
>>> from ezodf2 import newdoc, Sheet
>>> ods = newdoc(doctype='ods', filename='spreadsheet.ods')
>>> sheet = Sheet('SHEET', size=(10, 10))
>>> ods.sheets += sheet
>>> sheet['A1'].set_value("cell with text")
>>> sheet['B2'].set_value(3.141592)
>>> sheet['C3'].set_value(100, currency='USD')
>>> sheet['D4'].formula = "of:=SUM([.B2];[.C3])"
>>> pi = sheet[1, 1].value
>>> ods.save()
```

Les cellules sont gérées à l'aide de leurs coordonnées composées de lettres et chiffres.

2.2.4 Aller plus loin

La documentation du projet est assez détaillée et vous permettra de générer très vite des documents (http://pythonhosted.org/ezodf/). Vous pourrez aussi trouver des exemples concrets dans le dépôt du projet :
https://github.com/iwschris/ezodf2/tree/master/examples

2.3 Alternatives

2.3.1 lpod

Une bibliothèque similaire à ezodf2 et potentiellement plus adaptée à Python 2 est python-lpod. Elle est malheureusement mise à jour assez peu fréquemment et sa documentation est très insuffisante, mais c'est tout de même une bibliothèque assez complète. Il faut l'installer de la manière suivante :

```
$ sudo pip2.7 install lpod-python
```

Ouvrir une console Python et importer le module devrait alors fonctionner.

```
>>> import lpod
```

Voici maintenant un exemple minimaliste identique au précédent :

```
>>> from lpod.document import odf_new_document
>>> from lpod.heading import odf_create_heading
>>> from lpod.paragraph import odf_create_paragraph
>>> odt = odf_new_document('text')
>>> odt.get_body().append(odf_create_heading(1, u"Chapitre 1"))
>>> odt.get_body().append(odf_create_paragraph(u"C'est le début
de l'histoire."))
>>> odt.save(target='text.odt', pretty=True)
```

On voit que la syntaxe est beaucoup plus lourde, ce qui est la conséquence du fait que ce projet n'est pas conçu que pour Python, mais est fait pour correspondre aux capacités de différents langages.
On rappelle aussi que l'on est en Python 2 et qu'il ne faut utiliser que des chaînes de caractères Unicode.
On retiendra malheureusement surtout le fait que la documentation du projet est à la fois incomplète et non à jour. Ceci est un frein assez important à l'utilisation de cette bibliothèque.
Vous disposez cependant d'un certain nombre d'exemples qui peuvent vous permettre de vous en sortir : https://github.com/lpod/lpod-python-recipes/tree/master/Examples

2.3.2 Génération à partir de templates

Il existe de nombreux projets vous permettant de générer des documents à partir d'un template et d'un jeu de données. Le template peut utiliser du métacode inséré dans le document, lequel peut être totalement personnalisé, comme avec POD (http://appyframework.org/pod.html), ou utiliser un moteur de rendu déjà éprouvé comme Genshi (http://relatorio.readthedocs.org/en/latest/index.html) ou encore Jinja2 (https://github.com/christopher-ramirez/secretary).

© Editions ENI - All rights reserved

On notera également qu'il existe odfpy qui propose des outils en ligne de commande (https://github.com/eea/odfpy), ou encore un projet qui vous permet de gérer facilement la traduction de vos documents dans plusieurs langues (http://www.hforge.org/odf).
Enfin, sachez que vous pouvez scripter des outils comme LibreOffice à l'aide de Python-UNO, c'est-à-dire écrire des macros en Python au lieu de les écrire en scripts OOo Basic, ce qui est toujours plus confortable. Vous pouvez aussi démarrer LibreOffice en mode serveur et écrire en Python des clients qui envoient des données au serveur pour récupérer un document généré.

3. Travailler avec des images

3.1 Représentation informatique d'une image

Lorsque l'on pense image, on pense photographie et donc image bitmap qui, comme son nom l'indique, est une organisation d'octets. L'image a une taille fixe et une résolution donnée qui fait que l'on peut considérer une image comme trois tableaux statiques superposés d'octets, un pour le rouge, un autre pour le vert et un dernier pour le bleu, chaque case du tableau ou point de l'image portant donc comme valeur un 3-uplet d'octets.
Ceci correspond à ce que l'on appelle une image matricielle et est le type d'image le plus commun. C'est ce type d'image que prend un appareil photo, par exemple. En paraphrasant le paragraphe ci-dessus, on peut voir une image comme trois matrices superposées, chaque valeur étant un octet, ou comme une seule matrice de 3-uplets d'octets.
Si d'un premier abord la différence ne saute pas forcément aux yeux, ces façons de voir sont importantes car elles déterminent le contenu des algorithmes.
Ces images sont parfaitement adaptées à une utilisation informatique et à une restitution sur un écran d'ordinateur, puisque les trois composantes, rouge, verte et bleue sont celles de la synthèse additive. C'est-à-dire que l'addition des couleurs donne la couleur blanche et l'absence combinée des trois mêmes couleurs correspond au noir.
Au contraire, la reprographie nécessite d'utiliser la synthèse soustractive, celle qui est à l'œuvre lors des mélanges de pigments (ou de peinture) et qui est un peu plus connue du grand public. Le mélange de magenta et de cyan donne du violet, magenta et jaune de l'orange, jaune et cyan du vert et le mélange des trois donne du marron. Dans la synthèse soustractive, il est très difficile d'obtenir du noir et les niveaux de gris, il faut donc des pigments supplémentaires. C'est la raison pour laquelle la représentation CMJN (cyan, magenta, jaune, noir) a été conçue.
La représentation d'une telle image destinée à la reprographie nécessite donc non plus un 3-uplet, mais un 4-uplet pour chaque pixel ou 4 matrices au lieu de 3 pour l'image complète. Il est bien évident que le passage du 4-uplet vers le 3-uplet est unique, mais celui du 3-uplet vers le 4-uplet peut avoir plusieurs solutions. Les graphistes ont donc conçu des équivalences entre les deux modes de représentation, mais il y en a plusieurs et elles sont fortement liées aux constructeurs de matériels de reprographie. C'est en effet au bon passage de la synthèse additive à la synthèse soustractive que se juge la qualité de la fidélité des couleurs entre l'écran et l'impression et donc la qualité du matériel de reprographie.

L'autre grand format d'image est celui des infographistes. Il s'agit des images vectorielles. Celles-ci ne sont pas des matrices de pixels, mais des assemblages d'objets géométriques, donc descriptibles via une fonction mathématique. Leur avantage est de faciliter le travail de conception d'infographie et de pouvoir être facilement transformées en image matricielle (la réciproque n'est pas vraie), ce qui se produit lorsque l'on fait une simple impression. Nous verrons les deux types.

3.2 Présentation de Pillow

Pillow est un fork amical du projet PIL qui a permis, entre autres, son portage sous Python 3. PIL est l'acronyme de *Python Imaging Library*. Il s'agit, comme son nom l'indique, d'une bibliothèque de manipulation des images matricielles pour Python et elle supporte BMP, GIF, JPG, PNG et TIFF entre autres.

Elle dispose d'une documentation officielle intéressante (http://www.pythonware.com/library/pil/handbook/) et de beaucoup d'autres ressources sur Internet qui permettent de travailler assez facilement sur les images.

Les appareils photos numériques actuels permettent de réaliser des photographies de très bonne qualité et donc d'une dimension importante. Une des fonctionnalités essentielles est d'être capable de redimensionner une telle photographie.

Par exemple, on peut imaginer un site web proposant des galeries de photos. L'affichage de vingt photos par page à 15 Mo la photo serait une erreur grossière. La solution idéale est de proposer trois images pour chaque photo : une à largeur fixe de 100 ou 150 pixels pour la miniature affichée pour la galerie, une photo de 600 pixels de large pour l'affichage de la photo à l'écran et un lien vers la photo au format d'origine pour le téléchargement.

Pour installer la bibliothèque, il faut s'y prendre ainsi :

```
$ sudo pip-3.2 install pillow
```

Une fonction de redimensionnement peut s'écrire ainsi :

```
>>> from PIL import Image
>>> import os.path
>>> def resize(filename):
...     file, ext = os.path.splitext(filename)
...     im = Image.open(filename)
...     w, h = im.size
...     im.thumbnail((600, int(600.*h/w)), Image.ANTIALIAS)
...     im.save(file+"_screen"+ext, 'PNG')
...     im.thumbnail((150, int(150.*h/w)), Image.ANTIALIAS)
...     im.save(file+"_thumbnail"+ext, 'PNG')
...
>>> resize('original.png')
```

On ouvre une image, on récupère ses dimensions, on utilise la méthode **`thumbnail`** pour réduire sa taille en déterminant une largeur et en calculant la hauteur en conséquence afin de ne pas déformer l'image. Il ne reste plus qu'à sauvegarder le travail en déterminant le nouveau nom de l'image en fonction de l'image d'origine.

© Editions ENI - All rights reserved

On se retrouve donc avec deux images de largeur fixe : « original_screen.png » de 600 pixels de large et « original_thumbnail.png » de 150 pixels. Ce script est disponible en tant qu'exemple dans une version plus souple, permettant de redimensionner dans plusieurs formats au choix.

On voit donc dans ces quelques lignes que la manipulation d'images est relativement simple et que la maîtrise de Python permet de prendre en main cette bibliothèque sans difficultés. Seule la connaissance du fonctionnement des formats et des techniques liées au stockage des informations dans les images matricielles est, au final, nécessaire.

3.3 Formats d'images matricielles

On va se concentrer uniquement sur les formats BMP, GIF, JPG, PNG et TIFF que l'on va commencer par présenter rapidement.

Le format BMP est un format d'image matricielle qui représente pour chaque point sur un bit (mode noir et blanc), 2 bits (16 couleurs), 8 bits ou un octet pour 256 couleurs (ou un mode en niveau de gris) à 24 bits (16,8 millions de couleurs soit un 3-uplet d'octets).

Ces différences de représentations permettent de gagner en taille au détriment de la qualité de l'image et ont été utiles par le passé avec du matériel qui ne pouvait pas afficher beaucoup de couleurs. Aujourd'hui, seul le mode 24 bits est réellement utile, à de rares exceptions près (image en niveau de gris, par exemple) pour la simple et bonne raison qu'il existe d'autres moyens de gagner en taille à l'aide des autres formats.

Contrairement à une idée répandue, BMP dispose d'un mécanisme de compression des données qui lui permet de prendre moins de place que n'en nécessiterait un stockage sans compression (taille de la représentation d'un pixel multipliée par le nombre de pixels, soit pour une image de 800 par 600 à 24 bits 1,44 Mo), mais il est peu efficace au regard de ce que proposent les autres formats.

Ceci a pour conséquence que le format BMP est très utilisé pour la manipulation d'images, en particulier avec les logiciels (outre les formats liés aux logiciels eux-mêmes) et il a le grand avantage d'être facile à manipuler à bas niveau. Par contre, il n'est vraiment pas fait pour le Web, à cause de son poids, mais les graphistes gardent le BMP en tant que format original pour le retravailler et l'exportent vers d'autres formats spécifiquement pour le Web.

Le format GIF est un format d'image ancien, particulièrement destiné à corriger principalement le problème du poids du format BMP. La solution trouvée consiste à déterminer à partir du contenu de l'image une palette minimaliste de 256 couleurs tous canaux confondus et représentables sur un octet pour déterminer la couleur de chaque pixel par rapport à cette palette qui est une information qui fait partie intégrante de l'image. Pour une taille brute (non compressée) équivalente à un BMP de 8 bits, on obtient une image de meilleure qualité parce que l'on se concentre sur les couleurs réellement présentes dans l'image. Ce format prend donc toute son efficacité pour des images homogènes en termes de couleur.

Cette technique s'accompagne par ailleurs de l'utilisation d'un algorithme de compression plus efficace que celui de BMP. Le tout en a fait un élément essentiel des premières heures du Web, parce qu'il correspondait exactement au besoin. Par contre, son utilisation au sein des logiciels de traitement d'image est moins évidente, de par la perte de données due au format lui-même, et il faut conserver les formats d'images originaux pour pouvoir les retravailler par la suite.

Le format GIF permet également de stocker plusieurs images en leur attribuant un temps d'affichage, permettant la création très facile d'images animées.
Aujourd'hui, d'autres formats plus performants et moins complexes voient le jour et le GIF reste surtout utilisé pour le GIF animé parce que les concurrents ne sont pas libres pour certains ou pas totalement démocratisés pour d'autres.
JPEG est l'acronyme de *Joint Photographic Experts Group* et, comme son nom l'indique, est un comité d'experts. Le mot JPEG désigne la norme aussi bien que l'algorithme qui l'implémente et qui est en partie breveté.
JPEG désigne aussi l'extension de ce type d'image et le format lui-même, mais l'on devrait plutôt parler de JIFF pour *JPEG Image File Format*, car d'autres formats de fichiers peuvent également utiliser partiellement ces algorithmes.
Si le BMP est un format de travail et de qualité et le GIF un format orienté sur la résolution des problématiques, le JPEG est orienté sur les problématiques photographiques. Il résout la problématique de la qualité par rapport au poids du fichier en tirant profit des caractéristiques de l'œil humain pour limiter au maximum la perte de données sur celles auxquelles l'œil est sensible (luminance) au détriment de celles qu'il perçoit moins (chrominance). On a ainsi une perte de qualité focalisée sur les éléments les moins perceptibles par l'œil humain.
Cette façon originale de procéder est réalisée sur des ensembles de pixels de 8 par 8 ou de 16 par 16 et est complétée par une transformation numérique de chacun permettant de ne plus stocker les valeurs absolues des couleurs de chaque pixel, mais des matrices (cartes) de variation en fréquence et en amplitude. Les amplitudes les plus fortes sont alors atténuées et des opérations mathématiques sur l'ensemble des matrices sont réalisées afin d'obtenir des matrices d'une forme particulière permettant le stockage le plus léger possible. Le résultat ainsi obtenu est alors compressé.
Le résultat visuel est très net sur des photographies qui peuvent garder une très bonne qualité malgré une forte compression, mais se dégrade à tous les endroits où les teintes sont proches, amenant un effet pixelisé flagrant (aplat de couleurs).
PNG, acronyme de *Portable Network Graphics*, est un format ouvert d'image matricielle créé pour remplacer GIF. Il se positionne ainsi sur le même créneau, à savoir permettre de disposer d'images qui ne soient pas trop lourdes à télécharger et donc pénalisantes pour les clients web. Il a toutefois l'énorme avantage d'être libre, non destructeur (pas de perte de données et donc de qualité), et beaucoup plus performant, puisqu'il est à la fois adapté aux problématiques des aplats de couleurs, n'est pas limité à 256 couleurs, reste léger et dispose d'une compression efficace via l'algorithme **`deflate`** vu dans le chapitre Programmation système - Appréhender son système d'exploitation. Il permet de coder un pixel sur 1 bit (monochrome, mais pas forcément noir et blanc), 2 bits pour la quadrichromie, 8 bits pour du niveau de gris ou un mécanisme de palette de couleurs identique à celui de GIF jusqu'à 48 bits. Il dispose d'une couche alpha lui permettant de gérer des niveaux de transparence.
À part JPEG qui est plus adapté pour le cas précis de la photographie, il est à l'heure actuelle le format le plus adapté pour le Web.

© Editions ENI - All rights reserved

TIFF, acronyme de *Tagged Image File Format*, est un format d'image matricielle qui est un conteneur conçu pour être le plus flexible possible pour permettre la résolution de problématiques de bas niveau afin d'être utilisé par divers matériels. Il peut être little endian ou big endian, coder chaque pixel sur 1 à 64 bits, accepter plusieurs espaces colorimétriques (RVB comme le bitmap, CMJN comme les imprimantes, TcbCr comme les JPEG...). Il permet aussi le stockage de métadonnées et dispose d'un canal alpha. Il se trouve un peu au croisement des autres formats.

3.4 Récupérer des informations d'une image

Lorsque l'on manipule une image, il est nécessaire de connaître des informations à son propos, de manière à déterminer comment on va la manipuler, voire pour opérer des transformations préliminaires.
Pour ouvrir une image, il suffit d'une seule ligne (outre l'import) :

```
>>> from PIL import Image
>>> im = Image.open(filename)
```

La méthode **open** détermine le format de l'image et permet à partir de celui-ci de représenter l'image sous la forme qui convient. Et pour avoir des informations sur sa représentation, il faut procéder ainsi :

```
>>> im.format
'PNG'
>>> im.format_description
'Portable network graphics'
>>> im.tile
[('zip', (0, 0, 4288, 2848), 94, 'RGB')]
>>> im.getbbox()
(0, 0, 4288, 2848)
>>> im.size
(4288, 2848)
>>> im.mode
'RGB'
>>> im.getbands()
('R', 'G', 'B')
>>> im.getextrema()
((0, 255), (0, 255), (0, 255))
>>> im.info
{'dpi': (72, 72)}
```

Dans l'ordre d'apparition, on a le format de l'image et sa description, les informations générales sur l'image dont la compression, le cadre actuel que l'on peut aussi avoir par la méthode **getbbox**, et le mode que l'on retrouve également dans l'attribut éponyme, puis la taille de l'image, la liste des canaux, les valeurs qu'ils peuvent prendre et des informations annexes.
Ceci peut varier d'une image à une autre. Les modes principaux sont :
- 1 : monochrome ;
- L : niveaux de gris (peut être utilisé pour manipuler les données d'un seul canal) ;

- P : représentation sur un octet liée à une palette ;
- RGB : mode Bitmap (rouge, vert, bleu) ;
- RGBA : identique avec le canal alpha ;
- CMYK : CMJN, adapté à la reprographie (voir présentation en début de section) ;
- YCbCr : format utilisé par le JPEG (voir présentation des formats).

La palette n'est définie que pour les formats l'utilisant. Le passage d'un mode à un autre, ou la conversion d'un format à un autre, est réalisable, mais peut se faire avec perte si l'on passe d'un mode ou format détaillé à un autre plus restreint, par exemple, du RGBA vers le RGB ou vers le P.

3.5 Opérations d'ensemble sur une image

Il y a un certain nombre d'opérations classiques qui sont aisément réalisables : http://effbot.org/imagingbook/image.htm

Nous avons vu en introduction comment redimensionner une image. Nous avons utilisé la méthode **thumbnail** :

```
>>> im.thumbnail((new_w, new_h), Image.ANTIALIAS)
```

Cette méthode transforme l'image courante pour la retailler à une dimension forcément inférieure.

Il existe également la méthode **resize** qui n'agit pas sur l'objet courant, mais va renvoyer une nouvelle instance homogène à l'objet courant et qui sera la nouvelle image à la nouvelle taille.

Le second argument correspond à un filtre à appliquer à la transformation. Ces filtres, comme les autres concepts liés à la manipulation d'image, sont détaillés dans la documentation officielle : http://effbot.org/imagingbook/concepts.htm

Leur signification est :

- NEAREST : il s'agit de prendre le pixel de l'image originale le plus près de la position de la nouvelle image. Qualité mauvaise, mais rapide.
- BILINEAR : réalise une interpolation linéaire d'un carré de 4 pixels autour de cette même position, une sorte de moyenne. Qualité meilleure, un peu plus lent.
- BICUBIC : réalise une interpolation cubique sur un carré de 16 pixels autour de cette même position. Qualité encore meilleure, mais encore plus lent.
- ANTIALIAS : utilise un filtre complexe pour déterminer les pixels à prendre en compte pour calculer la valeur finale du pixel de sortie. Qualité meilleure que la précédente, mais encore un peu plus lente.

Une autre manipulation classique consiste à faire pivoter une image (degrés en sens horaire). Il existe pour cela la méthode **rotate** qui comme **resize** ne touche pas à l'image d'origine et crée une nouvelle image.

Voici un script :

```
>>> im = Image.open('original_thumbnail.png')
>>> im2 = im.rotate(10)
>>> im3 = im.rotate(30)
>>> im4 = im.rotate(45)
>>> im5 = im.rotate(90)
```

© Editions ENI - All rights reserved

```
>>> im6 = im.rotate(180)
>>> for i, im in enumerate([im2, im3, im4, im5, im6]):
...     im.save('original_thumbnail_%s' % i, 'PNG')
```

Le résultat montre qu'une rotation différente de 90° ne modifie pas le cadre, puisque la taille originelle est gardée et donc une partie de l'image est tronquée.
Pour réaliser une rotation de 90, 180 ou 270 degrés (toujours pris dans le sens horaire), il est possible d'utiliser la méthode **transpose** avec le degré demandé. L'opération est alors simple dès lors que l'on travaille sur des matrices :

```
>>> im = Image.open('original_thumbnail_30.png')
>>> im2 = im.transpose(Image.ROTATE_90)
>>> im2.save('original_thumbnail_80.png', 'PNG')
```

Cette même méthode est utilisable pour obtenir une symétrie de l'image de la gauche vers la droite ou du haut vers le bas :

```
>>> im = Image.open('original_thumbnail.png')
>>> im2 = im.transpose(Image.FLIP_LEFT_RIGHT)
>>> im2.save('original_thumbnail_flip.png', 'PNG')
```

Il est possible d'appliquer des modifications à l'ensemble des pixels de l'image :

```
>>> im2 = im.point(lambda i: 128+(i-128)//2)
>>> im2.save('original_thumbnail_point.png', 'PNG'
```

La fonction représente une transformation des valeurs qui est appliquée à chaque calque.
Il est également possible d'utiliser des filtres déjà tout prêts :

```
>>> from PIL import ImageFilter
>>> im2 = im.filter(ImageFilter.BLUR)
>>> im2.save('original_thumbnail_filter_blur.png', 'PNG')
>>> im2 = im.filter(ImageFilter.CONTOUR)
>>> im2.save('original_thumbnail_filter_contour.png', 'PNG')
>>> im2 = im.filter(ImageFilter.DETAIL)
>>> im2.save('original_thumbnail_filter_detail.png', 'PNG')
>>> im2 = im.filter(ImageFilter.EDGE_ENHANCE)
>>> im2.save('original_thumbnail_filter_edge_enhance.png', 'PNG')
>>> im2 = im.filter(ImageFilter.EDGE_ENHANCE_MORE)
>>> im2.save('original_thumbnail_filter_edge_enhance_more.png', 'PNG')
>>> im2 = im.filter(ImageFilter.EMBOSS)
>>> im2.save('original_thumbnail_filter_emboss.png', 'PNG')
>>> im2 = im.filter(ImageFilter.FIND_EDGES)
>>> im2.save('original_thumbnail_filter_find_edges.png', 'PNG')
>>> im2 = im.filter(ImageFilter.SMOOTH)
>>> im2.save('original_thumbnail_filter_smooth.png', 'PNG')
>>> im2 = im.filter(ImageFilter.SMOOTH_MORE)
>>> im2.save('original_thumbnail_filter_smooth_more.png', 'PNG')
>>> im2 = im.filter(ImageFilter.SHARPEN)
>>> im2.save('original_thumbnail_filter_sharpen.png', 'PNG')
```

Le plus simple pour comprendre l'effet produit par le filtre est de le tester, cependant les noms doivent parler à l'utilisateur infographiste, habitué à utiliser des logiciels de retouche d'image comme GIMP (dont Python est le langage de scripting).
Enfin, une fonctionnalité essentielle est le recadrage :

```
>>> im2 = im.crop((25, 15, 100, 75))
>>> im2.save('original_thumbnail_recadrage.png', 'PNG')
```

Elle prend en paramètre un 4-uplet qui représente les coins haut gauche et bas droit de l'image et dont les valeurs doivent être comprises dans celles disponibles pour l'image.
Au final, l'offre est relativement étoffée et permet de reproduire des opérations courantes en infographie, ce qui fait de Python un langage de scripting idéal.

3.6 Travailler avec les calques ou les pixels

Voici, en une ligne comment récupérer les calques d'une image, après avoir importé le module et ouvert l'image :

```
>>> r, v, b = im.split()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.3/dist-packages/Pillow/Image.py", line 1497,
in split
    if self.im.bands == 1:
AttributeError: 'NoneType' object has no attribute 'bands'
```

Visiblement, il y a un souci. Ceci est dû au lazy-loading, soit au chargement à la demande en français. On ne dispose pas des données tant que l'on n'en a pas besoin. Il faut donc passer par une ligne intermédiaire pour charger explicitement les données de l'image.

```
>>> im.load()
<PixelAccess object at 0x23a8190>
>>> r, v, b = im.split()
```

La méthode **load** renvoie un objet particulièrement utile dont on parlera en temps utile. Pour l'instant, on va se concentrer sur les trois calques que l'on a obtenus (le nombre de calques dépend du mode).
Il est possible, pour commencer, d'enregistrer chaque calque dans un fichier séparé :

```
>>> im2 = Image.merge('L', (r,))
>>> im2.save('original_thumbnail_R.png', 'PNG')
>>> im2 = Image.merge('L', (v,))
>>> im2.save('original_thumbnail_V.png', 'PNG')
>>> im2 = Image.merge('L', (b,))
>>> im2.save('original_thumbnail_B.png', 'PNG')
```

L'opération la plus simple des calques consiste à remplacer l'un par l'autre :

```
>>> im2 = Image.merge('RGB', (b, r, v))
>>> im2.save('original_thumbnail_BRV.png', 'PNG')
>>> im2 = Image.merge('RGB', (v, b, r))
>>> im2.save('original_thumbnail_VBR.png', 'PNG')
>>> im2 = Image.merge('RGB', (b, v, r))
```

© Editions ENI - All rights reserved

```
>>> im2.save('original_thumbnail_BVR.png', 'PNG')
>>> im2 = Image.merge('RGB', (r, b, v))
>>> im2.save('original_thumbnail_RBV.png', 'PNG')
>>> im2 = Image.merge('RGB', (v, r, b))
>>> im2.save('original_thumbnail_VRB.png', 'PNG')
```

Il est aussi possible de créer facilement un calque vide et ainsi de jouer avec les additions de calques :

```
>>> t = Image.new('L', im.size)
>>> im2 = Image.merge('RGB', (r, t, t))
>>> im2.save('original_thumbnail_RTT.png', 'PNG')
>>> im2 = Image.merge('RGB', (t, v, t))
>>> im2.save('original_thumbnail_TVT.png', 'PNG')
>>> im2 = Image.merge('RGB', (t, t, b))
>>> im2.save('original_thumbnail_TTB.png', 'PNG')
>>> im2 = Image.merge('RGB', (r, v, t))
>>> im2.save('original_thumbnail_RVT.png', 'PNG')
>>> im2 = Image.merge('RGB', (r, t, b))
>>> im2.save('original_thumbnail_RTB.png', 'PNG')
>>> im2 = Image.merge('RGB', (t, v, b))
>>> im2.save('original_thumbnail_TVB.png', 'PNG')
```

Cela donne des résultats surprenants, mais sympathiques (et livrés avec les autres exemples de code).
Une fois que l'on a un calque, on peut y appliquer toutes les transformations d'ensemble que l'on a vues pour les images. Attention cependant à ce que l'on fait :

```
>>> b2 = b.transpose(Image.FLIP_TOP_BOTTOM)
>>> v2 = v.transpose(Image.FLIP_LEFT_RIGHT)
>>> im2 = Image.merge('RGB', (r, v2, b2))
>>> im2.save('original_thumbnail_etrange.png', 'PNG')
```

Si l'on n'a pas encore été aveuglé par les images précédentes, on a des chances d'être plus surpris par la visualisation de cette dernière image, mais cela reste original.
On peut également travailler sur les pixels.

```
>>> pix = b.load()
>>> pix[0, 0]
45
```

L'opérateur crochet est utilisé pour accéder à la valeur d'un pixel du calque, dont la coordonnée est représentée par un tuple. Cette valeur se situe ici entre 0 et 255.
Ceci est exactement similaire à ce que l'on a vu au début de la section pour l'image où l'on retrouve alors notre 3-uplet (dont la troisième valeur est la précédente) :

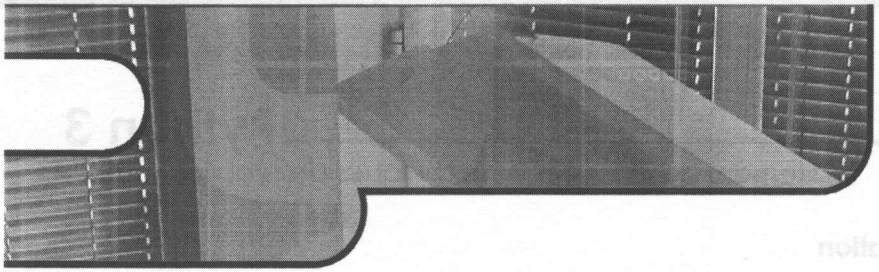
```
>>> pix = im.load()
>>> pix[0, 0]
(130, 77, 45)
```

Il est possible de modifier ces valeurs pour obtenir des effets, par une fonction mathématique ou par un algorithme. Voici un exemple assez trivial permettant de réaliser un cadre :

```
>>> im = Image.open('original_thumbnail.png')
>>> im.load()
<PixelAccess object at 0x23a80d0>
>>> pix = im.load()
>>> for x in [0, 1, 2, 3, 4, 145, 146, 147, 148, 149]:
...     for y in range(99):
...         pix[x, y] = (255, 255, 255)
...
>>> for y in [0, 1, 2, 3, 4, 94, 95, 96, 97, 98]:
...     for x in range(5, 145):
...         pix[x, y] = (255, 255, 255)
...
>>> im.save('original_thumbnail_cadre.png', 'PNG')
```

Réaliser des opérations sur des images directement par du scripting nécessite de bonnes connaissances sur les caractéristiques des images et les mathématiques qui y sont associées, mais Python propose une application simple.
Il est à noter la méthode **paste** qui permet de rajouter des calques dans l'image et d'offrir un panel d'outils complet qu'il convient de s'approprier.

© Editions ENI - All rights reserved



Chapitre 1.4
Qualité

1. Programmation dirigée par les tests

1.1 Tests unitaires

1.1.1 Principes

La programmation impérative ou objet nécessite la création de classes, d'instances et de fonctions.
Celles-ci ont un comportement déterministe qui est reproductible. En clair, une fonction utilisée avec les mêmes paramètres produit le même résultat.
Voici un exemple simple :

```
>>> def carre(x):
...     return x**2
...
```

Il est possible de créer un test sur cette fonction. Cela consiste à mettre en regard une série de paramétrages et pour chacun le résultat attendu :

```
>>> def test_carre():
...     for x, r in {2: 4, 0: 0, -2:4}.items():
...         if carre(x) != r:
...         print('test échoué')
...         break
...     else:
...         print('test réussi')
...
```

1.1.2 Interprétation

Lorsque l'on exécute cette fonction, deux résultats sont possibles. Il se peut que la fonction réussisse :

```
>>> test_carre()
test réussi
```

Cela ne signifie pas que la fonction est correcte, mais qu'elle passe tous les tests prévus. La représentativité des tests est donc cruciale. Voici un exemple :

```
>>> def moyenne(valeurs):
...     return sum(valeurs)/len(valeurs)
...
```

Si l'on ne teste pas spécifiquement la fonction avec une liste de valeurs vide, on passe à côté d'un cas d'utilisation. Voici un autre cas (tester [-5, 6, -1]) :

```
>>> def balance(valeurs):
...     sum([abs(n) for n in valeurs])/sum(valeurs)
...
```

L'autre possibilité est que le test soit négatif. Là encore, cela ne veut pas forcément dire que la fonction n'est pas valide. Cela signifie simplement que l'un des tests ne passe pas. Il faut alors vérifier s'il s'agit d'une erreur de la fonction testée ou d'une erreur du test.
Voici un exemple d'une fonction particulièrement mal écrite :

```
>>> def choix(valeur):
...     if valeur < 0:
...         return 0
...     elif valeur > 20:
...         return 20
...     elif valeur > 10:
...         return valeur-10
...     elif valeur > 5:
...         return int(valeur/2)
...     elif valeur > 15:
...         return int((valeur-10)/2)
...
```

Dans ce genre de fonction dont la composition est une suite de conditions, un des tests consiste à vérifier que l'on passe bien dans chaque branche. Or, clairement, cette fonction possède une branche morte. En effet, si la valeur est supérieure à 15, elle est également supérieure à 10. Par conséquent, la dernière partie de la fonction ne peut jamais être atteinte.

```
>>> choix(16)
6
>>> choix(16) == int((16-10)/2)
False
```

Un test unitaire, en plus de vérifier la conformité des résultats, doit aussi détecter les branches mortes du code.

© Editions ENI - All rights reserved

1.1.3 Couverture

Tout perfectionniste rêverait d'avoir une couverture universelle de tout son code, c'est-à-dire que chaque ligne de code soit vérifiée par un test unitaire.
En réalité, ceci n'est pas être perfectionniste, c'est être inefficace. Un test unitaire doit englober une fonctionnalité toute entière, et pas une ligne de code ou un ensemble de lignes de code.
Les tests doivent être conçus pour chaque fonctionnalité de manière à couvrir les cas d'utilisation testés. En clair, celui qui développe de tels tests doit simplement prévoir chaque exception qui pourrait se produire et créer un et un seul test pour le mettre en évidence. Il faut donc passer tous les tests unitaires et ne pas s'arrêter au premier échec afin d'avoir un état des lieux le plus complet possible.
En regard à la gestion de projet, il y a plusieurs politiques. La « vieille école » développe l'application, puis construit ses tests lors de la phase d'intégration. Ceux-ci sont centrés uniquement sur les fonctionnalités les plus exposées et plus fouillés par les sections critiques de l'application de manière à fournir un effort minimal pour un résultat allant à l'essentiel. La « nouvelle école » ou la « programmation dirigée par les tests » veut que le test soit rédigé avant la fonctionnalité pour que le développement de cette dernière s'appuie dessus.
Ceci ne doit pas être vu comme une perte de temps, mais comme un investissement sur une garantie forte et permet en plus au développeur d'affiner sa vision sur l'ensemble de la tâche qui l'attend en lui faisant mettre au point des cas d'utilisation simples en même temps que la fonctionnalité.
Par contre, passer du temps à réfléchir pour écrire une fonctionnalité, l'écrire, puis quelques semaines ou mois plus tard, devoir se replonger dans le cahier des charges pour sortir le test associé est une perte de temps.
Le plus difficile dans le développement n'est pas d'écrire du code, c'est de le concevoir, de créer le cheminement mental qui permet de trouver des solutions adaptées. Devoir répéter ce travail est consommateur de temps et rend les tests unitaires inutilement coûteux.
Tout ceci fait que la programmation dirigée par les tests se démocratise dans les pôles de développement modernes, permettant une couverture plus importante et une meilleure fiabilité du test sans trop augmenter les coûts.

1.1.4 Outils

La fonction **`test_carre`** exposée précédemment permet de montrer l'idée générale d'un test unitaire. Il faut retenir qu'un test unitaire est technique, qu'il englobe une fonctionnalité, qu'il doit être développé préalablement voire au même moment que la fonctionnalité qu'il teste et qu'il doit dresser un état des lieux précis, factuel et reproductible et doit s'exécuter de bout en bout sans intervention humaine.
En réalité, le test unitaire est un véritable enjeu à plusieurs niveaux. Il est un garant de la qualité et du bon fonctionnement d'une application ce qui en fait un enjeu opérationnel, mais c'est également un enjeu stratégique puisqu'une mauvaise maîtrise de ces aspects peut avoir des effets négatifs sur la confiance accordée au logiciel par ses clients, voire entraîner l'abandon d'un développement de la part d'une MOA. Si le test unitaire par lui-même n'assure pas la qualité par sa simple présence, elle permet au moins d'avoir des réponses sur les causes des problèmes lorsqu'ils se produisent et facilite l'intervention corrective, raccourcissant les délais de correction.

En ce sens, il est logique qu'un langage dispose d'outils professionnels réalisant cette tâche selon des règles précises et formatées.
C'est le rôle du module unittest. Voici la fonction **test_carre** refaite :

```
>>> import unittest
>>> class Testeur(unittest.TestCase):
...     testValues = {2: 4, 0: 0, -2:4}
...     def testCarre(self):
...         for v, r in self.testValues.items():
...             self.assertEqual(carre(v), r)
...
>>> unittest.main()
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Il faut créer une classe pour chaque test unitaire et utiliser la fonction **main**.
Une classe héritant de **TestCase** doit encapsuler toutes les méthodes nécessaires permettant de tester une et une seule fonctionnalité.
Lorsqu'il y a plusieurs fonctionnalités, il faut créer autant de classes de test.
Chaque classe de test peut contenir plusieurs méthodes pour tester le bon fonctionnement. Chacune commence par les quatre lettres « **test** » et est écrite en camelCase.
Elles utilisent les méthodes commençant par « **assert** » :

```
>>> [m for m in dir(unittest.TestCase) if m.startswith('assert')]
['assertAlmostEqual', 'assertAlmostEquals', 'assertCountEqual',
'assertDictContainsSubset', 'assertDictEqual', 'assertEqual',
'assertEquals', 'assertFalse', 'assertGreater', 'assertGreaterEqual',
'assertIn', 'assertIs', 'assertIsInstance', 'assertIsNone', 'assertIsNot',
'assertIsNotNone', 'assertLess', 'assertLessEqual', 'assertListEqual',
'assertMultiLineEqual', 'assertNotAlmostEqual', 'assertNotAlmostEquals',
'assertNotEqual', 'assertNotEquals', 'assertNotIn', 'assertNotIsInstance',
'assertNotRegex', 'assertRaises', 'assertRaisesRegex',
'assertRaisesRegexp', 'assertRegex', 'assertRegexpMatches',
'assertSameElements', 'assertSequenceEqual', 'assertSetEqual',
'assertTrue', 'assertTupleEqual', 'assertWarns', 'assertWarnsRegex',
'assert_']
```

Leur nom est assez explicite pour comprendre à quoi elles servent et la documentation du module complète ces règles de nommage :

```
>>> help(unittest.TestCase.assertCountEqual)
Help on function assertCountEqual in module unittest.case:

assertCountEqual(self, first, second, msg=None)
    An unordered sequence comparison asserting that the same
elements, regardless of order.  If the same element occurs more
than once, it verifies that the elements occur the same number of times.
```

© Editions ENI - All rights reserved

```
            self.assertEqual(Counter(list(first)),
                             Counter(list(second)))
        Example:
            - [0, 1, 1] and [1, 0, 1] compare equal.
            - [0, 0, 1] and [0, 1] compare unequal.
```

Voici un autre exemple :

```
>>> help(unittest.TestCase.assertSequenceEqual)
Help on function assertSequenceEqual in module unittest.case:

assertSequenceEqual(self, seq1, seq2, msg=None, seq_type=None)
    An equality assertion for ordered sequences (like lists and
tuples). For the purposes of this function, a valid ordered
sequence type is one which can be indexed, has a length, and has
an equality operator.
    Args:
        seq1: The first sequence to compare.
        seq2: The second sequence to compare.
        seq_type: The expected datatype of the sequences, or None
if no datatype should be enforced.
        msg: Optional message to use on failure instead of a list
of differences.
```

1.1.5 Autres outils

On notera que Python 3.3 introduit un nouveau type d'objets Mock (un rétroportage est disponible pour les versions antérieures (https://pypi.python.org/pypi/mock).
Ces objets sont faits pour qu'on puisse en faire n'importe quoi et que cela ne plante jamais. Ceci permet de largement simplifier les tests unitaires en ne testant que la partie que l'on souhaite et uniquement celle-ci.
La documentation officielle donne les informations minimales (https://docs.python.org/3/library/unittest.mock.html).
On proposera aussi de regarder du côté de Factory Boy (https://factoryboy.readthedocs.org/en/latest/), qui permet de créer des outils de plus haut niveau pour se faciliter l'automatisation des tâches (voir la section Examples de la documentation).

1.2 Tests de non-régression

1.2.1 Actions de développement

Des tests unitaires doivent être exécutés régulièrement pour être efficaces.
Il se peut qu'un test se mette à ne plus fonctionner. Cela permet de révéler une erreur qui a été commise par un rajout de fonctionnalité.

Deux cas sont à prendre en compte. Soit le test n'est plus vrai, parce que la fonctionnalité a changé, mais ce changement n'a pas été reporté par les tests (tel comportement renvoyait une exception de « division par zéro », maintenant il renvoie une erreur de type « argument non valide »), il faut donc modifier le test unitaire, soit cela met en évidence une régression qu'il faut alors corriger.
Le test unitaire a alors permis de prendre les devants et de corriger avant que la MOA découvre le problème et que cela fasse baisser la cote de confiance de la MOE.
D'autre part, lorsqu'une opération importante de refactoring est menée, le fait de pouvoir exécuter les tests avec succès est un gage de réussite qu'il faut moduler avec la couverture réelle de ces tests.

1.2.2 Gestion de la découverte d'une anomalie par une MOA

À chaque création, il faut rajouter le ou les tests qui vont avec. À chaque modification, il faut modifier ou adapter les tests si nécessaire. Lorsqu'une anomalie est découverte, ceci est la preuve d'un manquement dans la couverture fonctionnelle. Cela n'est pas une catastrophe, loin de là, et c'est le lot commun à toute application.
Par contre, si l'on peut accepter qu'un bug se produise une fois, il n'est plus acceptable qu'il se reproduise une deuxième fois.
Il y a donc plusieurs actions à mener et parmi celles-ci, apporter une correction n'est pas la plus importante :

- La première action consiste à reproduire l'anomalie :
  - Être capable de la constater.
  - Être capable de produire l'enchaînement d'action qui l'a fait apparaître.
- La seconde action consiste à isoler l'anomalie :
  - Trouver toutes les manières de reproduire l'anomalie.
  - Comprendre sa cause.
  - Trouver le bon niveau où agir (est-ce la fonction qui n'est pas bonne ou les paramètres passés lors de son appel, ou encore un composant connexe qui aurait dysfonctionné, entraînant un effet de bord ?).
- La troisième action est de produire un test unitaire :
  - Il doit mettre en évidence l'anomalie.
  - Il doit ne pas passer, puisqu'il n'a pas été corrigé.
  - Il peut aussi mettre en évidence des cas qui fonctionnaient avant et qui pourraient ne plus fonctionner après la correction.
- Enfin, la correction peut être apportée :
  - La reproduction manuelle du cas d'utilisation ayant entraîné l'anomalie ne l'entraîne plus.
  - La reproduction des autres cas découverts et testés manuellement n'entraîne plus d'anomalies.
  - La reproduction manuelle de quelques cas qui fonctionnaient fonctionne toujours.
  - Et enfin, tous les tests de la suite passent, y compris ceux mettant en évidence l'anomalie elle-même.

© Editions ENI - All rights reserved

Si, quelques mois après, la fonction est retouchée et redevient erronée, le passage des tests unitaires met en évidence le problème, qui pourra alors être corrigé avant sa mise en test et en production et donc avant qu'il soit redécouvert par la MOA.
Le test de non-régression est donc très proche du test unitaire, mais il ne trouve sa place dans un plan d'assurance qualité que s'il est accompagné d'une bonne base de tests unitaires, car sur un plan purement technique, il s'agit de la même chose.

1.3 Tests fonctionnels

Les tests fonctionnels permettent de tester une application dans son ensemble et leur but est de déterminer si le logiciel est conforme au cahier des charges. Ces mots sont importants, parce que conforme au cahier des charges ne veut pas forcément dire conforme aux attentes du client, parce que rien ne dit que le cahier des charges ait été parfaitement bien réalisé et soit réellement conforme, mais il reste la base de travail de la MOE.
Il y a donc deux risques qui sont que la MOA ait mal précisé ses attentes ou que la MOE les ait mal comprises ou résolues. Aucun de ces risques n'est pris en compte ici, puisque la base de travail du testeur fonctionnel reste le cahier des charges et que lui-même en réalise sa propre interprétation.
Ces tests peuvent être réalisés manuellement par l'exécution de scénarios de test préconçus ou de manière automatique en utilisant un logiciel pour réaliser des opérations répétitives. Sélénium ou funkload sont par exemple des solutions.
Idéalement, ils ne sont exécutés que lorsque les tests unitaires l'ont été.
Dans tous les cas, ces tests peuvent faire ressortir :

- Des erreurs techniques :
  - Il conviendra de poser des tests de non-régression.
  - Ce sont les erreurs à traiter en priorité.
  - Ils doivent faire l'objet d'un rapport circonstancié pour permettre leur traitement.
- Des erreurs fonctionnelles :
  - Il convient de la caractériser pour savoir comment la traiter.
  - Il convient de se référer au cahier des charges pour savoir si l'erreur en est réellement une et l'amender pour apporter les précisions qui manquaient.
  - Si l'erreur en est réellement une, elle doit être corrigée après acceptation des amendements sur le cahier des charges.
  - Si l'erreur peut faire l'objet d'un test unitaire et si c'est sa place, il peut être réalisé.
- Des erreurs cosmétiques :
  - Sur une GUI, des éléments non correctement placés ou interagissant mal entre eux.
  - Sur une interface web, des problèmes de CSS ou autre.
  - Ce genre d'erreur doit être traité au cas par cas, mais ne peut faire l'objet de tests unitaires, elle doit donc être particulièrement documentée.

Les documents caractérisant une erreur fonctionnelle sont les différents documents réalisables par UML et qui permettent de définir les relations entre objets, les étapes de chaque fonctionnalité, l'enchaînement des écrans…

Des tests fonctionnels complets ne peuvent se passer d'automates pour gérer des cas toujours plus complexes et mis à jour à chaque montée fonctionnelle du logiciel, mais ils ne peuvent pas non plus se passer de manipulations manuelles qui permettent à l'œil humain de détecter tout ce que les automates ne peuvent encore faire.

Lorsque le test fonctionnel est réussi, il peut être complété par un test d'acceptation qui consiste à laisser la MOA – qui connaît le fonctionnel métier et les attentes liées au logiciel – prendre le logiciel en main et s'en servir à la manière dont vont s'en servir les utilisateurs finaux. Ainsi, on s'assure que les attentes sont vraiment réalisées.

1.4 Tests de performance

Dès lors que l'on parle de tests de performance, il vient immédiatement à l'esprit les tests de charge. Et effectivement, ceux-ci font partie intégrante des tests de performance, mais ils ne sont pas le seul critère.

Les tests de charge ont pour but de démontrer que l'application est capable de répondre dans des temps acceptables à une certaine charge représentée par un certain nombre d'actions effectuées en des temps déterminés par un certain nombre d'utilisateurs.

La performance attendue est donc définie par le temps de réponse moyen et la charge est définie par le nombre d'utilisateurs simultanés et la quantité d'actions qu'ils mènent. Ces données doivent être prises en compte pour le développement et donc précisées avant que celui-ci commence, puisqu'elles ont un impact sur les choix architecturaux qui sont essentiels.

L'idée d'un test de charge réussi est de simuler des conditions réelles qui soient une prévision haute de la réalité. Inutile de tester cent mille utilisateurs simultanés sur un intranet pour une PME.

Si l'on prend un exemple d'un logiciel qui permette une imputation des temps et que la politique de l'entreprise est de demander cette saisie à chaque fin de semaine, on peut légitimement tester le cas où tout le personnel de l'entreprise déciderait d'imputer le vendredi entre 17 heures et 18 heures, par exemple.

Un test de charge consisterait à malmener le module correspondant pour simuler un scénario de saisie des temps pour un seul utilisateur. Ce scénario durerait autant que la durée réelle nécessaire à un humain pour faire la même opération et serait répété autant de fois que nécessaire. Au début avec une seule personne, qui se connecte par minute, puis 5, puis 10, puis 50, puis en redescendant jusqu'à ce qu'au bout d'une heure, il n'y ait plus de nouvelle connexion.

Un système qui ne s'engorge pas est déjà un bon système. Si en plus il a des temps de réponse adéquats, le test est alors dit réussi.

Mais le test de charge n'est pas tout. Il y a aussi le test de performance brute. Ce test consiste à faire utiliser une partie de l'application par plusieurs processus qui ne sont pas là pour simuler un fonctionnement en temps humain, mais pour aller le plus vite possible. Dès qu'une réponse est reçue par un des processus, il renvoie immédiatement une requête. Il est ainsi possible d'avoir des informations très localisées sur la performance brute d'une seule fonctionnalité. Cela est utilisé dans les applications web par exemple avec **`siege`** réalisant cela très simplement (http://www.joedog.org/index/siege-manual#a02), mais il existe aussi **`Pylot`** qui est un peu plus complet et fait plus que cela, il est à la croisée de plusieurs exigences.

© Editions ENI - All rights reserved

Aujourd'hui, en termes de performances, on regarde surtout du côté de la rapidité d'exécution, mais à performance égale ou similaire, regarder du côté de la consommation mémoire est également une bonne idée.

Même si les machines sont très puissantes, la trop grande gourmandise de certains logiciels oblige à utiliser la swap, entraînant alors des régressions en termes de rapidité hallucinantes. La solution alors communément adoptée est d'utiliser plusieurs serveurs et à démultiplier les besoins en architecture, et donc les coûts. Ce sont ces coûts qui peuvent être évités en Python qui font que ce dernier peut se démarquer, car s'il n'est pas toujours le plus rapide, son ratio temps/mémoire consommé est plutôt bon.

À ce niveau, le langage ne fait pas tout, puisque la manière dont le développeur développe a une influence considérable, l'exemple typique étant le choix entre recalculer une valeur à chaque appel ou la calculer une fois et la stocker pour les appels suivants.

Le choix doit être pesé en termes d'avantages et inconvénients.

Si le test de charge est réaliste par rapport à une prévision haute et simule un cas d'utilisation particulier de l'application, la simulation de tous les tests de charge de manière simultanée, sur plusieurs heures, en insistant tantôt sur un scénario, tantôt sur un autre permet de prévoir le comportement de l'application lors d'une très importante sollicitation et ainsi de voir quelles parties vont être limitatives.

C'est ce que l'on appelle un test de stress. Cela met en évidence la manière dont l'application peut se stabiliser ou tomber. Il peut éventuellement être rajouté au cours du test des défaillances système, l'arrêt d'un des serveurs d'une grappe ou tout autre incident qui pourrait se produire en production. Cela permet de voir comment l'application les gère, reprend la main, et de quelle manière l'équipe système peut relancer le processus, si nécessaire, sans trop de dommages.

Un autre type de test consiste à effectuer l'un des tests de charge en montant progressivement et sans limites le nombre d'utilisateurs simultanés jusqu'à ce que le système tombe. Cela permet de savoir pour chaque module représenté par un test de charge quelle est la charge limite. Il s'agit d'un test de montée en charge, à ne pas confondre avec le test de charge, qui lui est réaliste.

Enfin, le test de robustesse permet de simuler une activité moyenne très supérieure à l'activité moyenne prévue (sans forcément être au-delà des pics de charge prévus, il s'agit d'une activité moyenne). Il est alors testé les temps de réponse, mais aussi la réponse elle-même afin de vérifier qu'une charge trop importante n'entraîne pas une perte de données, un mauvais fonctionnement ou autre.

Prenons pour exemple une application intranet subissant vingt mille connexions par jour en moyenne à travers tout le site, dix mille le vendredi lors du pic de charge la saisie des temps, dix mille à nouveau en début de mois pour la saisie de rapports d'activités. On sait également qu'il peut y avoir des pics de charge sur la page d'accueil dû à la présence d'une actualité.

La première étape est de passer tous les tests unitaires et de les réussir, étape indispensable avant même d'aller plus loin.

La seconde étape est de passer les scénarios fonctionnels unitairement à l'aide d'un automate et de s'assurer de leur bonne réalisation.

Seulement par la suite, il est utile de s'intéresser à la performance, car quelle que soit l'application, une gestion des exigences cohérente veut que l'exigence de bon fonctionnement soit toujours supérieure à l'exigence de performance.

On écrira donc un test de charge sur le module de saisie des temps, un autre sur le module de saisie des rapports d'activités. Deux tests de charge seront donc effectués pour vérifier que l'ensemble des saisies de tout le personnel puisse se faire dans un temps beaucoup plus court que ce qu'il sera dans la réalité et de même pour la saisie des rapports. On réalisera également des tests de montée de charge pour vérifier à quel moment le système ne peut plus assumer de nouvelles connexions.

Des tests de performance brute seront effectués sur la page d'accueil et sur les fonctionnalités identifiées comme étant unitairement longues (grosses requêtes...).

On exécutera un test de robustesse pour simuler quarante mille connexions par jour, soit le double de la moyenne prévue et un test de stress pour savoir à combien de connexions l'on peut monter.

L'ensemble de ces résultats permettra de définir les points faibles de l'application, au niveau de la performance et permettra au chef de projet de diriger les actions à mener, conformément aux exigences et aux priorités.

1.5 Tests syntaxiques

Des outils de qualimétrie comme Pylint (http://pypi.python.org/pypi/pylint) ou PyChecker (http://pychecker.sourceforge.net/), ou encore flake8 (https://pypi.python.org/pypi/flake8, https://flake8.readthedocs.org/en/2.0/) peuvent se greffer sur des outils et signaler des problèmes éventuels qui auraient pu échapper au développeur. Ils sont un complément parfois indispensable. De la même manière, lancer le programme en ligne de commande en utilisant -Wdefault permet d'avoir un avertissement ResourceWarning si une ressource est mal fermée (fichier, socket, connexion FTP...) y compris si le ramasse-miettes fait le travail.

On peut aussi citer **mypy** qui permet d'exécuter des codes statiques en se basant sur les types hints. Il s'installe ainsi :

```
$ pip install mypy
```

Et se lance ainsi, sur un fichier en particulier ou sur un répertoire ou encore un module (fichier python) ou un package (répertoire contenant **__init__.py**) :

```
$ mypy file.py
$ mypy directory
$ mypy file1.py file2.py directory1, directory2
$ mypy -m my_package.my_subpackage
$ mypy -p my_package
```

Il existe également **prospector** qui permet d'utiliser tous ces outils ensemble et de faire un rapport général. Il s'installe ainsi :

```
pip install prospector[with_everything]
```

© Editions ENI - All rights reserved

Il installe les outils suivants :
- astroid
- coverage
- coveralls
- docutils
- dodgy
- mccabe
- mock
- mypy
- nose
- pep8-naming
- pycodestyle
- pyflakes
- pylint
- pyroma
- tox
- vulture

Il est vivement recommandé d'aller regarder individuellement chacun de ces outils et de se les approprier, car si mettre en place une procédure qualité est toujours assez long et complexe, le retour sur investissement est généralement plutôt rapide et cette démarche est ensuite assez aisée à étendre à d'autres projets une fois qu'elle est maîtrisée.

1.6 Intégration continue

L'idée de l'intégration continue est multiple. En premier lieu, tout projet doit utiliser un système de gestion de versions performant et de préférence moderne (que ce soit GIT ou Mercurial, par exemple, CSV et SVN n'étant pas capables de répondre aux exigences modernes).

Le principe de l'intégration continue est de passer les tests vus précédemment régulièrement (comprendre tous les soirs) de manière à savoir où l'on en est chaque matin.

Ainsi, si un test unitaire ne passe plus, la réponse est forcément dans un des commits de la veille, ce qui diminue drastiquement les efforts à fournir pour retrouver cette anomalie. Il peut alors être corrigé immédiatement et ne pas entraîner d'autres problèmes ou pertes de temps.

Si un test de performance a régressé, il est aisé de produire les mêmes types de déduction et de se pencher sur le problème. De la même manière, si des efforts ont été faits la veille pour améliorer les performances d'une des fonctionnalités, le résultat est visible le lendemain matin. Lorsque l'on cherche à améliorer les performances générales de l'application, on sait également déterminer à quel endroit l'effort est le plus utilement investi.

On peut citer un outil comme GitLab qui est facilement installable sur un serveur en allant tout simplement chercher une image Docker. Il est ensuite possible de monter ses projets git et de créer des tâches automatisées qui se déclencheront après avoir poussé son code ou après avoir fait une requête de merge de code.

Il est possible par exemple de monter des tests unitaires, de syntaxe, de respect des PEP, mais aussi de reconstruire la documentation, voire de créer des installateurs automatiquement.
C'est également un excellent outil pour faciliter le travail collaboratif et en particulier la revue de code.
L'intégration continue permet également d'effectuer des statistiques sur chacune de ces données et montrer l'évolution de la qualité du logiciel dans le temps, mais aussi la réactivité de l'équipe de développement lors de la découverte d'un problème, ce qui sont des indicateurs importants à faire valoir pour montrer la compétence d'une équipe de développement et pour communiquer autour du logiciel. Disposer d'une plateforme d'intégration opérationnelle et la mettre au cœur de sa stratégie de développement quelle que soit l'application permet aux équipes de se forger une cohérence d'ensemble sans pour autant rajouter des contraintes supplémentaires de développement.
Rien n'interdit de mettre son application sous contrôle de la plateforme d'intégration sans réaliser de tests unitaires, de scénarios fonctionnels, ou de scénarios de tests de charge. L'application bénéficie alors du minimum syndical qui est l'utilisation des outils de qualimétrie et permettant également de centraliser les processus.
D'autre part, rien de tout ce que l'on vient de voir n'est utile si l'on n'a pas de système de rapports d'erreurs et une organisation permettant de déléguer le travail entre les rapporteurs, les développeurs et les mainteneurs d'application. De la même manière, un outil permettant de centraliser la documentation et tous les livrables produits est indispensable.
À noter qu'une intégration de flake8 à Gedit est proposée, elle est également un exemple d'utilisation de cet outil et de mise en œuvre d'un plug-in pour Gtk (http://git.inspyration.org/?p=gedit/flake8;a=tree).

2. Programmation dirigée par la documentation

2.1 Documentation interne

2.1.1 À destination des développeurs

Écrire une documentation à l'intérieur même du code est le meilleur moyen de fournir les explications au lecteur du code.
Cela facilite donc la prise en main du code pour toute personne qui voudrait s'y investir, et dans un contexte de logiciel libre, par exemple, c'est indispensable. Le nombre de logiciels libres qui souffrent d'une communauté trop restreinte uniquement parce que le code est mal documenté est trop important et nuit gravement à la pérennité du logiciel libre en question.
Cela nous sert également à reprendre son propre code après être passé sur d'autres projets entre-temps.
Une des particularités de Python est de créer des tests unitaires au sein de la documentation (http://docs.python.org/library/doctest.html). Le procédé est simpliste, permet au lecteur de voir une mise en situation et peut servir de test unitaire minimaliste.
Ainsi, en Python, il n'y a pas de commentaire de code, mais de vraies documentations qui permettent d'appréhender le code dès lors qu'elles sont complètes et bien réalisées.

© Editions ENI - All rights reserved

2.1.2 À destination des utilisateurs

Aider le développeur à se plonger dans le code de son application ou de sa bibliothèque est loin d'être la seule motivation de ces documentations. En effet, Python propose des outils absolument magnifiques.
D'abord, il y a **pydoc** qui offre depuis python 3.2 une meilleure présentation, c'est un outil de recherche rapide et dont l'option **-b** ouvre dans un navigateur :

```
$ pydoc module_test
```

Cela permet de visualiser la documentation de la même manière que si l'on avait ouvert une console et utilisé la commande help sur le module.
Mais une autre commande permet de générer une documentation HTML :

```
$ pydoc -w module_test
```

Le résultat est désuet, mais efficace. Pour produire quelque chose visuellement plus attrayant, il y a **epydoc** :

```
$ epydoc --html module_test -o path
```

Il s'installe ainsi :

```
$ sudo aptitude install python-epydoc
```

Le tout fait que la documentation peut être publiée sans autre effort de rédaction.

2.2 Documentation externe

2.2.1 Présentation

Si l'on a vu que des documentations techniques pouvaient être générées aisément à partir du documentaire du code, Python permet d'éditer une documentation plus complète, plus orientée fonctionnel.
L'utilitaire de référence est Sphinx et il s'installe ainsi :

```
$ aptitude install python-sphinx
```

La documentation officielle de Python est générée en utilisant Sphinx, tout comme beaucoup de projets Python ou autres (http://www.sphinx-doc.org/en/master/examples.html).
Le format utilisé est reStructuredText auquel sont rajoutées des instructions supplémentaires. L'utilisation de cet outil nécessite donc des connaissances dans cette technologie et un tutoriel est disponible sur le site (http://www.sphinx-doc.org/en/master/usage/restructuredtext/basics.html).
Ce qui est présenté ici est un complément qui permet de mettre le pied à l'étrier pour débuter un projet Sphinx.

2.2.2 Démarrage rapide

Il s'agit d'un outil très complet qui met la documentation au cœur du processus de développement logiciel en créant des projets de documentation :

```
$ sphinx-quickstart
Welcome to the Sphinx 1.0.1 quickstart utility.[...]
```

Cette commande nécessite de répondre à un grand nombre de questions et comme cela se pratique usuellement (avec aptitude, par exemple) l'utilisateur est libre de répondre s'il n'y a pas d'indication, peut avoir une liste de choix présentée entre parenthèses et il peut y avoir un choix par défaut présenté entre crochets.
La procédure de démarrage rapide est très bien détaillée par quelques phrases expliquant la question posée.
Voici les étapes essentielles qui sont très bien documentées question après question :
- Il faut donner le chemin dans lequel on veut mettre le projet de documentation :

```
> Root path for the documentation [.]: projet/doc
```

- Il est possible (et recommandé) de séparer le code source (du projet de documentation) et le résultat produit, et cela se configure :

```
> Separate source and build directories (y/N) [n]: y
```

Un préfixe spécial peut être utilisé pour les templates et répertoires statiques qui peuvent être partagés entre différents projets :

```
> Name prefix for templates and static dir [_]:
```

- Il faut indiquer le nom du projet dont on réalise la documentation :

```
> Project name: projet
```

- Le nom de l'auteur du projet :

```
> Author name(s): sch
```

- La version du projet à laquelle se réfère la documentation :

```
> Project version: 0.1
```

- Le numéro de release uniquement si l'on utilise cette notion :

```
> Project release [0.1]:
```

- Le suffixe des fichiers sources peut être choisi librement, le contenu sera du reStructuredText (ceci a une importance dans le sens ou seuls les fichiers portant cette extension sont considérés comme des fichiers sources) :

```
> Source file suffix [.rst]:
```

- Le document principal, celui qui permet l'accès à la documentation peut porter un nom personnalisé :

```
> Name of your master document (without suffix) [index]:
```

- EPUB est un format de livre électronique et peut être également utilisé (en plus de HTML, PDF, LaTeX ...) :

```
> Do you want to use the epub builder (y/N) [n]: y
```

- La documentation du code peut être ou ne pas être intégrée à celle réalisée par Sphinx :

```
> autodoc: automatically insert docstrings from modules (y/N) [n]: y
```

© Editions ENI - All rights reserved

– Les éventuels doctest ou exemples de code écrits dans les docstring font l'objet du même choix :

```
> doctest: automatically test code snippets in doctest blocks (y/N) [n]: y
```

– Des liens peuvent être réalisés avec d'autres documentations. Ainsi, si une application est composée de plusieurs projets (bibliothèques, modules...), chaque projet peut avoir sa documentation sous forme de projet Sphinx et le projet de l'application elle-même peut avoir son propre projet de documentation qui intègre ceux des projets dépendants. Voici comment autoriser ces liens :

```
> intersphinx: link between Sphinx documentation of different
projects (y/N) [n]:
```

– Il est possible de choisir d'intégrer les TODO dans le projet, mais au moment de la génération du document le choix est donné de les faire apparaître ou non :

```
> todo: write "todo" entries that can be shown or hidden on build
(y/N) [n]:
```

– Il est possible d'avoir des informations sur la couverture de la documentation :

```
> coverage: checks for documentation coverage (y/N) [n]: y
```

– Le rendu des formules mathématiques peut être réalisé par une librairie travaillant sur PNG :

```
> pngmath: include math, rendered as PNG images (y/N) [n]: y
```

ou en JavaScript, côté client :

```
> jsmath: include math, rendered in the browser by JSMath (y/N) [n]:
```

– Il est prévu de conditionner le contenu rendu de la documentation par rapport à un fichier de configuration. Cela permet de générer plusieurs documentations avec plus ou moins de contenu ou du contenu différent et de gérer ces différences par un fichier de configuration par documentation à générer.

```
> ifconfig: conditional inclusion of content based on config
values (y/N) [n]: y
```

– Le code source peut être inclus dans la documentation :

```
> viewcode: include links to the source code of documented Python
objects (y/N) [n]: y
```

– Pour le rendu du document final à partir des sources du projet de documentation, il est nécessaire d'utiliser une commande et pour cela, il faut utiliser un makefile pour Linux ou un fichier dédié à Windows :

```
> Create Makefile? (Y/n) [y]: y
> Create Windows command file? (Y/n) [y]: n
```

Une fois cette étape terminée, un répertoire est créé portant le nom du projet et contenant deux répertoires dont l'un rassemble les sources du projet de documentation et l'autre les documents générés.

Une fois cette étape créée, il est possible de configurer d'autres paramètres plus fins, puis d'écrire la documentation, les fichiers et de structurer le tout.

2.2.3 Résultat

La réalisation de ce genre de documentation nécessite de l'investissement pour monter en compétence sur le produit, mais cela vaut le coup parce que la qualité est au rendez-vous et les documents rendus sont à la hauteur des attentes.

Ces documents peuvent être hébergés sur un serveur, être mis en téléchargement et servir à fédérer une communauté autour d'un logiciel libre. Ils peuvent également être des livrables pour un client.

La présence de la documentation et la performance des outils permettant de la gérer sont également un indicateur de la qualité d'un logiciel et en ce sens, Sphinx devient indispensable à de gros projets, en particulier lorsqu'ils doivent utiliser d'autres modules qui peuvent être partagés avec d'autres projets.

3. Optimisation

3.1 Qualimétrie

La qualimétrie consiste à produire des indicateurs de qualité pondérables et à évoluer un processus de développement logiciel sur cette base. La pondération fixe les priorités à donner à chaque indicateur et par conséquent, doit guider les équipes MOE et MOA dans leurs choix. Deux évaluations avec des indicateurs différents mènent donc potentiellement à des résultats distincts.

Ces indicateurs couvrent l'ensemble du processus de développement, pas uniquement la qualité du code elle-même. Cela peut être la qualité des tests unitaires, fonctionnels, de performance liés au projet, mais également les processus de signalisation d'un bug, la solidité de la solution de gestion de version utilisée, la formation des équipes de développement et la cohérence du travail en groupe ou encore la qualité de la documentation.

Ceux-ci sont fortement liés à la politique de gestion des exigences du projet qui peut être inexistante à très détaillée et précise.

Il existe une norme qui est ISO 9126 qui hiérarchise ces indicateurs :

- Capacité fonctionnelle (regarde la conformité fonctionnelle) :
  - Aptitude
  - Exactitude
  - Interopérabilité
  - Sécurité
- Fiabilité (donne le taux de confiance que l'on peut avoir dans le logiciel) :
  - Maturité
  - Tolérance aux pannes
  - Possibilités de récupération
- Facilité d'utilisation (indique la complexité de prise en main du logiciel au regard de sa complexité fonctionnelle et son taux d'adhésion) :
  - Compréhensibilité

© Editions ENI - All rights reserved

  - Facilité d'apprentissage
  - Exploitabilité
  - Attractivité
- Efficacité (mesure les « performances » du logiciel) :
  - Efficacité en termes de temps
  - Efficacité en termes de ressources
- Maintenabilité (indique l'effort à fournir pour maintenir/corriger/adapter le logiciel en assurant sa continuité de fonctionnement) :
  - Facilité d'analyse
  - Facilité de modification
  - Stabilité
  - Testabilité
- Portabilité (donne des éléments sur la qualité de la relation entre le logiciel et son environnement) :
  - Adaptabilité
  - Facilité d'installation
  - Coexistence
  - Interchangeabilité

Ces indicateurs peuvent parfaitement être utilisés pour analyser et auditer une application, mais ils peuvent surtout être l'objet d'une réflexion dès le début du projet et être partie prenante du processus d'intégration continue.

Cela nécessite certes un effort important au début du premier projet, mais une fois les outils mis en place, le gain de qualité le compense largement. L'effort est moindre pour les projets suivants grâce à la capitalisation d'expérience et au fait qu'une fois que la plateforme d'intégration est montée, les processus bien analysés et pris en main, leur application devient plus aisée.

Pour tous ces paramètres, le choix du langage a une grande influence positive ou négative. Le choix d'utiliser un framework ou une bibliothèque particulière en a également une tout comme les choix d'architecture.

En ce sens, l'utilisation de Python a un avantage certain en ce qui concerne tous les aspects relatifs à la fiabilité, l'efficacité, la maintenabilité et la portabilité mais pour maximiser des indicateurs, il faut respecter le langage, utiliser les bons outils aux bons endroits et avoir une véritable stratégie de conception. Le langage seul ne réalise pas tout.

En ce qui concerne les aspects relatifs à la fonctionnalité et à l'utilisabilité, seuls la qualité de la relation entre MOA et MOE, la précision des étapes de conceptions préalable au développement et un développement rigoureux permettent d'avoir des résultats positifs.

Cependant, quels que soient ces aspects, Python propose un outillage qui réalise des mesures ou permet un bon fonctionnement. Par exemple, pour évaluer l'aptitude ou l'exactitude d'un logiciel, des tests unitaires et fonctionnels sont indispensables et il a été vu comment les mettre en place. Il en est de même pour l'efficacité et la maintenabilité.

Il est aussi nécessaire d'avoir un logiciel de suivi d'anomalies d'un système de gestion de versions performant et décentralisé (faisant partie de la plateforme d'intégration) et de réaliser un suivi des exigences.

3.2 Outils de débogage

Python propose un excellent outil de débogage (http://docs.python.org/py3k/library/pdb.html). Sa commande la plus utilisée est celle-ci :

```
import pdb; pdb.set_trace()
```

Insérée dans un programme, cette commande est un point de débogage. Elle stoppe son exécution et propose une invite de commande permettant d'interagir avec lui comme si l'on était en console.

En plus de la console, les lettres suivantes peuvent être utilisées :

- **c** permet de reprendre l'exécution du programme jusqu'au prochain point de débogage ou la fin du programme (par une exception ou une fin normale) ;
- **n** permet d'aller jusqu'à la prochaine ligne, quelle que soit la ligne courante ;
- **s** permet d'entrer dans la première fonction ou méthode appelée dans la la ligne courante (on change alors d'espace de nommage local) ou passe à la ligne suivante si ce n'est qu'une expression ;
- **r** permet de faire l'inverse de s, c'est-à-dire terminer la fonction ou méthode courante et reprendre le débogage après la fin de l'appel de fonction ;
- **j** permet d'aller directement à une ligne déterminée ;
- **w** permet d'écrire une trace quel que soit l'endroit où l'on se trouve pour déterminer de quelle manière naviguer ;
- **l** et **ll** permettent de lister du code autour de la ligne courante ;
- **a** permet de lister les arguments avec lesquels la fonction a été appelée ;
- **q** permet de quitter le débogueur et le programme ;
- **restart [args...]** recommence le débogage à l'endroit initial.

Pour certaines applications, en particulier les applications web, un plantage entraîne, en mode debug, un affichage d'une trace et d'une console par niveau de profondeur, permettant alors d'entreprendre les actions nécessaires pour l'isolation de ce bug.

Il existe également un module logging assez simple d'utilisation (http://docs.python.org/py3k/library/logging.html) permettant de générer des messages en les plaçant à plusieurs niveaux (debug, info, warning, error, critical).

Lorsqu'une application tourne, en fonction de l'environnement sur lequel elle s'exécute, il est possible de générer plus ou moins de messages.

Il est aussi conseillé de mettre en place une rotation de log dans le cas de messages nombreux et des outils permettant de parser les fichiers de logs et de réaliser des statistiques.

3.3 Outils de profilage

Python dispose d'un petit module très utile pour effectuer un test unitaire de performance. Il suffit alors de procéder comme suit :

```
>>> import timeit
>>> timeit.timeit('from itertools import
permutations;list(permutations(range(8)))', number=1)
0.011569023132324219
>>> timeit.timeit('from itertools import
permutations;list(permutations(range(8)))', number=10) / 10
```

© Editions ENI - All rights reserved

```
0.007745885848999023
```

Comme le montre l'exemple, l'exécution est isolée et il faut réaliser les imports au sein de la chaîne présentant la commande à tester. Mais il dispose aussi de quelques modules permettant d'opérer du profilage de manière plus avancée. Pour que ceux-ci soient pleinement fonctionnels, il faut installer :

```
$ sudo aptitude install python3-profiler
```

Ces instructions doivent alors fonctionner :

```
>>> import cProfile
>>> import pstats
>>> import profile
```

Voici comment tester rapidement une méthode :

```
>>> cProfile.run('list(permutations(range(8)))')
         3 function calls in 0.021 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.020    0.020    0.020    0.020 <string>:1(<module>)
        1    0.000    0.000    0.021    0.021 {built-in method exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
```

On a alors les mesures classiques :
- nombre d'appels ;
- temps total passé (dans la fonction et uniquement) ;
- temps passé par appel ;
- temps cumulé (dans la fonction et les sous-fonctions) ;
- temps cumulé par appel.

Ces solutions sont utilisables également dans un terminal et il est possible de réaliser des tests de profiling dans un environnement d'intégration.
Ces tests permettent de repérer une fonction anormalement lente ou de discerner celles qui sont très souvent utilisées et par conséquent, celles sur lesquelles une optimisation est rentabilisée au maximum.

3.4 Règles d'optimisation

3.4.1 Pourquoi optimiser ?

Une fois que l'on a défini les indicateurs qualités, optimiser une application revient à mettre en œuvre une action permettant d'améliorer un ou plusieurs de ces indicateurs.

Une bonne optimisation est une optimisation qui fait monter significativement la notation qualité globale.

Si le **fonctionnel** est particulièrement important et peut être la seule chose qui compte pour une MOA, le fait d'avoir des tares dans les autres domaines peut entraîner un surcoût important pour l'ajout de toute nouvelle fonctionnalité.

Dans ce sens, un surcoût important peut également être généré lorsque la **maintenabilité** n'est pas assurée. Il est donc important que la MOE mette en œuvre des moyens pour s'assurer un niveau minimal et cela passe également par une documentation efficace du code, surtout si les équipes de développement peuvent changer.

D'autre part, l'**efficacité** est souvent pour une MOE le point d'optimisation essentiel vers lequel porte tous les efforts, puisque c'est le critère technique qui est le plus mesurable et qui est parfois dans les esprits le critère essentiel. Or, s'il convient d'assurer un niveau d'efficacité compatible avec la charge prévue, il n'est guère indispensable d'aller au-delà et de passer trop de temps sur ce point.

Un autre axe essentiel est, pour les utilisateurs autres que la MOA, c'est-à-dire ceux qui devront utiliser le logiciel sans avoir participé à sa conception, l'**utilisabilité**. Un logiciel qui présente une navigation difficile, une présentation des données peu lisible, un nombre d'opérations trop important pour effectuer une saisie de données ou des formulaires trop complexes peut entraîner le rejet de l'application.

Un axe sur lequel se rejoignent MOE, MOA et les utilisateurs est la **fiabilité**. En effet, une application qui connaît trop de pannes ou qui perd des données peut être facilement rejetée par ses utilisateurs. Lorsque cela se produit, les résistances à l'utilisation de l'application et à ses changements deviennent plus importantes.

Enfin, le dernier axe est assez spécifique au type de logiciel. La **portabilité** peut signifier pour un logiciel installable par l'utilisateur sur une machine le fait d'avoir un installateur de bonne qualité et relativement simple, cohérent avec les principes usuels inhérents aux systèmes d'exploitation (paquet RPM ou DEB pour linux, installateur autonome pour Windows). Cela peut signifier l'utilisation du logiciel sur différents supports comme un téléphone mobile par exemple (chacun ayant ses propres systèmes), cela peut également vouloir dire pour un logiciel appliquant des normes, leur respect de manière à pouvoir remplacer des briques logicielles par d'autres briques réalisant et présentant les mêmes interfaces. Pour un site intranet, cela peut vouloir dire qu'il faut prévoir l'intégration du logiciel avec les autres et des procédures adaptées pour l'installation et la mise à jour.

En ce sens, un équilibre entre les six branches est essentiel, de la même manière que celui entre MOA et MOE. C'est donc en réponse à ces règles et à ces indicateurs que doivent être fixées les priorités d'optimisation.

© Editions ENI - All rights reserved

3.4.2 Règles générales

La première étape dans la conception d'un logiciel est de déterminer avec précision son périmètre fonctionnel ainsi que son périmètre technique afin de choisir les bibliothèques à utiliser.

Bien entendu, le choix de bibliothèques performantes (fonctionnellement complètes, fiables, faciles à utiliser, efficaces, maintenables et portables) influe sur l'application finale.

Ensuite, il faut fixer les exigences, déterminer la volumétrie de données à traiter, les temps de réponse attendus, l'évolutivité attendue du logiciel afin de déterminer l'architecture logicielle à mettre en place (en changer pour une application existante est difficile, nécessite une réécriture, mais peut être largement amorti, il faut réaliser un audit pour évaluer la pertinence d'un tel changement).

Puis il faut fixer les indicateurs qualité et les associer aux exigences.

Le projet doit alors débuter par l'utilisation d'une plateforme d'intégration, la mise en place d'outils permettant d'avoir un regard juste sur la qualité du travail et la réalisation d'un code clair, respectant des contraintes de lisibilité et de qualité et qui soit associé à de la documentation.

L'écriture du code nécessite l'utilisation de motifs de conception dont les choix permettent de structurer le code. De plus, des objets plus ou moins rapides et plus ou moins consommateurs de ressources sont utilisés. La règle est de toujours chercher à utiliser le bon objet pour la bonne fonctionnalité. Par exemple, la liste est trop souvent utilisée alors que dans certains cas le n-uplet serait plus adapté et dans d'autres, il s'agirait d'un ensemble. La maîtrise des types de base est donc un enjeu essentiel. Dans la même ligne, les algorithmes écrits sont plus ou moins rapides et consommateurs de ressources. Python 3.x a largement généralisé les générateurs, les itérateurs et offre beaucoup de solutions. Leur maîtrise est également un enjeu.

En ce sens, l'élégance de Python permet de produire avec peu d'efforts du code de qualité qui soit lisible, peu complexe et dont l'exécution est rapide. Ne pas être capable de relire son propre code, même plusieurs mois après, n'est pas acceptable.

La division du code par une architecture bien pensée permet également la division du travail en équipe, le partage des responsabilités et peut être associée à des techniques de programmation comme la programmation agile.

Enfin, l'utilisation de toutes les possibilités offertes par le langage est également un axe important. En effet, la division du travail s'applique à l'application elle-même. En effet, une opération sur une grande quantité de données peut être transmise à des processus de manière à utiliser les ressources matérielles au mieux et à améliorer considérablement les performances.

Une certaine quantité de bonnes pratiques sont distillées, voire répétées tout au long des chapitres précédents, comme la bonne utilisation des types de données, la bonne manière d'itérer, et surtout, ne pas réinventer ce qui existe déjà. Par exemple, les modules **`itertools`** et **`functools`** sont de véritables petits trésors d'ingéniosité qui permettent de beaucoup gagner en performances.

3.4.3 Profiler un algorithme

On touche ici à des notions assez complexes qui nécessitent, si l'on veut aller au bout, de connaître les tréfonds du langage, c'est-à-dire la manière dont il est construit, puis d'être capable d'évaluer la complexité des algorithmes que l'on écrit.

Afin d'avoir une véritable idée de la complexité des opérations que l'on utilise couramment, il faut aller sur le wiki de Python : https://wiki.python.org/moin/TimeComplexity

On y apprendra par exemple que la méthode **`insert`** d'une liste est **`O(n)`** alors que la méthode **`appendleft`** d'un **`deque`** est **`O(1)`**, ce qui explique pourquoi le **`deque`** est plus intéressant que la liste.

Cependant, nous n'allons pas aller jusque-là dans cette section. Nous allons simplement présenter une petite introduction qui va vous permettre d'avoir les premières notions afin de de comprendre l'essentiel puis de découvrir le reste par vous-même.

Pour commencer, il faut savoir que ce qui compte lorsque l'on regarde la performance, c'est d'abord et avant tout le temps que l'on met à faire exécuter un algorithme.

La première chose à préciser ici et qui est absolument primordiale (et contraire à ce que vous pourrez lire ailleurs, parfois) est que si l'adaptation, la personnalisation d'un code dans le but d'avoir une meilleure performance est quelque chose de sain, le fait de le défigurer, de le rendre illisible ou plus généralement de le rendre moins pythonique est une hérésie.

Ainsi, vous pourrez par exemple lire dans certains blogs que pour obtenir de meilleures performances, il faut ne plus utiliser les objets mais les remplacer par des dictionnaires. Vous verrez aussi des idiotismes de programmation que je ne vous conseille pas de reproduire.

On touche ici à ce qui se fait de pire dans l'optimisation.

En effet, si vous écrivez du Python, faites-le dans les règles de l'art. Si une partie de Python n'est pas suffisamment bien optimisée, laissez les développeurs du langage l'améliorer, mais n'allez pas produire quelque chose d'illisible pour éviter d'avoir à l'utiliser, surtout pour gagner des miettes.

Mais avant de parler d'optimisation, il est important, en premier lieu, d'éviter des erreurs classiques :

```
>>> def test1(l):
... i=0
... while i < len(l):
... print(l[i])
... i += 1
...
```

Dans l'exemple précédent, outre le fait qu'il ne soit pas pythonique, on calcule la longueur de la liste autant de fois qu'il y a d'éléments dedans. Il s'agit là typiquement d'une véritable erreur de développement. Une meilleure solution serait :

```
>>> def test2(l):
... i, m = 0, len(l)
... while i < m:
... print(l[i])
... i += 1
...
```

© Editions ENI - All rights reserved

Et c'est un exemple parfait pour illustrer le fait que dégrader la qualité de son code n'est pas une bonne idée. En effet, on le comprend nettement moins bien que la solution idéale et pythonique qui serait tout simplement :

```
>>> def test3(l):
... for e in l:
... print(e)
...
```

Et comme on va le voir immédiatement, on obtient des performances moins bonnes :

```
>>> from timeit import timeit
>>> timeit('test1(l)', number=1000, setup="from __main__ import l, test1")
0.7539350385284424
>>> timeit('test2(l)', number=1000, setup="from __main__ import l, test2")
0.7414303462579703
>>> timeit('test3(l)', number=1000, setup="from __main__ import l, test3")
0.7103568627929688
```

La conclusion s'impose d'elle-même : faites confiance aux développeurs de Python. Derrière l'apparente simplicité de la boucle for classique se cachent des opérations qui sont en réalité très optimisées, grâce notamment à l'intervention de générateurs.

On en profite pour voir comment fonctionne le module **`timeit`**. Il suffit de préciser le nom de la fonction à exécuter, le nombre de boucles à réaliser (faire en sorte d'avoir au moins 100 boucles et un résultat qui soit dans des temps restant courts (pas plus de 10 secondes)).

On peut appliquer également ces mêmes tests sur la comparaison entre les paradigmes impératif et fonctionnel sur certains aspects :

```
>>> def test1(l):
... result = []
... for e in l:
... result.append(e**2)
... return result
...
>>> def test2(l):
... return [e**2 for e in l]
...
>>> def test3(l):
... return list(map(lambda x: x**2, l) )
...
>>> timeit('test1(l)', number=100000, setup="from __main__ import l, test1")
3.208026170730591
>>> timeit('test2(l)', number=100000, setup="from __main__ import l, test2")
2.8028101921081543
>>> timeit('test3(l)', number=100000, setup="from __main__ import l, test3")
3.5190188884735107
```

Ainsi, la programmation fonctionnelle à l'aide des compréhensions de liste est beaucoup plus performante que l'utilisation d'un algorithme impératif, mais la programmation fonctionnelle à l'aide de **`map`** l'est bien moins.

Avant de passer aux conclusions, on peut tester avec différentes volumétries pour voir comment cela évolue :

```
>>> l = list(range(10000))
>>> timeit('test1(l)', number=1000, setup="from __main__ import l, test1")
3.2622950077056885
>>> timeit('test2(l)', number=1000, setup="from __main__ import l, test2")
2.8443081378936768
>>> timeit('test3(l)', number=1000, setup="from __main__ import l, test3")
3.5359020233154297
```

Dans ce cas-ci, il n'y a pas de changements dans l'ordre de grandeur. Par contre, cela peut évoluer en fonction de la complexité de l'opération à réaliser. Ici, il ne s'agissait que de mettre un nombre au carré.

Vous pouvez donc parfaitement faire ce genre de petit test dans votre console et l'appliquer à tout ce qui vous passe par la tête, mais attention de ne pas forcément sauter aux conclusions trop vite : ce n'est pas parce que quelque chose est vrai dans un cas particulier qu'il faut le généraliser sans y réfléchir à deux fois. Prenez le temps de tester différents scénarios.

Si vous voulez aller plus loin, il est possible d'utiliser le module cProfile pour profiler un extrait de code.

Voici une fonction permettant de réaliser ce test :

```
>>> def tester(callback, *values, **kvalues):
... pr = cProfile.Profile()
... pr.enable()
... callback(*values, **kvalues)
... pr.disable()
... s = StringIO()
... ps = pstats.Stats(pr, stream=s).sort_stats('cumulative')
... ps.print_stats()
... return s.getvalue()
...
```

Pour avoir un exemple pertinent, voici cette fonction appliquée au crible d'Ératosthène :

```
def crible1(m):
"""Algorithme classique pour le crible d'Ératosthène"""
l, n = [i for i in range(2, m+1)], 2
while n:
for i in l[l.index(n)+1:]:
if i % n == 0:
l.remove(i)
if l.index(n) +1 < len(l):
n = l[l.index(n) + 1]
else:
return l

def crible2(m):
"""Algorithme pythonique pour le crible d'Ératosthène"""
l = [i for i in range(m+1)]
```

© Editions ENI - All rights reserved

```
l[1], n = 0, 2
while n**2 <= m:
l[n*2::n], n = [0] * (m//n-1), n+1
while not l[n]: n+= 1
return [i for i in l if i != 0]

def crible3(m):
"""Algorithme alternatif proche d'une syntaxe C"""
found, numbers, i = [], [], 2
while (i <= m):
if i not in numbers:
found.append(i)
for j in range(i, m+1, i):
numbers.append(j)
i += 1
return found

def crible4(m):
"""Algorithme alternatif utilisant NumPy (Python scientifique)"""
if m < 2**31:
t = 'i'
else:
if m>= 2**64:
print('AVERTISSEMENT, le maximum a été limité à %s' % 2**64-1)
t = 'L'
l, n = array(t), 2
l.extend([i for i in range(m+1)])
while n**2 <= m:
l[n*2::n], n = array(t, [0]*(m//n-1)), n+1
while not l[n]: n+= 1
return [i for i in l if i != 0]
```

Voici le résultat pour ces quatre fonctions :

```
Crible1 pour 1000 entiers:
0.0422821044921875
---
1505 function calls in 0.005 seconds

Ordered by: cumulative time

ncalls tottime percall cumtime percall filename:lineno(function)
1 0.001 0.001 0.005 0.005 ./eratosthene.py:39(crible1)
831 0.002 0.000 0.002 0.000 {method 'remove' of 'list' objects}
503 0.001 0.000 0.001 0.000 {method 'index' of 'list' objects}
1 0.000 0.000 0.000 0.000 ./eratosthene.py:41(<listcomp>)
168 0.000 0.000 0.000 0.000 {built-in method len}
```

On voit ici que le temps est très long et qu'il y a beaucoup d'appels, dont des appels de méthodes **O(n)**. Voici maintenant la solution la plus optimisée (50 fois plus rapide) :

```
Crible2 pour 1000 entiers:
0.0008630752563476562
---
4 function calls in 0.000 seconds

Ordered by: cumulative time

ncalls tottime percall cumtime percall filename:lineno(function)
1 0.000 0.000 0.000 0.000 ./eratosthene.py:51(crible2)
1 0.000 0.000 0.000 0.000 ./eratosthene.py:53(<listcomp>)
1 0.000 0.000 0.000 0.000 ./eratosthene.py:58(<listcomp>)
```

On voit clairement peu d'appels, et des appels performants. Voici maintenant une version réécrite en suivant des optimisations applicables à d'autres langages que Python :

```
Crible3 pour 1000 entiers:
0.09385895729064941
---
2296 function calls in 0.009 seconds

Ordered by: cumulative time

ncalls tottime percall cumtime percall filename:lineno(function)
1 0.009 0.009 0.009 0.009 ./eratosthene.py:60(crible3)
2294 0.000 0.000 0.000 0.000 {method 'append' of 'list' objects}
```

On a ici peu d'appels différents, mais on utilise beaucoup la méthode append de la liste qui est **O(1)**. La plus grande partie du temps est perdue dans l'itération et la gestion des variables. Au final, le code est plus obscur que pour la première solution et prend deux fois plus de temps. On perd donc sur tous les tableaux.
Enfin, voici un exemple utilisant les bibliothèques scientifiques :

```
Crible4 pour 1000 entiers:
0.002115964889526367
---
5 function calls in 0.000 seconds

Ordered by: cumulative time

ncalls tottime percall cumtime percall filename:lineno(function)
1 0.000 0.000 0.000 0.000 ./eratosthene.py:71(crible4)
1 0.000 0.000 0.000 0.000 {method 'extend' of 'array.array' objects}
1 0.000 0.000 0.000 0.000 ./eratosthene.py:80(<listcomp>)
1 0.000 0.000 0.000 0.000 ./eratosthene.py:84(<listcomp>)
```

Sans surprises, la performance est relativement bonne par rapport à la solution classique, en raison du fait que les objets scientifiques sont des objets de plus bas niveau, très optimisés. Malgré tout, ce code est deux fois et demie plus lent que le code pythonique.

© Editions ENI - All rights reserved

Lorsque la pile d'appels est importante et que l'algorithme utilise plusieurs objets, on est capable de savoir exactement combien d'opérations élémentaires ont été réalisées, le temps moyen d'une seule de ces opérations et le temps total.
On peut ainsi savoir où optimiser :
- en cherchant à optimiser une opération qui se produit rarement mais consomme une large fraction du temps ;
- en cherchant à optimiser une opération qui prend peu de temps mais est très utilisée ;
- en cherchant à réduire le nombre d'appels d'une fonctionnalité qui est coûteuse.

En effet, si l'on cherche à optimiser une fonction qui ne représente qu'une toute petite consommation du temps total et qui n'est appelée qu'une seule fois, on perd du temps.
Il faut noter que des frameworks comme pyramid rendent de telles statistiques disponibles pour vérifier les opérations les plus coûteuses pour chaque requête HTTP, ce qui permet de faire de l'optimisation très facilement. Lire ces données et les comprendre est donc important.
Lorsque c'est le cas, on obtient des traces très longues et pas forcément aisées à interpréter au premier coup d'œil. Heureusement, il y a des outils comme KCacheGrind qui permettent de voir immédiatement les éléments à optimiser.
Enfin, la dernière partie de cette section sera dédiée au désassemblage du code Python. Pour réaliser ceci, il suffit d'utiliser un simple module :

```
>>> from dis import dis
```

Comme le sujet est complexe, on va proposer des exemples progressifs :

```
>>> def test():
...     pass
...
>>> dis(test)
  2           0 LOAD_CONST               0 (None)
              3 RETURN_VALUE
```

Dans cet exemple, on définit une fonction vide. Elle se traduit ces par deux opérations de base. En effet, par défaut, toute fonction renvoie une et une seule valeur. En l'absence d'une instruction **`return`**, la valeur **`None`** est tout de même renvoyée.
Cette fonction va donc charger la valeur **`None`** et la renvoie.
Voici un second exemple :

```
>>> def test():
...     return 42
...
>>> dis(test)
  2           0 LOAD_CONST               1 (42)
              3 RETURN_VALUE
```

La fonction renvoie explicitement une valeur. Cette valeur est considérée comme une constante. Elle est chargée puis renvoyée.
Voici une légère variation :

```
>>> def test(a):
...     return a
```

```
...
>>> dis(test)
2 0 LOAD_FAST 0 (a)
3 RETURN_VALUE
```

Dans cet exemple, on renvoie explicitement une variable qui a été passée en paramètre. Les deux opérations sont le chargement de cette variable et son renvoi.
Réalisons maintenant une opération :

```
>>> def test(a):
... return a * 42
...
>>> dis(test)
2 0 LOAD_FAST 0 (a)
3 LOAD_CONST 1 (42)
6 BINARY_MULTIPLY
7 RETURN_VALUE
```

Comme précédemment, on reconnaît le chargement du paramètre a, le chargement de la constante 42 et le renvoi du résultat. Entre-temps, on a utilisé une opération de multiplication.
On peut produire le même exemple en utilisant deux paramètres. Comme attendu, seule la différence de méthode de chargement apparaît :

```
>>> def test(a, b):
... return a * b
...
>>> dis(test)
2 0 LOAD_FAST 0 (a)
3 LOAD_FAST 1 (b)
6 BINARY_MULTIPLY
7 RETURN_VALUE
```

On peut également voir comment on traite l'appel d'une fonction :

```
>>> def test(l):
... return len(l)
...
>>> dis(test)
2 0 LOAD_GLOBAL 0 (len)
3 LOAD_FAST 0 (l)
6 CALL_FUNCTION 1
9 RETURN_VALUE
```

On charge la fonction (présente dans les variables globales) puis le paramètre, on réalise l'appel de fonction et on renvoie la valeur obtenue.
Voici maintenant une itération :

```
>>> def test(l):
... for e in l:
... print(e)
...
```

© Editions ENI - All rights reserved

```
>>> dis(test)
2 0 SETUP_LOOP 24 (to 27)
3 LOAD_FAST 0 (l)
6 GET_ITER
>> 7 FOR_ITER 16 (to 26)
10 STORE_FAST 1 (e)

3 13 LOAD_GLOBAL 0 (print)
16 LOAD_FAST 1 (e)
19 CALL_FUNCTION 1
22 POP_TOP
23 JUMP_ABSOLUTE 7
>> 26 POP_BLOCK
>> 27 LOAD_CONST 0 (None)
30 RETURN_VALUE
```

On visualise immédiatement qu'une simple boucle nécessite en réalité beaucoup d'opérations primaires. On commence par **SETUP_LOOP** puis on charge le paramètre.
À partir de ce paramètre, on recherche l'itérateur associé (méthode **__iter__**) puis on l'utilise (ligne 7).
On utilise **STORE_FAST** et **LOAD_FAST** pour gérer la variable de boucle (ici l'élément **e**) qui contient la valeur de la liste pour l'itération courante.
Ensuite, on reconnaît **LOAD_GLOBAL** qui permet de charger la fonction **print** comme précédemment pour la fonction **len** et le **CALL_FUNCTION** qui permet de l'exécuter. Ces opérations sont réalisées pour chaque itération.
Enfin, **JUMP_ABSOLUTE** permet de revenir au début de la boucle (on note la présence du compteur **7** comme paramètre) et **POP_BLOCK** marque la fin de la boucle.
Les deux dernières instructions permettent de renvoyer None puisque cette fonction n'a pas d'instruction **return**.
On peut procéder ainsi sur divers éléments classiques d'algorithmique jusqu'à en connaître les principales instructions puis commencer à étudier des algorithmes plus complexes (et aller jusqu'à étudier les différentes fonctions du crible d'Ératosthène).
Pour clore ce chapitre, on va se contenter d'analyser les différences entre les approches impératives et fonctionnelles vues précédemment :

```
>>> def test1(l):
... result = []
... for e in l:
... result.append(e**2)
... return result
...
>>> dis(test1)
2 0 BUILD_LIST 0
3 STORE_FAST 1 (result)

3 6 SETUP_LOOP 31 (to 40)
9 LOAD_FAST 0 (l)
12 GET_ITER
```

```
>> 13 FOR_ITER 23 (to 39)
16 STORE_FAST 2 (e)

4 19 LOAD_FAST 1 (result)
22 LOAD_ATTR 0 (append)
25 LOAD_FAST 2 (e)
28 LOAD_CONST 1 (2)
31 BINARY_POWER
32 CALL_FUNCTION 1
35 POP_TOP
36 JUMP_ABSOLUTE 13
>> 39 POP_BLOCK

5 >> 40 LOAD_FAST 1 (result)
43 RETURN_VALUE
```

On reconnaît ici la création d'une variable contenant une liste (**BUILD_LIST**, **STORE_FAST**), les divers éléments permettant la gestion de la boucle (**SETUP_LOOP**, **GET_ITER**, **FOR_ITER**, **JUMP_ABSOLUTE** et **POP_BLOCK**), la gestion de la variable de boucle et des opérations menées (**BINARY_POWER**) et enfin le retour du résultat (**LOAD_FAST** et **RETURN_VALUE**).

On peut comparer cela à l'utilisation d'une compréhension de liste :

```
>>> def test2(l):
... return [e**2 for e in l]
...
>>> dis(test2)
2 0 LOAD_CONST 1 (<code object <listcomp> at 0x7f746dc8fe88,
file "<stdin>", line 2>)
3 MAKE_FUNCTION 0
6 LOAD_FAST 0 (l)
9 GET_ITER
10 CALL_FUNCTION 1
13 RETURN_VALUE
```

La différence est flagrante.

Enfin, voici l'alternative fonctionnelle par l'utilisation de **map** :

```
>>> def test3(l):
... return list(map(lambda x: x**2, l) )
...
>>> dis(test3)
2 0 LOAD_GLOBAL 0 (list)
3 LOAD_GLOBAL 1 (map)
6 LOAD_CONST 1 (<code object <lambda> at 0x7f746dc8f938,
file "<stdin>", line 2>)
9 MAKE_FUNCTION 0
12 LOAD_FAST 0 (l)
15 CALL_FUNCTION 2
18 CALL_FUNCTION 1
21 RETURN_VALUE
```

© Editions ENI - All rights reserved

On voit le chargement des fonctions, de la fonction lambda, des données, puis les appels de fonction et le renvoi du résultat.
Pour terminer, insistons sur le fait que l'ensemble de ce qui est présenté dans cette section n'est pas réservé aux experts. D'une part, il est toujours utile d'avoir un petit peu de culture générale sur ces aspects techniques de Python, et d'autre part la connaissance de ces mécanismes et un peu de curiosité peuvent permettre de faire les bons choix et de gagner en expérience de façon tout à fait autodidacte.

3.4.4 Optimiser l'utilisation de la mémoire

Une des nouveautés de Python 3.4 est qu'il permet également d'avoir une vision claire de ce qu'il se passe en mémoire, ce qui permet également des optimisations. Il s'agit du module **`tracemalloc`**.
Voici un exemple permettant de retrouver les dix objets les plus gourmands en mémoire :

```
#!/usr/bin/env python3.4

import tracemalloc

def test1(l):
result = []
for e in l:
result.append(e**2)
return result

def test2(l):
return [e**2 for e in l]

def test3(l):
return list(map(lambda x: x**2, l) )

def tester_memoire_top10(callback, *args, **kwargs):
tracemalloc.start()
callback(*args, **kwargs)
snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')
print("[ Top 10 ]")
for stat in top_stats[:10]:
print(stat)

if __name__ == '__main__':
l = range(100)
print('Test 1 pour l=range(100)')
tester_memoire_top10(test1, l)
print('Test 2 pour l=range(100)')
tester_memoire_top10(test2, l)
print('Test 3 pour l=range(100)')
tester_memoire_top10(test3, l)
```

Et voici une sélection sur le résultat associé (que vous pouvez reproduire en exécutant le code) :

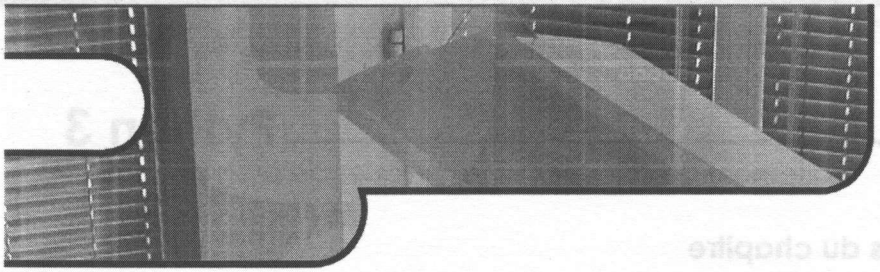
```
Test 1 pour l=range(100)
[ Top 10 ]
Test 2 pour l=range(100)
[ Top 10 ]
./test_memoire.py:12: size=356 B, count=1, average=356 B
Test 3 pour l=range(100)
[ Top 10 ]
./test_memoire.py:15: size=376 B, count=2, average=188 B
```

L'interprétation est relativement simple. La première méthode ne consomme que très peu de mémoire, on n'a donc aucune alerte. La seconde en consomme plus, à hauteur de 356 octets, ce qui est relativement peu, tandis que la troisième consomme à peine plus.
Enfin, pour terminer, voici comment procéder pour détecter une fuite mémoire :

```
import tracemalloc
tracemalloc.start()
# ... start your application ...

snapshot1 = tracemalloc.take_snapshot()
# ... call the function leaking memory ...
snapshot2 = tracemalloc.take_snapshot()

top_stats = snapshot2.compare_to(snapshot1, 'lineno')

print("[ Top 10 differences ]")
for stat in top_stats[:10]:
print(stat)
```

L'idée est de prendre deux captures et de les comparer. Ceci permet d'identifier l'origine d'une fuite de mémoire.

© Editions ENI - All rights reserved



Partie 2 : Programmation système, réseau, web et scientifique

Chapitre 2.1 Programmation système

1. Présentation

1.1 Définition

La programmation système se définit par opposition à la programmation d'applications. Il ne s'agit pas de concevoir un logiciel qui va utiliser le système et ses ressources pour effectuer une action, mais de concevoir une des briques qui va s'intégrer au système lui-même. Cela peut donc être le développement d'un pilote pour un matériel, d'une interface réseau ou encore la gestion des ressources.

Par extension, la création d'un programme qui utilise d'autres programmes systèmes est de la programmation système. Le terme programmation système s'étend alors à tout ce qui peut permettre à un administrateur système de résoudre les problématiques usuelles concernant son domaine de compétence, à savoir la gestion des utilisateurs, des périphériques, des processus, de la sauvegarde...

Ainsi, par extension, l'utilisation des commandes **`bash`** est de la programmation système. L'utilisation des commandes **`mysql`** et **`mysqldump`** pour opérer des actions de sauvegarde quotidiennes l'est également.

Un système d'exploitation moderne est écrit majoritairement en C, le reste étant de l'assembleur spécifique à la machine. Ce même système d'exploitation – moderne – a des fonctionnalités de haut niveau, telles que le gestionnaire de paquets qui a besoin par exemple d'effectuer des opérations diverses telles que des échanges réseau pour télécharger des paquets, vérifier leur intégrité, les ouvrir, les installer.

Celles-ci sont le plus souvent écrites en Python, parce qu'il est un langage facile à manipuler, efficace, complet et surtout fiable, disposant de l'outillage nécessaire.

1.2 Objectifs du chapitre

Dans ce chapitre sont présentés les moyens mis à disposition par Python pour permettre d'exécuter des commandes système de manière à effectuer des opérations de maintenance, l'utilisation du système de fichiers, les moyens permettant à Python d'être une alternative crédible à Bash, en particulier par le traitement du passage d'arguments.

Les problématiques liées à la gestion des protocoles réseau, souvent associées à de la programmation système, sont également présentées, et orientées vers la communication entre différents types de clients et de serveurs et différentes technologies. Cela couvre également les services web.

La gestion des tâches et processus à haut niveau est également une problématique faisant partie de la programmation système. Mais étant de haut niveau, elle est utilisée beaucoup plus pour des applications qu'en tant qu'élément de programmation système pure. C'est la raison pour laquelle cette partie est traitée dans un chapitre dédié, le chapitre Programmation parallèle.

2. Appréhender son système d'exploitation

2.1 Avertissement

L'exécution de commandes externes est intimement liée au système sur lequel se trouve installé Python. D'une part, chaque système d'exploitation possède ses propres commandes. Par exemple, pour lister un répertoire, on utilisera **`ls`** ou **`dir`**.

Cette section traite principalement les commandes Unix.

Au-delà des commandes système classiques, certaines commandes comme **`mysql`** ou **`mysqldump`** ne peuvent être utilisées que si les programmes adéquats ont été installés, quel que soit le système.

2.2 Système d'exploitation

Python propose un module de bas niveau permettant de gérer des informations sur le système d'exploitation :

```
>>> import os
```

Voici les deux principaux moyens de vérifier la nature du système :

```
>>> os.name
'posix'
>>> os.uname()
('Linux', 'nom_donne_au_host', '2.6.38-11-generic', '#50-Ubuntu
SMP Mon Sep 12 21:17:25 UTC 2011', 'x86_64')
```

La première commande donne une standardisation de l'environnement et la seconde des détails sur le nom du système d'exploitation, de la machine, le nom du noyau et sa version ainsi que l'architecture de la machine. Tester ces valeurs permet d'effectuer des choix et d'adapter une application à un environnement précis pour certaines opérations qui le nécessitent.

© Editions ENI - All rights reserved

Voici comment trouver la liste des variables d'environnement :

```
>>> list(os.environ.keys())
```

Et voici comment aller chercher la valeur d'une de ces variables :

```
>>> os.getenv('LANGUAGE')
'fr_FR:fr'
```

L'exploitation de ces variables d'environnement permet également de diriger des choix permettant l'adaptation de l'application. Dans le cas qui vient d'être vu, le choix de la locale peut servir à produire une interface adaptée au langage de l'utilisateur. Il est possible de lire les variables d'environnement sous forme d'octets avec **os.environb** (utile lorsque Python est unicode, mais pas le système).
Python permet également de modifier ces variables d'environnement à l'aide de la méthode **putenv**. L'environnement est alors affecté dans tous les sous-processus.

2.3 Processus courant

Python permet d'obtenir des informations sur le processus courant. Ces fonctionnalités ne sont disponibles que pour Linux.
La principale d'entre elles est l'identifiant du processus courant et celui du parent :

```
>>> os.getpid()
5256
>>> os.getppid()
3293
```

Le parent peut correspondre à l'identifiant d'un processus Python père ou à celui de la console lorsque Python est lancé en mode console depuis la console.
Il peut récupérer l'utilisateur affecté au processus et l'utilisateur effectif :

```
>>> os.getuid()
1000
>>> os.geteuid()
1000
```

On peut également avoir des informations textuelles :

```
>>> os.getlogin()
'sch'
```

Et des informations sur les groupes rattachés au processus :

```
>>> os.getgroups()
[4, 20, 24, 46, 112, 120, 122, 1000]
```

Ainsi que sur le 3-uplet (utilisateur courant, effectif, sauvegardé) :

```
>>> os.getresuid()
(1000, 1000, 1000)
```

Voici le 3-uplet des groupes associés (courant, effectif, sauvegardé) :

```
>>> os.getresgid()
(1000, 1000, 1000)
```

L'identifiant 0 est celui de root, 1000 celui du premier utilisateur créé (lors de l'installation du système) pour Unix.
Si l'on ne veut pas que l'application puisse être lancée par root, on peut faire :

```
>>> if os.getuid() == 0:
...     print('Ne doit pas être lancé avec root...')
```

Enfin, il est possible de retrouver le terminal contrôlant le processus :

```
>>> os.ctermid()
'/dev/tty'
```

Pour plus d'informations :

```
$ man tty
```

Pour tester les différences, la console peut être lancée en root :

```
$ sudo python3
```

2.4 Utilisateurs et groupes

Un système d'exploitation moderne est multi-utilisateur et permet de retrouver des informations sur les utilisateurs déclarés.
Python permet de rechercher des renseignements sur les utilisateurs en se servant des fichiers qui stockent ces informations :

```
>>> with open("/etc/passwd") as f:
...     users = [l.split(':', 6) for l in f]
...
```

On peut ainsi afficher les données de l'utilisateur simplement :

```
>>> users[0]
['root', 'x', '0', '0', 'root', '/root', '/bin/bash\n']
```

Elles correspondent respectivement aux :
- nom d'utilisateur ;
- mot de passe encrypté (ou stocké chiffré dans un fichier séparé) ;
- identifiant de l'utilisateur ;
- identifiant de son groupe ;
- nom complet ;
- répertoire d'accueil ;
- shell au démarrage de sa session.

Avec ces données, celui qui connaît le fonctionnement de son système sait à quels niveaux il peut intervenir pour effectuer des modifications.
Voici comment obtenir l'ensemble des valeurs utilisées pour les différents utilisateurs :

```
>>> {u[6] for u in users}
{'/bin/sh\n', '/bin/false\n', '/bin/sync\n',
'/bin/bash\n', '/usr/sbin/nologin\n'}
>>> {u[5] for u in users}
{'/home/sch', '/var/www', '/root', '/var/lib/bacula', [...]}
```

© Editions ENI - All rights reserved

On peut utiliser les mêmes procédés pour les groupes :

```
>>> with open("/etc/group") as f:
...     groups = [l.split(':', 3) for l in f]
...
>>> groups[0]
['root', 'x', '0', '\n']
```

Le troisième élément est le numéro du groupe servant de lien avec l'utilisateur.
Voici comment mettre cette relation en évidence :

```
>>> guser = {u[3]: u[0] for u in users}
>>> user_group = [(guser.get(g[2]), g[0]) for g in groups]
```

On voit donc en peu d'exemples comment utiliser la puissance des types de Python. Le bon type pour la bonne utilisation.
Voici un script testant l'existence des éléments caractéristiques d'un utilisateur, nom d'utilisateur, le nom de groupe associé et son dossier personnel à l'emplacement par défaut :

```
>>> for username in ['sch', 'existepas']:
...     if username in (u[0] for u in users):
...         print("L'utilisateur %s existe déjà" % username)
...     if username in (g[0] for g in groups):
...         print("Le groupe %s existe déjà" % username)
...     home = '/home/%s' % username
...     if os.path.exists(home):
...         print('Le dossier %s existe déjà' % username)
...
L'utilisateur sch existe déjà
Le groupe sch existe déjà
Le dossier sch existe déjà
```

Enfin, pour terminer, les utilisateurs courants du système (pas les utilisateurs Apache, Bacula ou autre) ont un identifiant compris entre certaines bornes :

```
>>> max_user_id = max([id for id in (int(u[2]) for u in users) if
1000 < id < 19999])
>>> max_group_id = max([id for id in (int(g[2]) for g in groups)
if 1000 < id < 19999])
```

Voici les résultats pour ma machine qui ne contient que deux comptes d'utilisateurs courants :

```
>>> max_user_id, max_group_id
(1001, 1001)
```

On peut aussi signaler le module **pwd**, qui permet d'avoir une interface de plus haut niveau pour visualiser ces informations :

```
>>> import pwd
>>> pwd.getpwuid(1000)
pwd.struct_passwd(pw_name='sch', pw_passwd='x', pw_uid=1000, pw_gid=1000,
pw_gecos='Sébastien CHAZALLET,,,', pw_dir='/home/sch', pw_shell='/bin/bash')
```

On peut également accéder à l'information directement depuis le nom de l'utilisateur :

```
>>> infos = pwd.getpwnam('sch')
```

Il ne reste plus qu'à savoir exploiter ces données, semblables à une structure C :

```
>>> infos = pwd.getpwuid(1000)
>>> infos.pw_name
'sch'
>>> infos.pw_shell
'/bin/bash'
```

On peut ainsi utiliser toutes les possibilités offertes par le langage Python :

```
>>> [info.pw_name for info in pwd.getpwall() if info.pw_shell=='/bin/bash']
['root', 'sch', 'postgres', 'couchdb', 'openerp']
```

Les informations sur les groupes sont également accessibles par l'utilisation d'un autre module de plus haut niveau :

```
>>> import grp
>>> grp.getgrgid(1000)
grp.struct_group(gr_name='sch', gr_passwd='x', gr_gid=1000, gr_mem=[])
>>> grp.getgrnam('postgres')
grp.struct_group(gr_name='postgres', gr_passwd='x', gr_gid=132, gr_mem=[])
>>> infos = grp.getgrnam('audio')
>>> infos.gr_mem
['pulse', 'timidity']
```

2.5 Mots de passe et authentification

Dans les systèmes Unix modernes, les mots de passe sont encodés dans un fichier à part (**/etc/shadow/**), en **sha512**. On parle d'empreinte de mot de passe. Pour authentifier un utilisateur, on lui demande de saisir un mot de passe et l'on encode ce dernier. Si le résultat obtenu est égal à l'empreinte, alors le mot de passe est correct. *La politique de sécurité* de ces systèmes est un élément important qui est également inclus dans ce fichier.
Il existe un module permettant d'accéder aux empreintes de mots de passe, mais il est nécessaire pour l'utiliser d'avoir le droit de lire le fichier contenant ces empreintes et par conséquent faire partie du groupe **shadow** ou être **root** :

```
>>> import spwd
>>> spwd.getspnam('sch')
spwd.struct_spwd(sp_nam='sch', sp_pwd='$6$6kBpTzs5$PqAyNY10gN/M4Cg-
FPwXUE9vdS3zzx/tW8zhi.Z6dFX4ndO9C2HgXnaKlGNm60ll2gl3.buvyGVg-
mWn4ACXZbK0', sp_lstchg=15560, sp_min=0, sp_max=99999, sp_warn=7,
sp_inact=-1, sp_expire=-1, sp_flag=-1)
```

Il s'agit là encore d'une structure C à exploiter :

```
>>> infos = spwd.getspnam('sch')
>>> infos.sp_min
15
>>> infos.sp_max
30
```

© Editions ENI - All rights reserved

Pour authentifier un utilisateur du système, on peut procéder ainsi :

```
>>> import spwd, os
>>> from passlib.hosts import host_context
>>> def authentifier(user, passwd):
...     try:
...         hash = spwd.getspnam(user).sp_pwd
...     except KeyError:
...         return False
...     return host_context.verify(passwd, hash)
...
```

La boucle d'exception permet de gérer le cas d'un utilisateur qui n'existe pas. Voici comment utiliser cette fonction :

```
>>> authentifier("sch", "IlovePython")
True
>>> authentifier("sch", "123456")
False
>>> authentifier("existe_pas", "123456")
False
```

Remarque

Attention cependant, cette fonction ne pourra pas être utilisée si l'on n'appartient pas au groupe ***shadow*** *(il n'est pas conseillé, de manière générale, d'utiliser* ***root****).*

2.6 Constantes pour le système de fichiers

Le système d'exploitation définit certaines caractéristiques par rapport au système de fichiers. Il s'agit de la notation du répertoire courant, du répertoire parent, du séparateur de répertoire (il peut y en avoir un second), du séparateur d'extensions, de la séparation entre chemins lorsqu'ils sont écrits l'un après l'autre, et le séparateur qui définit le changement de ligne :

```
>>> os.curdir, os.pardir
('.', '..')
>>> os.sep, os.altsep, os.extsep, os.pathsep, os.linesep
('/', None, '.', ':', '\n')
```

Chaque système définit également un chemin par défaut (liste de répertoires séparés par le séparateur **os.pathsep**) et un chemin vers une interface nulle :

```
>>> os.defpath
':/bin:/usr/bin'
>>> os.devnull
'/dev/null'
```

Il existe encore un certain nombre d'opérations plus spécialisées, mais celles présentées permettent déjà de réaliser un code indépendant du système :

```
>>> os.sep.join([os.curdir, 'rep', 'fname' + os.extsep + 'ext'])
'./rep/fname.ext'
```

2.7 Gérer les chemins

Python 3.4 introduit une sémantique objet permettant de gérer les chemins beaucoup plus simplement que des chaînes de caractères :

```
>>> from pathlib import Path
>>> mypath = Path('/var/www')
>>> mypath.exists()
True
>>> mypath.is_dir()
True
```

Pour les versions antérieures à Python 3.4, il est possible d'installer le module **pathlib2** :

```
$ sudo pip install pathlib2
```

Ce module est parfaitement intégré avec les modules Python déjà existants, parmi lesquels le fameux module **glob** qui permet de faire des recherches de fichier très efficaces :

```
>>> www = Path('/var/www')
>>> list(www.glob('**/*.html'))
[PosixPath('index.html'), PosixPath('doc/index.html')]
```

Là où l'outil devient vraiment très efficace sémantiquement parlant, c'est qu'il profite à fond de la capacité de Python à surcharger les opérateurs et qu'il l'utilise intelligemment de manière à rendre son utilisation très basique et surtout particulièrement lisible :

```
>>> doc, index = PurePath('doc'), PurePath('index.html')
>>> myfile = mypath / doc / index
```

Et cela fonctionne aussi en utilisant un opérande issu de pathlib et de simples chaînes de caractères :

```
>>> myfile = www / 'doc' / 'index.html'
>>> myfile.is_dir()
False
```

Enfin, voici comment ouvrir un fichier à partir de l'objet précédent :

```
>>> with myfile.open() as f:
... f.readline()
'<html>\n'
```

Vous trouverez beaucoup d'autres cas d'utilisation de ce module dans sa documentation : http://docs.python.org/3.4/library/pathlib.html

2.8 Utilitaires

2.8.1 Comparer deux fichiers

Voici un utilitaire très pratique qui est utilisé pour intervenir sur des fichiers de données. Nous partons d'un exemple de deux fichiers contenant un enregistrement par ligne et nous comparons différentes versions.
Voici les deux fichiers :

```
>>> old, new = 'data.old.sql', 'data.sql'
```

© Editions ENI - All rights reserved

Voici comment récupérer les données :

```
>>> with open(old) as f:
...     odatas = f.readlines()
...
>>> with open(new) as f:
...     ndatas = f.readlines()
...
```

Voici comment traiter l'ensemble des données pour obtenir un fichier unifié avec les différences notées par un signe en début de ligne. Ceci est à utiliser pour un affichage dans une interface graphique, mais est difficilement exploitable.

```
>>> from difflib import Differ
>>> result = list(Differ().compare(odatas, ndatas))
>>> len(result)
354811
>>> len(odatas)
354809
>>> len(ndatas)
354809
```

On a deux ajouts et deux suppressions, donc deux lignes supplémentaires, les lignes ajoutées sont précédées du signe plus et celles retirées du signe moins.
Voici une ligne non modifiée :

```
>>> result[0]
'  -- MySQL dump 10.13  Distrib 5.1.54, for debian-linux-gnu (x86_64)\n'
```

Voici comment obtenir la liste des lignes modifiées :

```
>>> [r for r in result if r.startswith('- ')]
>>> [r for r in result if r.startswith('+ ')]
```

Mais cela ne permet pas de savoir la localisation des lignes, cela ne sert que pour un affichage, car le but d'une comparaison est aussi d'être le plus léger possible.
Voici une autre manière de parvenir à ce résultat :

```
>>> from difflib import ndiff
>>> nd = list(ndiff(odatas, ndatas))
>>> len(nd)
354811
>>> nd[0]
'  -- MySQL dump 10.13  Distrib 5.1.54, for debian-linux-gnu (x86_64)\n'
```

Ce que l'on veut, c'est disposer d'une comparaison qui ne mette en évidence que les différences et non pas tout le fichier et qu'il soit au format ndiff :

```
>>> from difflib import unified_diff
>>> u = list(unified_diff(odatas, ndatas, n=3))
>>> len(u)
22
```

On a ici un contexte de trois lignes autour des différences. Il est possible de supprimer ce contexte :

```
>>> u = list(unified_diff(odatas, ndatas, n=0))
>>> len(u)
9
```

Voici comment afficher le résultat :

```
>>> for l in u:
...     print(l)
...
---

+++

@@ -28,0 +28,1 @@
[...]
```

Il peut être utile d'afficher les noms des fichiers ayant permis d'effectuer les différences.

```
>>> u = list(unified_diff(odatas, ndatas, old, new, n=0))
>>> for l in u:
...     print(l)
...
--- data.old.sql

+++ data.sql

@@ -28,0 +28,1 @@
[...]
```

La documentation officielle offre un utilitaire très bien fait et complet qui permet également de voir au-delà de la simple problématique de la comparaison.

2.8.2 Utilitaire de sauvegarde

Un programme peut avoir une ou plusieurs manières de se terminer. Il peut également se terminer par une exception. Pour ces raisons, il faut prévoir un moyen d'effectuer des actions quelle que soit la manière dont le programme se termine (cela ne fonctionne pas lorsque Python est terminé par des signaux externes non pris en compte ou des erreurs fatales) :

```
>>> datas = {}
>>> @atexit.register
... def quit():
...     print('Sauvegarde des données')
...
```

Exécuter [Ctrl] + d dans la console pour la quitter et tester la fonctionnalité.

© Editions ENI - All rights reserved

2.8.3 Lire un fichier de configuration

Les fichiers de configuration peuvent être de différentes structures. Certains sont en XML et dans ce cas, il faut utiliser un parser XML pour récupérer les informations utiles (comme les fichiers zcml de zope) ou les fichiers de configuration de beaucoup d'applications Java, comme Sonar, par exemple.
Les autres sont les fichiers classiques CONF pour Unix ou INI pour Windows, mais également Unix (php.ini, par exemple).
Il existe un module spécifique qui permet de lire les fichiers INI tels que ceux que l'on peut trouver sous Windows. En voici un :

```
$ cat /tmp/test.ini
[SQL]
backoffice_url = postgres://user:pass@host/base
frontoffice_url = postgres://user:pass@host/base

[EXPORT]
directory = '/var/saves/csv/'
filename = 'export_%s.txt'

[SERVER]
port = 8016
webdav = yes
sftp = yes
```

Voici comment parser ce fichier :

```
>>> import configparser
>>> parser = configparser.ConfigParser()
>>> parser.read('/tmp/test.ini')
['/tmp/test.ini']
```

Ensuite, il est aisé de récupérer les différentes sections :

```
>>> parser.sections()
['SQL', 'EXPORT', 'SERVER']
```

Le parseur est utilisable comme un dictionnaire et chaque section aussi :

```
>>> parser['SQL']
<Section: SQL>
>>> [k for k in parser['SQL'].keys()]
['backoffice_url', 'frontoffice_url']
>>> parser['SQL']['backoffice_url']
'postgres://user:pass@host/base'
```

Et comme les dictionnaires, on dispose des méthodes **get** permettant de fournir une valeur de substitution si la clé (paramètre de configuration) n'est pas présente dans le fichier de configuration et ainsi définit un paramétrage par défaut.
Pour des questions de cohérence avec d'autres fonctionnalités similaires, il est conseillé de passer cet argument via un paramètre nommé **fallback** :

```
>>> parser['SERVER'].get('threads', fallback=8)
```

```
8
>>> parser['SERVER'].get('port', fallback=8080)
'8016'
```

2.8.4 Pickle

La bibliothèque **pickle** est un module permettant de rendre persistantes très facilement des données simplement en les écrivant sur le disque dur sous une représentation très aisément lisible et facile à écrire.
Le module est porté sur la branche 3.x et s'importe ainsi :

```
>>> import pickle
```

L'avantage, outre les performances et la simplicité, est de ne pas dépendre d'un logiciel tiers, d'une technologie tierce.
Voici les données que nous allons stocker :

```
>>> datas = ['datas']
```

Et voici comment :

```
>>> with open('data.pkl', 'wb') as f:
...     pickle.Pickler(f).dump(datas)
...
```

Pour récupérer des données stockées, il faut procéder comme suit :

```
>>> with open('data.pkl', 'rb') as f:
...     datas2 = pickle.Unpickler(f).load()
...
>>> datas2
['datas']
```

Procéder ainsi permet de voir quels sont les mécanismes à l'œuvre, ce qui est important pour personnaliser le fonctionnement de cette procédure. On voit en effet qu'il est possible d'écrire deux classes héritant de **Pickler** et **Unpickler**.
La plupart du temps, le module en lui-même suffit à répondre aux besoins et il a été ajouté deux primitives permettant d'être encore plus concis :

```
>>> with open('data.pkl', 'wb') as f:
...     pickle.dump(datas, f)
...
>>> with open('data.pkl', 'rb') as f:
...     datas2 = pickle.load(f)
...
```

On peut vérifier que la donnée n'est pas altérée ainsi :

```
>>> datas == datas2
True
```

À noter que la donnée est en première position et le fichier en seconde.

© Editions ENI - All rights reserved

Lorsque le fichier est ouvert en mode écriture, il est automatiquement vidé. Ainsi, si l'on souhaite ajouter une donnée à un fichier, on peut soit charger son contenu, le modifier et sérialiser à nouveau l'ensemble, soit procéder comme suit :

```
>>> with open('data.pkl', 'ab') as f:
...     pickle.dump(['Autres données'], f)
...
```

Ainsi, les données sont stockées les unes à la suite des autres, par vague. Si on charge les données comme suit, on n'obtient que la première vague :

```
>>> with open('data.pkl', 'rb') as f:
...     pickle.load(f)
...
['datas']
```

Cependant, on ne doit pas charger des données qui ne sont pas stockées sous peine de générer une erreur :

```
>>> with open('data.pkl', 'rb') as f:
...     pickle.load(f)
...     pickle.load(f)
...     pickle.load(f)
...
['datas']
['Autres données']
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
EOFError
```

Le moyen le plus élégant pour chercher tous les éléments sérialisés dans un même fichier est de créer un générateur. Ce dernier nécessite l'utilisation d'une boucle pour gérer l'exception qui, lorsqu'elle se déclenche, signifie la fin du générateur :

```
>>> def load_generator(filename):
...     with open(filename, 'rb') as f:
...         while True:
...             try:
...                 yield pickle.load(f)
...             except EOFError:
...                 return
...
```

Un autre moyen, plus élégant, permet de se passer du mécanisme des exceptions, qui est toujours plus lourd en termes de performances qu'une simple vérification. On se contente alors simplement de vérifier que l'on n'est pas à la fin du fichier avant d'aller récupérer la donnée.

```
>>> def load_generator(filename):
...     with open(filename, 'rb') as f:
...         while True:
...             if f.read(1) == b'':
...                 return
```

```
...                 f.seek(-1, 1)
...                 yield pickle.load(f)
...
```

Même dans l'esprit, cela est mieux que de dire « faisons-le, et si cela ne marche pas, c'est que l'on avait déjà terminé le travail ».
Voici comment utiliser l'une ou l'autre des solutions :

```
>>> for d in load_generator('data.pkl'):
...     print(d)
...
['datas']
['Autres données']
```

Pour terminer, il est important de savoir que la technique utilisée pour représenter les données dans le fichier est nommée le protocole pickle et que cette dernière utilise la version 2 pour Python 2 et la version 3 pour Python 3. Python 3.4 introduit la version 4.
De manière à lire un fichier écrit par un autre interpréteur, il faudra veiller à la version utilisée, ce qui est utile, par exemple, lorsque pickle est le support de communication utilisé par pyro.

2.9 Compresser et décompresser un fichier

2.9.1 Tarfile

Compresser un fichier est un moyen de diminuer sa taille sans aucune perte d'information. Pour cela, il existe différents algorithmes qui sont utilisables en Python.
Avant de les aborder, il faut savoir qu'il est possible de compresser plusieurs fichiers par ces mêmes algorithmes tout simplement en accolant les fichiers les uns derrière les autres d'une manière réversible pour ensuite passer l'algorithme de compression sur le fichier unique ainsi obtenu. Les fichiers unitaires sont alors nommés membres du fichier global. Pour obtenir une archive contenant des fichiers déjà compressés (fichiers PNG ou ODT par exemple, un tar est suffisant).
L'algorithme permettant de réaliser cette opération est tar et la bibliothèque Python tarfile permet de travailler avec de tels fichiers.
Soient les deux fichiers suivants :

```
$ cat /tmp/fichier1.txt
Première ligne
Seconde ligne
$ cat /tmp/fichier2.txt
Première ligne
Deuxième ligne
Troisième ligne
```

Voici comment construire un fichier tar :

```
>>> import tarfile
>>> with tarfile.open('/tmp/test.tar', 'w') as f:
...     f.add('/tmp/fichier1.txt')
...     f.add('/tmp/fichier2.txt')
...
```

© Editions ENI - All rights reserved

Le résultat est :

```
$ cat /tmp/test.tar
tmp/fichier1.txt000064400017500001750000000000036116413223750134700
0ustar  schsch00000000000000Première ligne
Seconde ligne
tmp/fichier2.txt000064400017500001750000000000061116413224240134620
0ustar  schsch00000000000000Première ligne
Deuxième ligne
Troisième ligne
```

Cette opération est très proche d'un travail sur un fichier classique.
Il est également possible de lire un fichier en utilisant en second paramètre **r** à la place de **w** ou en n'en mettant pas :

```
>>> with tarfile.open('/tmp/test.tar') as f:
...     f.extractall('/tmp/test')
...
```

Là encore, l'opération est simple.
Voyons maintenant le résultat :

```
$ ll /tmp/test/
total 4
drwxr-xr-x  3 sch  sch    16 2011-09-30 13:37 ./
drwxrwxrwt 17 root root 4096 2011-09-30 13:37 ../
drwxr-xr-x  2 sch  sch    44 2011-09-30 13:37 tmp/
$ ll /tmp/test/tmp/
total 8
drwxr-xr-x 2 sch sch 44 2011-09-30 13:37 ./
drwxr-xr-x 3 sch sch 16 2011-09-30 13:37 ../
-rw-r--r-- 1 sch sch 30 2011-09-30 13:16 fichier1.txt
-rw-r--r-- 1 sch sch 49 2011-09-30 13:16 fichier2.txt
```

En clair, l'archive a créé l'arborescence complète des fichiers depuis la racine.
Ceci est dû au fait que les chemins stockés dans l'archive (ceux avec lesquels les fichiers ont été ajoutés dans l'archive, ici avec la méthode **add**) contiennent le répertoire « tmp », comme on peut le voir plus bas. Il peut éventuellement être envisageable que les chemins stockés soient absolus ou commencent par **..** qui font référence au parent, auquel cas les fichiers pourraient alors être créés dans des endroits non désirés et pourraient éventuellement écraser des fichiers de configuration importants et mettre à mal la sécurité du système.
Il est donc particulièrement important de vérifier le contenu d'une archive avant de l'extraire, en particulier si elle provient d'une source qui n'est pas une source de confiance.
Pour réaliser ceci, il est possible d'obtenir des informations sur les chemins et les membres d'une archive tar de cette manière :

```
>>> with tarfile.open('/tmp/test.tar') as f:
...     print(f.getnames())
...     print(f.getmembers())
...
```

```
['tmp/fichier1.txt', 'tmp/fichier2.txt']
[<TarInfo 'tmp/fichier1.txt' at 0x248b7a0>, <TarInfo
'tmp/fichier2.txt' at 0x248b600>]
```

Il est possible d'avoir encore plus d'informations ainsi :

```
>>> with tarfile.open('/tmp/test.tar') as f:
...     f.list()
...
-rw-r--r-- sch/sch         30 2011-09-30 13:16:13 tmp/fichier1.txt
-rw-r--r-- sch/sch         49 2011-09-30 13:16:36 tmp/fichier2.txt
>>> with tarfile.open('/tmp/test.tar') as f:
...     f.list(False)
...
tmp/fichier1.txt
tmp/fichier2.txt
```

Ces informations se veulent similaires à ce que produirait un **`ls -l`** ou un **`ls`**.
Il existe également beaucoup de possibilités, par exemple n'extraire que certains fichiers en passant leur chemin en second paramètre de la méthode **`extract`**.
De très nombreuses subtilités du module permettent de réaliser beaucoup d'opérations et ceci relativement simplement, en comparaison avec un script bash ou sh.

2.9.2 Gzip

Gzip est un acronyme pour *GNU zip* et est basé sur l'algorithme **`deflate`** aussi utilisé par le format d'image PNG. Il a été développé à partir de 1991 pour offrir une solution performante et libre.
Une ouverture de fichier **`tar.gz`** se réalise exactement de la même manière que pour les fichiers **`tar`**, la compression étant détectée et l'ensemble étant donc transparent. Il est éventuellement possible de forcer l'utilisation de l'algorithme Gzip par l'utilisation de l'option **`r:gz`**. Concernant l'écriture, il faut spécifier l'algorithme de compression :

```
>>> with tarfile.open('/tmp/test.tar.gz', 'w:gz') as f:
...     f.add('/tmp/fichier1.txt')
...     f.add('/tmp/fichier2.txt')
...
```

Pour ajouter un fichier à une archive, il suffit d'utiliser l'option **`a`** sans plus de précision puisque l'on sait quel algorithme est utilisé vu que l'archive existe déjà. L'algorithme de compression est disponible dans le module **`gzip`** et peut s'appliquer directement à un seul fichier selon les mêmes principes (http://docs.python.org/library/gzip.html?highlight=gzip#module-gzip). Ce module contient également des fonctions **`compress`** et **`decompress`** permettant de compresser et décompresser des chaînes d'octets en mémoire (**`bytes`**).
Certaines applications écrivent leurs erreurs dans un fichier textuel, mais le compressent après rotation (un fichier de log par jour), d'autres applications en Python peuvent directement écrire dans un fichier compressé, lisible avec **`zcat`**.

© Editions ENI - All rights reserved

2.9.3 Bz2

Bzip2 est un algorithme de compression développé à partir de 1996 qui propose un taux de compression meilleur que gzip, mais plus lent à la compression et à la décompression. Une alternative à BZ2 est LZMA qui offre une compression encore meilleure, des temps plus courts, mais une consommation mémoire plus élevée. Une bibliothèque externe permet d'utiliser cet algorithme mais n'est pas porté sous python3 (http://www.joachim-bauch.de/projects/pylzma/).

Les fichiers bz2 sont gérés comme les fichiers Gzip. L'ouverture est transparente ou peut être forcée via l'utilisation de l'option **r:bz2** et l'écriture se fait ainsi :

```
>>> with tarfile.open('/tmp/test.tar.bz2', 'w:bz2') as f:
...     f.add('/tmp/fichier1.txt')
...     f.add('/tmp/fichier2.txt')
...
```

L'algorithme de compression est disponible dans le module bz2 et peut s'appliquer directement à un seul fichier selon les mêmes principes (http://docs.python.org/library/bz2.html).

2.9.4 Zipfile

L'algorithme zip est un algorithme qui réalise l'équivalent de tar et de gzip. Il est devenu un standard pour le système d'exploitation Windows et est également utilisé dans certains formats, comme les JAR ou les fichiers OpenDocument (qui sont une arborescence de documents XML).

Il présente l'avantage d'être supporté plus facilement pour les utilisateurs de Windows, mais l'inconvénient de mal stocker les séparateurs de répertoires et de poser des problèmes en cas de différences d'encodage.

Pour éviter ces problèmes, le logiciel LibreOffice embarque son propre algorithme de compression et décompression.

Avec Python, la création d'un fichier zip avec l'ajout de fichiers ressemble beaucoup à la création d'un fichier avec l'ajout de contenu.

```
>>> import zipfile
>>> with zipfile.ZipFile('/tmp/test.zip', 'w') as f:
...     f.write('/tmp/fichier1.txt')
...     f.write('/tmp/fichier2.txt')
...
```

L'archive créée de cette manière n'est pas compressée :

```
$ ll /tmp/test.zip
-rw-r--r-- 1 sch sch 317 2011-09-30 15:29 /tmp/test.zip
```

Voici comment activer la compression :

```
>>> with zipfile.ZipFile('/tmp/test.zip', 'w') as f:
...     f.write('/tmp/fichier1.txt', compress_type=zipfile.ZIP_DEFLATED)
...     f.write('/tmp/fichier2.txt', compress_type=zipfile.ZIP_DEFLATED)
...
```

Et le résultat :

```
$ ll /tmp/test.zip
-rw-r--r-- 1 sch sch 297 2011-09-30 15:58 /tmp/test.zip
```

Par défaut, la compression est non activée, **zipfile.ZIP_STORED** étant l'option correspondante.

Le fait que le choix de compresser ou non est réalisé fichier par fichier au lieu de concerner l'ensemble de l'archive est inhabituel par rapport aux autres algorithmes de compression, mais peut servir à gagner du temps. Par exemple en décidant de ne compresser que les fichiers sur lesquels on peut vraiment gagner une marge, comme les fichiers textuels, ou en excluant les fichiers déjà compressés comme les images PNG, documents OpenDocument.

Pour lire un fichier zip, la technique est assez différente :

```
>>> with zipfile.ZipFile('/tmp/test.zip') as f:
...     for fileinfo in f.filelist:
...         print(f.read(fileinfo))
...
```

Le résultat est une séquence de bytes :

```
b'Premi\xc3\xa8re ligne\nSeconde ligne\n'
b'Premi\xc3\xa8re ligne\nDeuxi\xc3\xa8me ligne\nTroisi\xc3\xa8me ligne\n'
```

Si l'on souhaite n'extraire qu'un nombre précis de fichiers dont on connaît le nom, il est possible d'agir ainsi :

```
>>> with zipfile.ZipFile('/tmp/test.zip') as f:
...     for fileinfo in ['tmp/fichier1.txt', 'existepas.txt']:
...         try:
...             print(f.read(fileinfo))
...         except:
...             print("Ce fichier n'est pas présent dans l'archive")
...
b'Premi\xc3\xa8re ligne\nSeconde ligne\n'
Ce fichier n'est pas présent dans l'archive
```

Il faut penser à gérer le fait que le fichier puisse ne pas exister dans l'archive.

Comme toujours, il est particulièrement important de regarder un peu ce que contient une archive avant d'en extraire quoi que ce soit. La méthode **filelist** est à ce titre importante.

Voici comment obtenir d'autres informations sur une archive .zip :

```
>>> with zipfile.ZipFile('/tmp/test.zip') as f:
...     for info in f.infolist():
...         print("Fichier: %s" % info.filename)
...         print("Comment: %s" % info.comment)
...         print('Modified: %s' % datetime.datetime(*info.date_time))
...         print('System: %s (0 = Windows, 3 = Unix)' % info.create_system)
...         print('ZIP version: %s' % info.create_version)
...         print('Compressed: %s bytes' % info.compress_size)
...         print('Uncompressed: %s bytes' % info.file_size)
```

© Editions ENI - All rights reserved

```
...
Fichier: tmp/fichier1.txt
Comment: b''
Modified: 2011-09-30 13:16:12
System: 3 (0 = Windows, 3 = Unix)
ZIP version: 20
Compressed: 26 bytes
Uncompressed: 30 bytes
Fichier: tmp/fichier2.txt
Comment: b''
Modified: 2011-09-30 13:16:36
System: 3 (0 = Windows, 3 = Unix)
ZIP version: 20
Compressed: 33 bytes
Uncompressed: 49 bytes
```

L'algorithme de compression est géré par la librairie **zlib** qui peut également être utilisée directement sur un fichier.

2.9.5 Interface de haut niveau

Le module **shutil** propose une méthode de haut niveau permettant de créer une archive dans les règles de l'art en une seule ligne.
Pour cela, il faut auparavant réaliser un import :

```
>>> from shutil import make_archive
```

Puis définir ce que l'on veut archiver en positionnant **root_dir** et **base_dir**, puis le nom de l'archive :

```
>>> root_dir, base_dir, archive_name = '/', '/var/www/tests',
'tests.tar.bz2'
```

Maintenant, il ne reste plus qu'à créer l'archive :

```
>>> archive_path = make_archive(archive_name, 'bztar', root_dir, base_dir)
```

Et le résultat est l'endroit où l'on peut trouver son archive toute prête.
L'archive ainsi créée contient le répertoire **var** qui contient le répertoire **www** qui contient lui-même les éléments que l'on voulait archiver, c'est-à-dire le répertoire **tests** et son contenu.
Cela permet d'extraire le répertoire directement à la racine et de voir les fichiers prendre place au bon endroit. Le répertoire home de l'utilisateur est également la source de ce genre d'archives.
Le module permet de savoir quels sont les formats d'archives supportés :

```
>>> for format, name in get_archive_formats():
...     print('%10s: %s' % (format, name))
...
     bztar: bzip2'ed tar-file
     gztar: gzip'ed tar-file
       tar: uncompressed tar file
       zip: ZIP file
```

Les fonctions **register_archive_format** et **unregister_archive_format** permettent de rajouter des formats d'archives parmi la liste de ceux utilisables. Il faut alors fournir un moyen d'invoquer l'utilitaire de compression.
Il faut fournir la liste des extensions pour laquelle le format est valide.
De la même manière, il est possible de gérer un utilitaire de décompression avec **register_unpack_format** et **unregister_unpack_format** et la liste de ceux qui sont supportés est lisible avec la fonction **get_unpack_formats**.
Ceci permet de décompresser une archive et la procédure est encore plus simple que pour la compression parce qu'il y a encore moins de paramètres :

```
>>> from shutil import unpack_archive
>>> unpack_archive(archive_path, 'path/to/unpack')
```

Le premier paramètre est le nom de l'archive (celui retourné par **make_archive**) et le second paramètre est le répertoire dans lequel décompresser. Il s'agit du répertoire de base, celui correspondant à **root_dir** pour la compression.

3. Gestion d'un fichier

3.1 Changer les droits d'un fichier

Cette section est toujours relative à la manipulation d'un fichier, mais il y a une différence fondamentale avec ce qui précède. Ici, on n'est plus dans des opérations d'entrée/sortie sur un fichier, mais plutôt dans des opérations sur le système de fichiers.
Il est alors évident que cette fonctionnalité est intimement liée au système d'exploitation et il y en a deux catégories. Ceux s'appuyant sur une technologie Unix, que ce soit rigoureusement identique ou en y apportant une touche personnelle (GNU/Linux, FreeBSD, et donc Mac qui s'appuie sur FreeBSD) et le reste du monde, c'est-à-dire Windows.
Il est donc parfaitement logique de trouver ces fonctionnalités dans le module os. De plus, nous aurons également besoin du module **stat** qui renferme les différentes constantes permettant de décrire un droit sur un fichier.

```
>>> import os, stat
```

Le système d'exploitation Windows permet de marquer un fichier en lecture seule ou en lecture et écriture.
Voici comment positionner un fichier en lecture seule :

```
>>> os.chmod(filepath, stat.S_IREAD)
```

Voici comment le mettre en lecture et écriture :

```
>>> os.chmod(filepath, stat.S_IWRITE)
```

Toute autre option passée ne sera simplement pas prise en compte (pas d'erreurs).
Pour Unix, la liste des options est plus conséquente (http://docs.python.org/library/os.html#os.chmod). Ces options permettent de positionner ou retirer les droits de lecture, écriture, exécution pour l'utilisateur, le groupe et les autres. Les explications des options sont abordées dans la documentation officielle (http://docs.python.org/library/stat.html#stat.S_ISUID).

© Editions ENI - All rights reserved

Comment retenir ces sigles barbares ? L'idée, en réalité, est d'avoir un nom de constante court. Il faut commencer par retenir R pour lecture, W pour écriture et X pour exécution. Lorsqu'une seule consigne est donnée, trois lettres désignent le « qui » et si RWX est appliqué simultanément, une seule lettre est utilisée. Il s'agit de U ou USR pour le propriétaire, G ou GRP pour le groupe et O ou OTH pour les autres.
Ensuite, comment utiliser ces constantes ? L'idée est de décrire exactement les droits donnés à un fichier. Si l'on donne un droit, il est présent, si on ne le précise pas, il est absent. Chaque constante est positionnée sur un bit différent (voir extrait de code suivant), c'est-à-dire est une puissance de deux différente et ainsi, l'addition de constantes permet de définir avec précision un droit.
Voici un tableau récapitulatif des constantes disponibles :

Constantes des droits		Utilisateur	Groupe	Autres
		(USR, U)	(GRP, G)	(OTH, O)
Lecture	(R)	S_IR*USR*	S_IR*GRP*	S_IR*OTH*
Écriture	(W)	S_IW*USR*	S_IW*GRP*	S_IW*OTH*
Exécution	(X)	S_IX*USR*	S_IX*GRP*	S_IX*OTH*
Tous les droits	(RWX)	S_IRWX*U*	S_IRWX*G*	S_IRW

```
>>> stat.S_IREAD, stat.S_IWRITE, stat.S_IREAD + stat.S_IWRITE,
stat.S_IREAD | stat.S_IWRITE (256, 128, 384, 384)
```

La séquence d'exemples suivante a été réalisée directement en console pour bien comprendre ce qu'il se passe et voir les conséquences (à l'aide de l'option **-c**).
D'abord, je crée un fichier en console et je regarde ses droits :

```
$ touch /tmp/test.txt
$ ll /tmp/test.txt
-rw-r--r-- 1 sch sch 0 2011-09-30 11:09 /tmp/test.txt
```

Puis, avec Python, je modifie ces droits :

```
$ python3 -c 'import os, stat; os.chmod("/tmp/test.txt", stat.S_IWOTH)'
$ ll /tmp/test.txt
--------w- 1 sch sch 0 2011-09-30 11:09 /tmp/test.txt
```

Et maintenant, j'utilise une combinaison de constantes pour donner tous les droits au propriétaire, le droit de lecture et d'écriture au groupe et le droit de lecture aux autres :

```
$ python3 -c 'import os, stat; os.chmod("/tmp/test.txt",
stat.S_IRWXU + stat.S_IRGRP + stat.S_IWGRP + stat.S_IROTH)'
$ ll /tmp/test.txt
-rwxrw-r-- 1 sch sch 0 2011-09-30 11:09 /tmp/test.txt*
```

Il est en réalité possible de procéder de différentes manières. Lister les droits :

```
sch@portableboulot:~$ python3 -c 'import os, stat;
os.chmod("/tmp/test.txt", stat.S_IRUSR + stat.S_IWUSR)'
sch@portableboulot:~$ ll /tmp/test.txt
-rw------- 1 sch sch 0 2011-09-30 11:09 /tmp/test.txt
```

Ou mettre tous les droits et en retirer un :

```
sch@portableboulot:~$ python3 -c 'import os, stat;
os.chmod("/tmp/test.txt", stat.S_IRWXU - stat.S_IXUSR)'
sch@portableboulot:~$ ll /tmp/test.txt
-rw------- 1 sch sch 0 2011-09-30 11:09 /tmp/test.txt
```

Attention cependant aux opérations effectuées. Ici, deux constantes identiques sont ajoutées l'une à l'autre et du coup, on change de bit et donc de droit :

```
sch@portableboulot:~$ python3 -c 'import os, stat;
os.chmod("/tmp/test.txt", stat.S_IWUSR + stat.S_IWUSR)'
sch@portableboulot:~$ ll /tmp/test.txt
-r-------- 1 sch sch 0 2011-09-30 11:09 /tmp/test.txt
```

Au lieu de donner le droit d'écriture sur le fichier, c'est le droit de lecture qui est accordé.
Les systèmes Unix possèdent des liens symboliques et changer les droits sur le lien n'impacte pas ceux de la cible du lien. Aussi, pour accéder à la ressource par le lien faut-il avoir les droits sur le lien et sur la ressource ciblée. Il est possible de modifier le lien et la cible en utilisant **`os.lchmod`** pareillement à **`os.chmod`**.
Il est à noter qu'il existe également **`os.fchmod`** qui réalise la même opération en prenant en paramètre non pas le chemin du fichier, mais un descripteur sur le fichier.

3.2 Changer de propriétaire ou de groupe

Cette notion n'a de sens que pour les systèmes Unix. Sur ces derniers, un fichier est possédé par un utilisateur et un groupe. Chaque utilisateur est identifié par un nombre, de même que les groupes. Il y a deux fonctions dont les signatures sont :

```
os.chown(filepath, uid, gid)
os.fchown(filedescriptor, uid, gid)
```

Il est à noter que tous les paramètres sont obligatoires, mais que si l'on ne souhaite pas modifier le groupe ou l'utilisateur, il suffit de remplacer son identifiant par -1.
La question est comment connaître les identifiants des groupes et utilisateurs ? La réponse est que ceci relève du domaine de compétence de l'administrateur système qui peut faire le lien en allant chercher dans le système ces informations :

```
$ getent passwd
$ getent group
```

Il est également possible de réaliser des scripts pour retrouver des informations à propos des groupes en analysant des données obtenues en lançant des commandes externes, mais les résultats sont dépendants de la machine et de son paramétrage.
Python propose cependant quelques méthodes permettant d'avoir des informations sur l'utilisateur et le groupe qui l'exécute et la liste des groupes concernés :

```
>>> os.getuid()
1000
>>> os.getuid()
1000
>>> os.geteuid()
1000
```

© Editions ENI - All rights reserved

```
>>> os.getgid()
1000
>>> os.getegid()
1000
>>> os.getgroups()
[4, 20, 24, 46, 112, 120, 122, 1000]
```

3.3 Récupérer des informations sur un fichier

Il est possible de récupérer tout type d'information sur un fichier :

```
>>> import os
>>> os.stat('/tmp/test.txt')
posix.stat_result(st_mode=33024, st_ino=168, st_dev=2056, st_nlink=1,
st_uid=1000, st_gid=1000, st_size=0, st_atime=1317373751,
st_mtime=1317373751, st_ctime=1317375012)
```

La liste des données présentée est expliquée dans la documentation officielle (http://docs.python.org/library/os.html#os.stat). Ce qui est important par rapport à ce que l'on a vu précédemment est le premier attribut qui est le mode (à travailler avec les constantes du module **stat**) et les arguments uid et gid qui sont respectivement l'identifiant de l'utilisateur qui a lancé le programme courant et celui du groupe auquel appartient cet utilisateur.

Il y a aussi les dates du dernier accès (**atime**) et de la dernière modification (**mtime**) exprimées en secondes depuis 1970 et utilisables via un objet **datetime**.

Pour convertir un tel nombre en date, il faut procéder ainsi :

```
>>> from datetime import datetime
>>> datetime.fromtimestamp(1317375012)
datetime.datetime(2011, 9, 30, 11, 30, 12)
```

La liste des attributs est dépendante du système d'exploitation, utilisant l'ensemble du spectre fonctionnel de celui-ci.

À noter qu'il existe également **os.lstat** qui permet de gérer les liens symboliques. En effet, **os.stat** « suit » automatiquement ces liens en les gérant comme s'ils étaient la cible elle-même. Il devient alors possible d'obtenir des informations sur le lien lui-même.

3.4 Supprimer un fichier

Supprimer un fichier relève là encore de la gestion de fichier, il ne s'agit pas de gérer des flux d'entrée/sortie. La fonction correspondante fait donc partie du module **os** et s'utilise trivialement.

```
$ python3 -c 'import os; os.remove("/tmp/test.txt")'
$ ll /tmp/test.txt
ls: impossible d'accéder à /tmp/test.txt: Aucun fichier ou dossier
de ce type
```

Python offre donc toutes les fonctionnalités nécessaires pour gérer des fichiers.

4. Alternatives simples à des commandes bash usuelles

4.1 Répertoires

Python permet de parcourir des répertoires, d'en créer, d'en supprimer et de positionner des droits. On utilise toujours le même module :

```
>>> import os
```

Voici comment se positionner dans ce dossier (équivalent de **cd**) :

```
>>> os.chdir('test')
```

Et comment savoir dans quel dossier on est positionné (équivalent de **pwd**) :

```
>>> os.getcwd()
'/home/sch/Documents/Bouquin_Python/exemples/14/test'
```

Créer un dossier est tout aussi aisé (équivalent de **mkdir**) :

```
>>> os.mkdir('rep1')
```

On peut aussi créer une arborescence (équivalent de **mkdir -p**) :

```
>>> os.mkdir('rep1', 'rep2', 'rep3')
```

Pour créer une arborescence de dossiers, il y a également une solution toute faite :

```
>>> os.makedirs(os.sep.join(['rep2', 'test', '14']))
```

Voici comment parcourir l'arborescence ainsi créée (équivalent de **ls -R**) :

```
>>> for cur, dirs, files in os.walk(os.curdir):
...     for name in (os.path.join(cur, d) for d in dirs):
...         print('%10d %s' % (os.path.getsize(name), name))
...
         0 ./rep1
         0 ./rep2
         0 ./rep2/test
         0 ./rep2/test/14
```

Cette partie mérite un peu plus d'explications. La fonction **os.walk** renvoie un générateur qui parcourt l'arborescence des fichiers et dossiers :

```
>>> for cur, dirs, files in os.walk(os.curdir):
...     print('Dans %s, il y a %d dossiers et %d fichiers' % (cur,
len(dirs), len(files)))
...     if len(dirs) > 0:
...         print('\t> Dossiers %s' % os.pathsep.join(dirs))
...     if len(files) > 0:
...         print('\t> Fichiers %s' % os.pathsep.join(files))
...
Dans ., il y a 2 dossiers et 0 fichiers
   > Dossiers rep1:rep2
Dans ./rep1, il y a 0 dossiers et 0 fichiers
Dans ./rep2, il y a 1 dossiers et 0 fichiers
   > Dossiers test
```

© Editions ENI - All rights reserved

```
Dans ./rep2/test, il y a 1 dossiers et 0 fichiers
   > Dossiers 14
Dans ./rep2/test/14, il y a 0 dossiers et 0 fichiers
```

Ainsi, le générateur renvoie un résultat par dossier traversé et il traverse d'abord tous les dossiers frères avant d'aller en profondeur. Le premier élément est ce dossier et les deux autres la liste des fichiers et dossiers de premier niveau.
Si l'on ne veut pas parcourir en profondeur, il faut procéder ainsi :

```
>>> current, dirs, files = next(os.walk(os.curdir))
```

On profite donc, encore une fois, de la souplesse des types de données Python, parce qu'en procédant ainsi, on n'utilise le générateur qu'une seule fois et l'on évite d'aller calculer les autres données qui ne serviront pas.
Voici le résultat :

```
>>> print(current)
.
>>> print(dirs)
['rep1', 'rep2']
>>> print(files)
[]
```

Notons que Python 3.5 introduit une nouvelle fonctionnalité :

```
>>> generateur_contenu_repertoire =  os.scandir()
```

Ce générateur renvoie des objets de type DirEntry (https://docs.python.org/3/library/os.html#os.DirEntry) qui sont utilisables ainsi :

```
>>> exemple = next(generateur_contenu_repertoire)
>>> exemple.name
'README'
>>> exemple.path
'./README'
>>> exemple.is_dir()
False
>>> exemple.is_file()
True
>>> exemple.is_symlink()
False
>>> exemple.stat()
os.stat_result(st_mode=33188, st_ino=10150542, st_dev=2057, st_nlink=1,
st_uid=1000, st_gid=1000, st_size=275, st_atime=1457706204,
st_mtime=1338896182, st_ctime=1457706204)
>>> exemple.inode()
10150542
```

Il s'agit là d'un excellent moyen d'itérer sur les fichiers et dossiers du répertoire courant et cette méthode est à la fois plus facile à appréhender que **`os.walk`** et plus performante que **`os.listdir`**.

Il est également possible de déplacer un répertoire (équivalent à **mv**) :

```
>>> os.rename(os.sep.join(['rep2', 'test']), os.sep.join(['rep2',
'test2']))
```

L'arborescence des parents de la destination doit exister, sinon, il faut utiliser :

```
>>> os.rrenames(os.sep.join(['rep2', 'test2']),
os.sep.join(['rep2', 'test3', '14']))
```

Un répertoire vide peut être supprimé (équivalent de **rmdir**) :

```
>>> os.rmdir('rep1')
```

Une chaîne de répertoires peut également être supprimée (équivalent de **rm -r**):

```
>>> os.removedirs(os.sep.join(['rep2', 'test', '14']))
```

Si le répertoire le plus profond existe et est vide, il est supprimé. Sinon, une exception est levée. On s'intéresse alors de proche en proche aux répertoires parents. S'il est vide, il est supprimé, sinon, l'action s'interrompt silencieusement.
Voici le résultat :

```
>>> print(next(os.walk(os.curdir)))
('.', [], [])
```

Enfin, pour connaître la taille d'un répertoire, il faut réaliser un petit script. En effet, cette donnée n'est pas portée par le répertoire lui-même, mais il s'agit de la somme de la taille de tous les fichiers qu'il contient. On va donc parcourir récursivement le répertoire et travailler sur ces fichiers (équivalent de **du -sk**) :

```
>>> size = 0
>>> for current, dirs, files in os.walk(path):
>>>     for f in files:
>>>         size += os.path.getsize(os.path.join(root, f))
```

De la même manière, si l'on veut des niveaux de détails plus élevés correspondant à des options des commandes bash, un petit script suffit et ce dernier peut être capitalisé dans une fonction pour être facilement applicable et plusieurs de ces fonctions peuvent être capitalisées dans un module.
Enfin, il existe également **os.expanduser** qui utilise la tilde désignant le répertoire home de l'utilisateur courant.

4.2 Fichiers

On a déjà vu comment manipuler un fichier, mais il existe quelques adjonctions permettant d'effectuer en Python des opérations naturelles de Bash.
Voici pour commencer une petite astuce qui consiste à réaliser l'équivalent d'un **touch** dans un terminal avec le résultat :

```
>>> with open('test.txt', 'a') as f:
...     pass
...
>>> print(next(os.walk(os.curdir)))
('.', [], ['test.txt'])
```

© Editions ENI - All rights reserved

Les commandes vues pour le renommage d'un répertoire fonctionnent également pour celui d'un fichier.
Voici un exemple qui permet de créer artificiellement un fichier pour mettre en évidence la manière d'obtenir la taille d'un fichier déjà vue pour celle du répertoire :

```
>>> with open('data.txt', 'a') as f:
...     f.write(os.linesep.join(['Donnees'*50]*100))
...
35099
>>> current, dirs, files = next(os.walk(os.curdir))
>>> for f in files:
...     print('%10d %s' % (os.path.getsize(f), f))
...
         0 test.txt
     35099 data.txt
```

Il est évident que **cat** peut se réaliser ainsi :

```
>>> with open('data.txt', 'r') as f:
...     print(f.readlines())
...
```

La fonctionnalité **head** peut se voir ainsi :

```
>>> with open('data.txt', 'r') as f:
...     for i in range(10):
...     print(f.readlines())
...
```

Il en est à peu près de même pour **tail** :

```
>>> from collections import deque
>>> with open('data.txt', 'r') as f:
...     for l in dequee(f.readlines(), 10):
...     print(l)
...
```

Il faut noter l'utilisation de **deque** qui permet de ne pas charger toutes les lignes pour ne garder que les dernières, mais de prendre les premières lignes si elles existent et d'évacuer la première de la liste dès qu'une nouvelle arrive (fonctionnement FIFO). Lorsque l'une des deux fonctionnalités est appliquée à d'autres données que des fichiers via un pipe en bash, Python se contente d'appliquer par une manipulation similaire sur l'objet Python représentant la donnée.
On reste ici dans des fonctionnalités bas niveau.

4.3 Module de haut niveau

Comme cela peut être constaté, il n'y a pas d'équivalent à la commande **cp** qui a été présentée jusqu'ici. On pourrait imaginer ouvrir un fichier en lecture, un autre en écriture et réaliser une copie, puis positionner des droits.

Préparons un fichier pour l'exemple avec une génération de contenu et en bash :

```
sudo chown sch:testeur datas.txt
$ chmod g+w datas.txt
$ chmod o+x \n datas.txt
$ ll \n datas.txt
-rw-rw-r-x 1 sch testeur 35099 2011-10-30 11:45 datas.txt*
```

On a donc créé sans Python un fichier avec des droits définis. Notez que l'utilisateur courant n'avait pas le droit d'effectuer le changement d'utilisateur.
Voici ce dont on a besoin pour copier un fichier :

```
>>> import os
>>> src, dst = 'datas.txt', 'datas2.txt'
```

Voici un algorithme qui copierait le fichier (alternative à **cp**) :

```
>>> with open(src, 'r') as fr, open(dst, 'a') as fw:
...     fw.write(fr.read())
...
35099
```

Or, les métadonnées du fichier ne sont pas préservées, le fichier est copié et porte les métadonnées relatives à celui qui a copié. Ce qui est intéressant est de proposer une copie en préservant ces métadonnées (alternative à **cp -p**) :

```
>>> import os
>>> src, dst = 'datas.txt', 'datas2.txt'
>>>
>>> with open(src, 'r') as fr, open(dst, 'a') as fw:
...     fw.write(fr.read())
...     stat = os.stat(src)
...     os.chmod(dst, stat[0])
...     os.chown(dst, stat[4], stat[5])
...
35099
```

Avec l'utilisateur courant, on obtient ce message d'erreur :

```
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
OSError: [Errno 1] Operation not permitted: 'datas2.txt'
```

Si on lance Python en superutilisateur (**sudo python3**), alors cela fonctionne :

```
-rw-rw-r-x  1 sch testeur 35099 2011-10-30 11:55 datas2.txt*
-rw-rw-r-x  1 sch testeur 35099 2011-10-30 11:45 datas.txt*
```

Il y a cependant une différence essentielle avec la commande **cp -p**, c'est que les données sur la date de création et de dernière modification ont été modifiées. Il faudrait donc fournir un effort supplémentaire.
Heureusement, Python propose une interface de haut niveau permettant de réduire le travail à effectuer et qui gère correctement la copie de fichier et de répertoires avec leur contenu.

© Editions ENI - All rights reserved

Pour cela, il faut utiliser **shutil** dont la documentation est assez complète :
http://docs.python.org/library/shutil.html?highlight=shutil#module-shutil
Voici comment reprendre le script précédent (équivalent à **cp**) :

```
>>> shutil.copyfile(src, dst)
```

Et en préservant les données :

```
>>> shutil.copy2(src, dst)
```

Ce qui est équivalent à :

```
>>> shutil.copy(src, dst)
>>> shutil.copystat(src, dst)
```

On a là une vraie alternative aux commandes bash (et **cp** n'est pas la moindre).
Il existe une fonction intermédiaire entre **copyfile** et **copy2** qui est **copy** et qui reprend le propriétaire, le groupe, mais pas les dates de création et de dernière modification.
Il existe également un moyen simple de modifier les permissions d'un fichier en les reprenant d'un autre avec **copymode**.
Enfin, il y a une autre alternative au déplacement de fichier (équivalent de **mv**) :

```
>>> shutil.move('datas.txt', 'datas8.txt')
```

Et enfin, la fonctionnalité la plus complexe, la copie ou la suppression d'un répertoire (**cp -r**) :

```
>>> shutil.copytree('test', 'test2')
```

Ceci s'effectue en une seule ligne et reste vraiment accessible.
De plus, la fonctionnalité permet de rajouter un patron de fichiers à ignorer de manière à les exclure de la copie :

```
>>> shutil.copytree('test', 'test3',
ignore=shutil.ignore_patterns('*.pyc'))
```

4.4 Recherche d'un fichier

Python offre la possibilité de rechercher un fichier par rapport à une règle de nommage qui peut prendre en compte les expressions régulières :

```
>>> from glob import glob
>>> glob('./d*[0-9].txt')
['./datas2.txt', './datas3.txt', './datas4.txt', './datas8.txt']
```

Cela est une alternative à la commande **find** qui reste facile d'utilisation. Il existe également le module **fnmatch** qui peut présenter un intérêt.

5. Exécuter des commandes externes

5.1 Exécuter et afficher le résultat

Python propose plusieurs moyens de réaliser des appels à des commandes systèmes et de récupérer le résultat.
Le module **os** comporte des outils de bas niveau très pointus, mais destinés à ceux qui connaissent déjà les grands principes de la programmation système.
Pour les besoins de ce chapitre, nous voulons être capables de lancer dans la console Python des commandes que l'on pourrait lancer dans un terminal.
Le module utilisé est le suivant :

```
>>> import subprocess
```

Voici comment procéder :

```
>>> retcode = subprocess.call('ls')
data.txt  test.txt
```

Le résultat de la commande est affiché à l'écran et le code de retour est récupéré au retour de la fonction call :

```
>>> print(retcode)
0
```

Tout programme qui se termine correctement renvoie un 0, sinon il renvoie un chiffre qui correspond à un code d'erreur.
Lorsque l'on veut passer des paramètres, il faut alors différencier la commande et les paramètres :

```
>>> retcode = subprocess.call(['ls', '-l'])
total 128
-rw-r--r-- 1 sch sch 35099 2011-10-22 17:56 data.txt
-rw-r--r-- 1 sch sch     0 2011-10-22 17:38 test.txt
```

Les paramètres doivent être passés les uns après les autres dans la liste comme dans cet exemple :

```
>>> retcode = subprocess.call(['ls', '-l', '-a'])
total 128
drwxr-xr-x 2 sch sch     0 2011-10-22 17:56 .
drwxr-xr-x 3 sch sch     0 2011-10-22 16:49 ..
-rw-r--r-- 1 sch sch 35099 2011-10-22 17:56 data.txt
-rw-r--r-- 1 sch sch     0 2011-10-22 17:38 test.tx
```

La fonction **check_call** réalise exactement la même chose, mais permet de générer une exception en cas de retour non nul, ce qui permet d'utiliser un système de gestion d'exception dans des méthodes utilisant de tels appels :

```
>>> retcode = subprocess.check_call(['ls', ''])
ls: impossible d'accéder à : Aucun fichier ou dossier de ce type
Traceback (most recent call last):
>>> retcode
2
```

© Editions ENI - All rights reserved

Les deux fonctions génèrent une erreur si la commande n'existe pas.

5.2 Exécuter et récupérer le résultat

Au lieu d'afficher à l'écran les données retournées par la commande, il est possible de les récupérer dans une variable qui peut être utilisée dans un second temps :

```
>>> retcode, result = subprocess.getstatusoutput('ls')
>>> retcode
0
>>> result.split(os.linesep)
['data.txt', 'test.txt']
```

Il existe aussi **`subprocess.getoutput`** pour n'avoir que la sortie, mais obtenir le code de retour est judicieux.
Voici un exemple plus technique permettant de sauvegarder une base de données. On déclare les paramètres utiles et on construit la commande :

```
>>> user, mdp, base = 'root', 'motdepasse', 'core'
>>> params = ['mysqldump', '--default-character-set=utf8', '--comments',
'--no-data', '--add-drop-database', '--add-drop-table',
'--result-file=struct.sql', '--user=%s' % user, '--password=%s' % mdp,
'--databases', base]
```

Une fois assemblée, la commande ressemble à ceci :

```
>>> print(' '.join(params))
mysqldump --default-character-set=utf8 --comments --no-data
-add-drop-database --add-drop-table --result-file=struct.sql
--user=root --password=motdepasse --databases core
```

Elle est lancée ainsi :

```
>>> retcode = subprocess.call(params)
```

Cet export crée un fichier **`struct.sql`** qui ne contient que la déclaration de la structure des tables. Voici un autre exemple monté différemment qui permet d'avoir le contenu de ces mêmes tables (les enregistrements).

```
>>> pattern='mysqldump --comments --skip-extended-insert
--complete-insert --no-create-db --no-create-info --skip-add-locks
--skip-disable-keys --result-file=data.sql --user=%(user)s
--password=%(mdp)s %(base)s'
>>> params = (pattern % {'user': 'root', 'mdp': 'motdepasse',
'base': 'core'}).split(' ')
```

Voici le résultat, et il s'exécute exactement de la même manière :

```
>>> params
['mysqldump', '--comments', '--skip-extended-insert', '--complete-insert',
'--no-create-db', '--no-create-info', '--skip-add-locks',
'--skip-disable-keys', '--result-file=data.sql', '--user=root',
'--password=motdepasse', 'core']
```

Cette commande crée un fichier **data.sql** qui contient un **insert** par enregistrement. Ceci n'est absolument pas performant pour l'injection de données, mais cela permet d'effectuer des différentiels rapidement entre deux fichiers réalisés à des dates différentes, ce qui sera présenté dans une prochaine section.
Les fichiers ainsi obtenus à deux dates différentes peuvent être comparés et cela va être également présenté.

5.3 Pour Python 3.5 et supérieur

Python 3.5 introduit une nouvelle interface de plus haut niveau : **subprocess.run**. Elle permet de gérer l'ensemble des cas d'utilisation précédents.
Ainsi, pour appeler une commande externe et afficher son résultat :

```
>>> import subprocess
>>> subprocess.run(["ls", "-al"], stdout=sys.stdout)
```

Il est possible d'utiliser **shell=True** pour passer la commande sous la forme d'une chaîne de caractères, d'utiliser **check=True** pour lancer une exception en cas de retour en erreur de la commande et d'écrire le résultat dans un fichier, en procédant ainsi :

```
>>> import sys
>>> with open("test.txt", "w") as f:
...     subprocess.run("ls -al", shell=True, check=True, stdout=f)
...
```

Cette interface plus universelle devrait permettre d'éviter de retenir trop de noms de méthodes différents.

6. IHM système

6.1 Présentation

En programmation système, l'application interagit principalement avec le système. Elle va donc utiliser les outils du système, ce qui est vrai en particulier pour l'interface homme-machine, c'est-à-dire pour la manière de donner de l'information à l'utilisateur et de lui en demander.
Dans ce contexte, l'utilisateur a accès à un terminal. La première chose qu'il va faire est de lancer l'application. Ceci permet de passer des arguments à l'application, comme le montrent ces exemples classiques :

```
$ top
$ cp fichier1 fichier 2
$ cp -t directory fichier1 fichier2 fichier3
```

En Python, ces arguments sont disponibles dans **sys.argv** et le premier d'entre eux est le nom de l'exécutable si l'on est en console ou celui du fichier exécuté. On peut le visualiser en écrivant ceci dans un fichier :

```
import sys
for arg in sys.argv:
    print(arg)
```

© Editions ENI - All rights reserved

Et voici comment on exécute ce programme :

```
$ python3 argv.py -t dest f1 f2 f3 --test
argv.py
-t
dest
f1
f2
f3
--test
```

La gestion manuelle de ces arguments pourrait s'avérer extrêmement compliquée sans l'aide du module de haut niveau **argparse**. Ce module permet également de générer une documentation, gérer les erreurs et éventuellement une logique complexe derrière des arguments.

Une fois que l'application se termine, alors elle doit renvoyer un nombre entier positif : soit 0 si elle s'est terminée correctement, soit un nombre strictement positif. La valeur de ce nombre n'a pas d'importance pour le système, mais vous pouvez lui donner une signification en attribuant une valeur à une erreur spécifique.

Pour quitter un programme Python, soit vous laissez s'exécuter la toute dernière ligne du programme et il rend la main avec **0**, soit vous utilisez **`sys.exit(42)`** par exemple.

Enfin, pendant l'utilisation du programme, vous pourrez communiquer avec l'utilisateur via les trois canaux de communication que le système met en place : l'entrée standard, la sortie standard et la sortie d'erreur.

6.2 Travailler avec des arguments

Habituellement, lorsqu'on lance un programme depuis un terminal, quel que soit le type de ce programme, il est possible de passer des arguments. Ceux-ci sont alors traités et le programme doit les parser et leur donner une signification.

Or, les programmes complexes peuvent avoir plusieurs options, potentiellement plusieurs paramètres qui peuvent être ordonnés ou préfixés. Il suffit d'effectuer un man sur une page d'une commande bash un peu complexe pour avoir une idée de la complexité que cela peut représenter :

```
$ man find
```

La base de la réalisation d'un programme système efficace est de pouvoir recevoir des arguments d'une manière standard et pouvoir les traiter d'une façon professionnelle. L'alternative est un traitement classique d'une simple liste et une litanie de boucles conditionnelles permettant de leur donner une signification tout en vérifiant manuellement que l'on utilise des options qui sont bien compatibles entre elles.

De plus, il faut être capable de fournir une documentation claire pouvant apparaître dans une page d'aide similaire à celle d'un man (par l'utilisation maintenant classique de l'option **`--help`**).

Pour cela, Python dispose d'un outil de haut niveau qui permet de décrire les arguments attendus ou un ensemble de possibilités et qui peut, en fonction de ce qu'il reçoit, rediriger vers un cas ou un autre.

La facilité de mise en œuvre par rapport à la complexité de la problématique est une excellente illustration des qualités de Python et permet de réaliser un interfaçage avec la ligne de commande qui soit vraiment très sûr.
Voici comment initialiser un tel parseur :

```
import argparse
parser = argparse.ArgumentParser(
  prog='./testeur.py',
  description='''Testeur pour arguments''',
  epilog='''Exemple du chapitre 14'''
)
```

Les arguments donnés sont des renseignements utiles mis en forme lorsque l'on rajoute l'option **--help** à la commande.
Lorsque la commande effectue plusieurs actions de nature distincte, il faut utiliser un sous-parseur par action possible :

```
subparsers = parser.add_subparsers(help='commands')
test_parser = subparsers.add_parser(
  'test',
  description='Test du parseur',
  help='Permet de tester le parseur',
)
```

Ici, on déclare que cette option est prise en compte lorsque le premier argument après le nom du programme est le mot « test ». Le reste des arguments est utile pour l'aide associée et il ne faut pas hésiter à être précis.
Il faut maintenant relier ce cas d'utilisation à l'exécution d'une fonction :

```
test_parser.set_defaults(func=test)
```

Cette fonction a une signature particulière (l'exemple est volontairement simpliste) :

```
def test(namespace):
  return namespace
```

Elle prend en paramètre un espace de nommage qui contient la fonction à utiliser ainsi que les éventuels arguments qui sont ceux passés après le nom du programme et le nom de l'action à effectuer liée au sous-parseur.
L'exécution du programme donnerait :

```
$ ./testeur.py test
Namespace(func=<function test at 0x2040ea8>)
```

Prenons maintenant un cas un peu plus pratique. Voici un exemple simpliste de données :

```
datas = {
  'carrés': {'%s' % x:(x+1)**2 for x in range(20)},
  'cube': {'%s' % x:(x+1)**3 for x in range(20)},
}
```

Les deux fonctions suivantes utilisent ces données minimalistes pour agir dessus :

```
def available_keys(namespace):
   return list(datas.keys())
```

© Editions ENI - All rights reserved

```
def get(namespace):
  return '%s:%s' % (
    datas['carrés'].get(namespace.value),
    datas['cube'].get(namespace.value))
```

Dans le premier cas, on ne se sert pas de l'espace de nommage, il n'y a donc pas d'interactions entre d'éventuels arguments supplémentaires et le résultat.
Dans le second cas, on se sert d'un argument supplémentaire qui est « value » et qui provient de l'espace de nommage, donc de la ligne de commande. Ce dernier est simplement restitué à la fonction purement Python par un attribut de l'espace de nommage.
Ces fonctions sont donc des éléments intermédiaires entre le parseur et les fonctionnalités purement Python.
Voici un parseur qui permet de travailler avec ces fonctions :

```
def getParser():
   parser = argparse.ArgumentParser(
        prog='./testeur.py',
        description='''Testeur pour arguments''',
        epilog='''Exemple du chapitre 14'''
   )

   subparsers = parser.add_subparsers(help='commands')
```

Voici les parties spécifiques utilisant la première fonction décrite, celle qui ne prend pas d'arguments :

```
   list_parser = subparsers.add_parser(
        'list',
        aliases=['l'],
        description='liste des données disponibles',
        help='Permet de récupérer la liste des fonctionnalités
disponibles',
   )
   list_parser.set_defaults(func=available_keys)
```

Voici à quoi ressemblera l'appel du programme qui passera par ce sous-parseur pour utiliser la fonction :

```
$ ./testeur.py list
['carrés', 'cube']
```

Dans la déclaration du sous-parseur, on a également défini un alias, voici comment l'utiliser dans la ligne de commande :

```
$ ./testeur.py l
['carrés', 'cube']
```

La partie qui vient montre comment prendre en compte un argument pour le restituer à la seconde fonction qui en a besoin :

```
   get_parser = subparsers.add_parser(
        'get',
```

```
        description='une valeur',
        help='Permet de donner toutes les valeurs relatives à l\'argument'
    )
    get_parser.add_argument(
        'value',
        help='valeur'
    )
    get_parser.set_defaults(func=get)
```

Voici ce à quoi correspond l'appel du programme correspondant :

```
$ ./testeur.py get 4
25:125
```

À la fin de la fonction de création du parseur, on le retourne :

```
    return parser
```

Maintenant, voici comment utiliser le parseur dans le module principal Python pour lui demander de parser les arguments et pour lancer la fonction adéquate :

```
if __name__ == "__main__":
    parser = getParser()
    args=parser.parse_args()
    print(args.func(args))
```

On réalise trois étapes : création du parseur, parsing des arguments, lancement de la bonne fonction, déterminée par les arguments reçus.
Le seul effort à fournir est de créer les fonctions qui lient une fonctionnalité Python et le namespace que l'on recevra du parseur et la création du parseur lui-même.
Mais le bon traitement des arguments est très loin d'être la seule utilité de ce parseur.
Ce qui est tout aussi important est l'aide que l'on doit fournir à l'utilisateur du programme et qui se trouve ici générée grâce à la structure que l'on donne au parseur ainsi qu'aux renseignements supplémentaires fournis (descriptions...).
Voici le résultat avec l'exemple conçu au-dessus et reprenant les trois sous-parseurs :

```
$ ./testeur.py -help
usage: ./testeur.py [-h] {test,list,l,get} ...

Testeur pour arguments

positional arguments:
  {test,list,l,get}  commands
    test             Permet de tester le parseur disponible
    list (l)         Permet de récupérer la liste des
                     fonctionnalités disponibles
    get              Permet de donner toutes les valeurs relatives
                     à l'argument

optional arguments:
  -h, --help         show this help message and exit

Exemple du chapitre 14
```

© Editions ENI - All rights reserved

Cet outil est une différence essentielle qui met Python sur un autre niveau par rapport à d'autres technologies.
Ce module couvre un enjeu essentiel de Python (d'ailleurs, il en remplace un autre qui en a lui-même remplacé un troisième, preuve que la créativité dans ce domaine est forte et les contributions importantes). Cela signifie également que la documentation officielle du module est particulièrement bien faite (http://docs.python.org/py3k/library/argparse.html).
En d'autres termes, il est vraiment utile de la consulter et de tester ce qui est proposé après avoir suivi l'introduction plus facilement abordable qui en a été faite ici. La seule difficulté réelle est l'utilisation du paramètre action (http://docs.python.org/py3k/library/argparse.html#action) qui est bien détaillée.
Elle offre une très large gamme d'utilisation des arguments et permet d'avoir une ligne de commande qui soit sémantiquement porteuse d'informations et donc plus facile à utiliser pour celui qui veut arriver à un résultat avec notre programme.
L'argument choice est également particulièrement utile puisqu'il permet de restreindre une valeur à une liste prédéterminée. Enfin, la possibilité de créer des groupes d'arguments (http://docs.python.org/py3k/library/argparse.html#argument-groups) et des exclusions (http://docs.python.org/py3k/library/argparse.html#mutual-exclusion) complètent le dispositif et en font un outil particulièrement complet.

6.3 Fermeture du programme

Comme on l'a déjà dit un peu plus haut, pour quitter un programme Python, soit vous laissez s'exécuter la toute dernière ligne du programme et il rend la main avec **0**, soit vous utilisez **`sys.exit`** et vous lui donnez une valeur :

```
import sys
try:
    int(input("saisissez un nombre "))
except:
    sys.exit(1)
```

Voici comment mettre en évidence le retour de la fonction :

```
$ python3 quitter.py
saisissez un nombre 42
$ echo $?
0
```

Dans cet exemple, il n'y a pas eu d'exception et le programme s'est terminé car il n'y avait plus d'autres instructions à exécuter. La commande bash **`echo $?`** permet d'afficher la valeur de retour du programme.
Voici maintenant ce qu'il se passe lorsqu'il y a eu une exception :

```
$ python3 quitter.py
saisissez un nombre NON
$ echo $?
1
```

On a bien la valeur avec laquelle on a quitté notre programme.

Décider de quitter un programme peut se faire de n'importe quel endroit du code. Cependant, il se peut que l'on veuille faire certaines actions avant de quitter un programme. Pour éviter de répéter ces actions à tous les endroits où l'on quitte le programme, il est possible d'utiliser le module **atexit** :

```
import atexit
atexit.register(fonction_de_fin_de_programme)
```

Ce module permet de déclencher l'exécution de la fonction juste avant que le programme ne quitte.
Enfin, pour les utilisateurs de Windows, il est possible d'empêcher la console de quitter automatiquement à la fin de l'exécution de l'application ou du script :

```
import subprocess
subprocess.call('pause', shell=True)
```

Ceci peut s'avérer indispensable si l'on souhaite visualiser les sorties à la fin du programme.

6.4 Entrée standard

Nous venons de voir comment gérer des arguments passés à notre programme, ce qui est un outil essentiel pour la programmation système. L'autre manière de recevoir de l'information de l'utilisateur est l'entrée standard. Il s'agit d'un canal de communication qui permet à l'utilisateur de taper des caractères et au programme de les récupérer :

```
>>> data = input("Votre age: ")
```

Cette donnée sera comprise comme une chaîne de caractères. Selon votre souhait, vous devrez gérer la conversion de la donnée, en particulier les scénarios d'erreurs.

Remarque

En Python 2, il existe une fonction ***input****, mais celle-ci a une faille et il ne faut plus l'utiliser. Pour des raisons de compatibilité, elle n'a pas été supprimée, mais dépréciée et une nouvelle fonction* ***raw_input*** *a été créée. En Python 3, la fonction défectueuse a été purement et simplement supprimée et la fonction* ***raw_input*** *a été renommée* ***input****, permettant de revenir au nom original.*

Input utilise **sys.stdin**, mais en tant que développeur d'application, nous n'avons jamais à utiliser directement cet objet.
Il est aussi possible de faire en sorte de ne pas afficher la saisie, ce qui est utile pour la saisie d'un mot de passe :

```
>>> import getpass
>>> pwd = getpass.getpass("Veuillez saisir votre mot de passe:")
Veuillez saisir votre mot de passe:
>>> print(pwd)
secret
```

© Editions ENI - All rights reserved

À noter que ce module permet aussi de récupérer le nom de l'utilisateur courant :

```
user = getpass.getuser()
>>> print(user)
pythoniste
```

6.5 Sortie standard

Pour écrire dans la sortie standard, rien de plus simple :

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

On utilise ici implicitement **sys.sdtout**. Cet objet n'est quasiment jamais utilisé directement sauf si l'on veux gérer le tampon directement :

```
>>> sys.stdout.write(chaine)
>>> sys.flush()
```

Dans cet exemple, la méthode **flush** permet de forcer l'affichage, sans quoi l'affichage peut être décalé du temps réel.
Il est à noter que la fonction **print** a deux paramètres **end** et **sep** qui permettent de préciser comment séparer plusieurs variables dans l'affichage (par défaut un espace) et comment le terminer (par défaut un passage à la ligne).
Il existe également des caractères spéciaux qui permettent de toujours écrire au même endroit, de mettre à jour une ligne :

```
>>> for i in range(40):
...     print(f"\r[{'*' * i}{' ' * (39 - i)}]", end="")
...     time.sleep(0.1)
```

Cet exemple permet de construire une jauge de progression. À noter que pour faire cela, il y a un outil dédié, installable comme suit :

```
$ pip install progressbar
```

Voici un code assez direct pour avoir une jauge simpliste :

```
>>> import time
>>> from progressbar import ProgressBar
>>> progress = ProgressBar()
>>> for i in progress(range(40)):
...     time.sleep(0.1)
...
100% |#################################################
```

Il est aussi possible de faire quelque chose de plus évolué :

```
>>> import time
>>> from progressbar import ProgressBar
>>> from progressbar import (ProgressBar, Percentage, RotatingMarker,
                             FileTransferSpeed, Bar, ETA)
>>> def print_bar():
...     widgets = ["Avancement: ", Percentage(), " ",
                   Bar(marker=RotatingMarker()), " ",
                   ETA(), " ", FileTransferSpeed()]
```

```
...     pbar = ProgressBar(widgets=widgets, maxval=4000).start()
...     for i in range(4000):
...         pbar.update(1)
...         time.sleep(0.001)
...     pbar.finish()
...
>>> print_bar()
Avancement: 100% |#######################################| Time: 0:00:04
872.87
```

On voit ici à la fois la jauge, mais également son évaluation en pourcentage, une estimation du temps restant et une estimation du taux de transfert.

6.6 Sortie d'erreur

La sortie d'erreur est un canal de communication dédié à l'affichage des erreurs. Pour l'utiliser, il suffit de procéder ainsi :

```
>>> import sys
>>> print("Erreur", file=sys.stderr)
Erreur
```

Il s'agit d'un canal distinct de la sortie standard. Du point de vue d'un utilisateur et sans personnalisation du terminal, la différence entre les deux sorties peut être difficile à voir. Les messages s'affichent au même endroit et de la même couleur.
Voici un exemple de programme pour mettre cela en évidence :

```
import sys
print("Ceci est un message affiché dans la sortie standard")
print("Ceci est un message affiché dans la sortie d'erreur", file=sys.stderr)
```

Et voici ce que l'on obtient en exécutant un tel programme :

```
$ python3 test_sorties.py
Ceci est un message affiché dans la sortie standard
Ceci est un message affiché dans la sortie d'erreur
```

Pour mettre en évidence qu'il y a bien deux canaux, on peut procéder ainsi :

```
$ python3 test_sorties.py 1>out.txt 2>err.txt
```

Ici, on a redirigé la sortie standard vers un fichier **out.txt** et la sortie d'erreur vers **err.txt**. Il suffit maintenant d'afficher leur contenu :

```
$ cat out.txt
Ceci est un message affiché dans la sortie standard
$ cat err.txt
Ceci est un message affiché dans la sortie d'erreur
```

© Editions ENI - All rights reserved

6.7 Taille du terminal

Nous avons vu précédemment la manière de créer une jauge ou encore comment formater une chaîne de caractères pour qu'elle prenne un espace pré-déterminé. Pour utiliser au mieux tout l'espace mis à disposition par le terminal, il faut connaître sa taille :

```
>>> import shutil
>>> shutil.get_terminal_size()
os.terminal_size(columns=206, lines=55)
```

Lorsque cette commande a été tapée, le terminal était en plein écran. Le voici maintenant réduit :

```
>>> shutil.get_terminal_size()
os.terminal_size(columns=102, lines=25)
```

On peut l'exploiter ainsi :

```
>>> shutil.get_terminal_size().columns
206
>>> shutil.get_terminal_size().lines
55
```

Ceci est disponible à partir de Python 3.3. Pour les versions antérieures, il faudra écrire cette fonction :

```
>>> import subprocess
>>> import re
>>> def termsize():
...     output = subprocess.check_output(['/bin/stty', '-a'])
...     matched = re.search(
...         'rows\D+(?P<rows>\d+); columns\D+(?P<columns>\d+);',
...         output.decode('utf-8')
...     )
...     if matched:
...         return matched.group('columns'), matched.group('rows')
...     raise OSError('Bad response: %s' % (output))
...
```

Le résultat est un 2-uplet contenant le nombre de colonnes puis le nombre de lignes :

```
>>> termsize()
('206', '55')
```

7. Curses

Remarque

Ce chapitre cible en particulier les développeurs qui sont déjà à l'aise avec Python et habitués à la programmation système.

7.1 Présentation

La bibliothèque Curses permet le développement d'interfaces homme-machine textuelles avancées à haut niveau.

Elle se charge donc de prendre en compte les particularités de chaque terminal (Linux, Unix, Xterm ou rxvt) et offre toutes les fonctionnalités nécessaires pour contrôler finement mais simplement les interactions entre la machine et l'utilisateur.

Comme la plupart des bibliothèques graphiques (Gtk, Qt, wxWidgets) et Pygame, Curses se charge d'initialiser l'écran, de réaliser l'affichage et offre une boucle événementielle permettant d'écouter les actions de l'utilisateur.

Nous réaliserons ici une présentation rapide du module, en montrant comment faire les premiers pas avec cette bibliothèque.

7.2 Gérer le démarrage et l'arrêt

Le terminal fonctionne de manière classique. Il affiche ce que l'utilisateur saisit et réagit à ce qu'il demande. Lorsque Curses démarre, le terminal change de comportement pour offrir non plus quelque chose de linéaire, mais une interface prenant tout l'écran.

De même, lorsque l'application utilisant Curses se termine (normalement ou par une erreur), le terminal doit être restauré.

C'est ce que fait pour nous **`curses.wrapper.wrapper`**.

```
curses.wrapper.wrapper(fonction_application, arguments, ...)
```

La fonction **`application`** a pour premier argument l'objet représentant l'écran, appelé par convention **`stdscr`**.

```
def fonction_application(stdscr, arguments, …):
    pass
```

Ses autres arguments sont ceux qui ont été passés à la fonction **`curses.wrapper.wrapper`**.

À notre niveau, il nous suffit de savoir qu'il faut utiliser cette fonction, ce qui garantit le retour à un terminal fonctionnel même en cas d'erreur dans notre programme.

Pour aller plus loin, voici un équivalent simplifié :

```
import curses

# initialisation
stdscr = curses.initscr()  # Initialisation de l'écran
curses.start_color()       # Activation des couleurs
curses.noecho()            # Suppression de l'affichage des caractères
                           # saisis par l'utilisateur
```

© Editions ENI - All rights reserved

```
curses.cbreak()             # Chaque touche fait réagir l'application
stdscr.keypad(1)            # Ne plus prendre en compte les séquences
                            # multi-octets (Page Up ou Home, par exemple)
try:
    # Utilisation de la fonction application
    fonction_application(stdscr, arguments, …)
finally:
    # Rétablissement du terminal à l'état initial
    curses.nocbreak()
    stdscr.keypad(0)
    curses.echo()
    curses.endwin()         # Retour au terminal classique
```

Pour rendre le curseur invisible, on pourrait rajouter **`curses.curs_set(0)`**.
Le mode **`cbreak`** permet de ne pas attendre que l'utilisateur valide sa saisie (touche [Entrée]) et permet de faire en sorte que le programme réagisse à chaque appui de touche.
Une fois que les couleurs ont été activées, il est possible de savoir si le terminal les prend en compte en utilisant l'instruction **`curses.has_colors`**.

7.3 Affichage de texte

Comme pour une application graphique, il faut se préoccuper de définir le lieu où l'on souhaite positionner du texte.
La bibliothèque Curses a la particularité d'utiliser dans ses signatures **y**, puis **x**. Le point d'origine se trouve en haut à gauche de l'écran, l'axe **Y** descend vers le bas et l'axe **X** va vers la droite.
L'unité n'est pas le pixel, mais un emplacement pris par un caractère (le terminal utilise une police d'écriture à chasse fixe).
L'affichage d'un texte se fait très simplement ainsi :

```
stdscr.addstr("Origine")
stdscr.addstr(0, 8, "à droite de l'origine")
stdscr.addstr(1, 0, "en dessous de l'origine")
```

La fonction **`curses.refresh()`** permet de réaliser l'affichage.

7.4 Gestion des attributs

Les attributs permettent de personnaliser l'affichage. Il s'agit de nombres à la puissance de deux. Il est donc possible d'additionner ces attributs (ou d'utiliser le ET logique) :

```
>>> bin(A_REVERSE)          # 2**18
'0b1000000000000000000'
>>> bin(A_BOLD)             # 2**21
'0b100000000000000000000'
>>> bin(A_BOLD+A_REVERSE)   # Addition
'0b100100000000000000000'
>>> bin(A_BOLD|A_REVERSE)   # ET logique : Même effet que +
'0b100100000000000000000'
```

Voici comment utiliser des attributs :

```
stdscr.addstr(4, 8, "TEXT BOLD", A_BOLD)
stdscr.addstr(8, 16, "TEXT UNDERLINE", A_UNDERLINE)
stdscr.addstr(16, 16, "TEXT STANDOUT&REVERSE", A_BOLD+A_UNDERLINE)
```

7.5 Gestion des couleurs

Les couleurs ne sont pas gérées par tous les terminaux, auquel cas elles ne seront tout simplement pas utilisées et l'affichage restera noir et blanc.
Les couleurs fonctionnent comme les autres attributs :

```
>>> bin(color_pair(1))       # Paire de couleurs
0b100000000
```

Dans le cas contraire, il suffit d'initialiser les couleurs, puis de les utiliser :

```
curses.init_pair(2, curses.COLOR_BLUE, curses.COLOR_YELLOW)
stdscr.addstr(0, 78, "X", curses.color_pair(2))
```

Elles peuvent bien entendu être combinées avec les autres attributs :

```
curses.init_pair(1, curses.COLOR_RED, curses.COLOR_WHITE)
stdscr.addstr(21, 0, "Pour quitter, appuyer sur ", curses.color_pair(1))
stdscr.addstr(21, 26, "q", curses.color_pair(1)|curses.A_REVERSE)
```

7.6 Gestion des événements

Enfin, pour pouvoir interagir avec l'utilisateur, le programme doit se mettre à son écoute. La réponse usuelle à ce type de problématique consiste à utiliser les possibilités offertes par la programmation événementielle :

```
while True:                     # Boucle infinie
    event = stdscr.getch()      # Capture des événements
    if event == ord("q"):       # Si l'on appuie sur la touche 'q'
        break
```

Vous avez ici toutes les clés pour vous permettre de créer un programme Curses. Un exemple concret sera présenté dans la dernière partie du livre.

8. Accès aux périphériques

Remarque

Ce chapitre cible en particulier les développeurs qui sont déjà à l'aise avec Python, habitués à la programmation système et qui ont déjà des connaissances sur D-Bus.

© Editions ENI - All rights reserved

8.1 Introduction

Il existe de nombreux moyens d'accéder aux périphériques, dont beaucoup sont à l'heure actuelle dépréciés. Aujourd'hui, les noyaux Linux 2.6 utilisent le système de communication inter-processus nommé D-Bus (http://www.freedesktop.org/wiki/Software/dbus), qui est actif avec KDE4, GNOME, Enlightenment, Maemo ou encore Androïd. Son principal développeur est Red Hat, dans le cadre du projet freedesktop.org. Il en existe un portage sous Windows.

Pour accéder aux périphériques, il est nécessaire de faire des opérations de bas niveau et de connaître l'API du projet (http://dbus.freedesktop.org/doc/api/html/index.html). Les principaux objets sont **`DBusConnection`** et **`DBusMessage`**.

Une documentation générale est également disponible sur le site de freedesktop.org (http://dbus.freedesktop.org/doc/dbus-python/doc/tutorial.html).

Le langage de programmation Python dispose d'un module qui permet d'utiliser D-Bus à bas niveau. Le développeur qui souhaite le faire devra donc bien connaître D-Bus ou avoir une documentation sous la main.

Dans cette partie, nous allons voir quelques cas concrets d'utilisation qui seront présentés comme des recettes à appliquer et à améliorer selon nos besoins, mais nous ne détaillerons pas le fonctionnement du module, qui relève plus de la connaissance de D-Bus que de Python.

8.2 Voir les partitions

Le langage Python permet d'utiliser D-Bus à bas niveau pour accéder à des informations système. Cette première recette va montrer à quel point l'important est de connaître l'API du projet D-Bus pour s'en sortir, puisque la partie Python reste relativement accessible.

Voici donc une recette qui donne des informations sur les partitions présentes sur la machine hôte du système :

```
import dbus

bus = dbus.SystemBus()
proxy = bus.get_object(
    "org.freedesktop.Udisks",
    "/org/freedesktop/Udisks"
)
manager = dbus.Interface(proxy, 'org.freedesktop.Udisks')

for dev in manager.EnumerateDevices():
    device_obj = bus.get_object("org.freedesktop.UDisks", dev)
    device_props = dbus.Interface(
        device_obj,
        'org.freedesktop.Dbus.Properties'
    )
    print(device_props.Get(
        'org.freedesktop.Udisks.Device',
        "DriveVendor"
    ))
```

```
    print(device_props.Get(
        'org.freedesktop.Udisks.Device',
        "DeviceMountPaths"
    ))
    print(device_props.Get(
        'org.freedesktop.Udisks.Device',
        "DriveSerial"
    ))
    print(device_props.Get(
        'org.freedesktop.Udisks.Device',
        "PartitionSize"
    ))
```

Voici un exemple de résultat :

```
ATA
dbus.Array([dbus.String('/media/backup')], signature=dbus.Signa-
ture('s'), variant_level=1)
9VPF0P2W
100002921984
```

8.3 Voir les clés USB

Avec cette même méthode, nous pouvons aussi retrouver les disques USB présents sur le système hôte :

```
import dbus

bus = dbus.SystemBus()
proxy = bus.get_object(
    "org.freedesktop.Udisks",
    "/org/freedesktop/Udisks"
)
manager = dbus.Interface(proxy, 'org.freedesktop.Udisks')

for dev in manager.EnumerateDevices():
    device_obj = bus.get_object("org.freedesktop.UDisks", dev)
    device = dbus.Interface(
        device_obj,
        'org.freedesktop.Dbus.Properties'
    )
    if str(device.Get(
        'org.freedesktop.Dbus.Properties',
        "DriveConnectionInterface"
    )) == 'usb':
        name = str(device.Get('org.freedesktop.DBus.Properties',
'DeviceFile'))
        print('USB device: %s' % name)
```

© Editions ENI - All rights reserved

8.4 Liste des informations disponibles

Nous avons choisi une courte sélection d'informations pour les deux exemples précédents, mais il y en a beaucoup plus. Il est important d'avoir accès à la documentation pour faire le bon choix.

Voici comment toutes les obtenir :

```
[...]
manager = dbus.Interface(proxy, 'org.freedesktop.UDisks')

for dev in manager.EnumerateDevices():
    device_obj = bus.get_object("org.freedesktop.UDisks", dev)
    device = dbus.Interface(
        device_obj,
        'org.freedesktop.DBus.Properties'
    )
    for k, v in device.GetAll('').items():
        print('%s : %s' % (k, v))
    print('\n================')
```

8.5 Détecter le branchement d'une clé USB

Pour faire cette opération, il faut procéder en plusieurs étapes. La première est l'initialisation du gestionnaire :

```
from dbus.mainloop.glib import DbusGMainLoop
DbusGMainLoop(set_as_default=True)

import dbus

bus = dbus.SystemBus()
proxy = bus.get_object(
    "org.freedesktop.Udisks",
    "/org/freedesktop/Udisks"
)
manager = dbus.Interface(proxy, "org.freedesktop.UDisks")
```

La seconde est la création des méthodes et connexions des signaux :

```
def on_device_add(dev_path):
    print('Added %s' % dev_path)
manager.connect_to_signal('DeviceAdded', on_device_add)

def on_device_changed(dev_path):
    print('Changed %s' % dev_path)
manager.connect_to_signal('DeviceChanged', on_device_changed)

def on_device_removed(dev_path):
    print('Removed %s' % dev_path)
manager.connect_to_signal('DeviceRemoved', on_device_removed)
```

Puis le lancement de la boucle de traitement :

```
from gi.repository import Gobject
loop = Gobject.MainLoop()
loop.run()
```

On utilise ici la programmation événementielle pour déclencher des actions lorsque certains signaux se produisent, lesquels sont gérés par les spécifications `Udisks`.
Voici ce qu'il se passe lorsque l'on connecte une clé USB :

```
Added /org/freedesktop/Udisks/devices/sdc
Added /org/freedesktop/Udisks/devices/sdc1
Changed /org/freedesktop/Udisks/devices/sdc1
```

Et lorsqu'on la déconnecte :

```
Changed /org/freedesktop/Udisks/devices/sdc1
Changed /org/freedesktop/Udisks/devices/sdc
Removed /org/freedesktop/Udisks/devices/sdc1
Changed /org/freedesktop/Udisks/devices/sdc
Removed /org/freedesktop/Udisks/devices/sdc
Removed /org/freedesktop/UDisks/devices/sdc
```

Le résultat obtenu met en évidence les appels des fonctions de rappels et vous permet de savoir laquelle utiliser et à quel moment.

8.6 Communiquer via un port série

Python a très tôt offert des outils pour réaliser des opérations de très bas niveau consistant à communiquer via un port série. La fonctionnalité principale consiste à ouvrir un port de la machine, et de lire le flux de données qui arrive ou d'en écrire un.
Aujourd'hui, avec la montée en puissance de la robotique et en particulier d'Arduino, ce module visant les spécialistes de l'automatisme, de la domotique ou de la robotique trouve un second souffle et un nouvel intérêt.
Les fonctionnalités ne sont cependant utilisables que si l'on a du matériel pour les tester, il faut donc investir un minimum (cinquante euros maxi pour un bon début) pour pouvoir tester efficacement ce module.
Ce module est une bibliothèque externe disponible pour Python 3 :

```
$ sudo aptitude install python3-serial
```

Ou à l'aide de pip :

```
$ pip install pyserial
```

La première étape est d'importer le module puis de créer une connexion :

```
>>> import serial
>>> s = serial.Serial('COM4')          # Pour Windows
>>> s = serial.Serial('/dev/ttyS1')    # Pour Linux
```

Une fois la connexion ouverte, il est possible de lire des données :

```
>>> s.read()
>>> s.readline()
```

© Editions ENI - All rights reserved

Ou d'en écrire :

```
>>> s.write(b'datas')
```

Et une fois que la communication est terminée, on peut fermer le port série ainsi :

```
>>> s.close()
```

Pour des raisons de sécurité applicative, il est nécessaire de réaliser ces opérations dans une boucle d'exception :

```
import serial
def action():
    with serial.Serial('/dev/ttyS1') as s:
        [... read, write, ...]
```

Pour aller plus loin, il faut connaître les spécifications de ce avec quoi on communique et posséder le manuel adéquat. Pour le reste, le dialogue est beaucoup plus aisé en Python qu'en C, mais il demande néanmoins une certaine maîtrise (utilisation de **setDTR**, de **setDTS**...).
Il existe une documentation assez fournie (https://github.com/pyserial/pyserial) et également un autre module **pyUSB** qu'il est intéressant de regarder pour traiter ces problématiques (http://pyusb.sourceforge.net/docs/1.0/tutorial.html).

8.7 Communiquer via un port USB

Le processus de communication USB est plus complexe. Pour rester simple, ces périphériques sont tous sur un même nœud et celui qui veut communiquer avec un périphérique lui envoie un jeton avec son numéro. Lorsque le périphérique se reconnaît, il envoie une réponse.
Ensuite, il faut ouvrir un canal de communication et envoyer les données. Il existe quatre types de transferts :

- le transfert de commande :
  - données de taille restreinte, garantie de livraison ;
  - utilisé pour configurer le périphérique ;
- le transfert d'interruption :
  - données de taille restreinte, latence faible ;
  - utilisé par claviers et souris ;
- le transfert isochrone :
  - garantie sur la bande passante mais pas la livraison ;
  - utilisé pour flux audio et vidéo ;
- le transfert en masse :
  - garantie sur la livraison, mais pas la bande passante ;
  - utilisé pour le stockage.

Pour commencer, on peut chercher à obtenir la liste des périphériques USB disponibles :

```
>>> devs = usb.core.find(find_all=True)
>>> printers = usb.core.find(find_all=True, bDeviceClass=7)# imprimantes
>>> hubs = usb.core.find(find_all=True, bDeviceClass=9)     # hubs
>>> wless = usb.core.find(find_all=True, bDeviceClass=224) # sans fil
```

Les périphériques USB sont identifiables, entre autres, par deux éléments que sont `idVendor` et `idProduct`. Il faut donc connaître la valeur attendue de ces éléments pour les identifier et pouvoir y accéder :

```
>>> dev = usb.core.find(idVendor=0x1d6b, idProduct=0x0002)
```

Ces deux informations, comme d'autres, peuvent être retrouvées :

```
>>> dev.idVendor
7531
>>> hex(dev.idVendor)
'0x1d6b'
>>> dev.iProduct
2
>>> dev.iManufacturer
3
>>> dev.iSerialNumber
1
```

Il s'agit des propriétés du périphérique USB. Une fois que l'on a identifié et trouvé le périphérique, il faut le configurer :

```
dev.set_configuration()
```

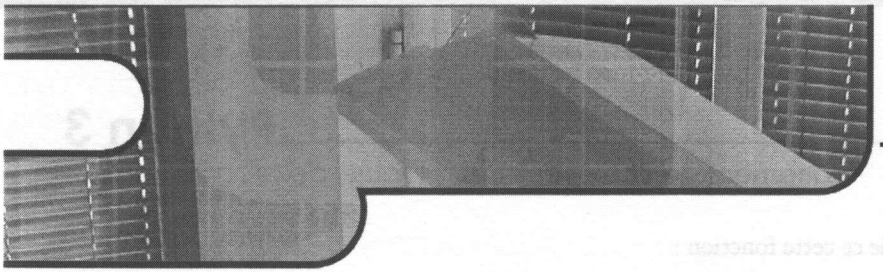
Puis, il faut ouvrir un canal de communication (nommé **endpoint**) :

```
cfg = dev.get_active_configuration()
interface_number = cfg[(0,0)].bInterfaceNumber
alternate_settting = usb.control.get_interface(interface_number)
intf = usb.util.find_descriptor(
    cfg,
    bInterfaceNumber = interface_number,
    bAlternateSetting = alternate_setting
)
def my_custom_match(e):
    endpoint_direction = usb.util.endpoint_direction(e.bEndpointAddress)
    return endpoint_direction == usb.util.ENDPOINT_OUT
ep = usb.util.find_descriptor(
    intf,
    custom_match = my_custom_match,
)
```

On peut maintenant envoyer des données :

```
ep.write('datas')
```

© Editions ENI - All rights reserved



Chapitre 2.2
Programmation réseau

1. Écrire un serveur et un client

1.1 Utilisation d'une socket TCP

TCP est l'acronyme de *Transmission Control Protocol*, soit protocole de contrôle de transmission et a été développé en 1973 et documenté au sein de la RFC 793. Il est situé dans le modèle OSI au sein de la couche transport et est répandu de par sa fiabilité.

Il se caractérise par la manière de mettre en place la synchronisation entre client et serveur et par le découpage en segments des octets à transmettre, chaque segment étant parfaitement identifiable et disposant d'un système de contrôle d'intégrité qui fait que celui qui reçoit un paquet peut savoir que le paquet est corrompu et le redemander.

Le flux TCP utilise les sockets et c'est le module éponyme de Python qui va nous permettre de travailler avec TCP.

L'idée ici est de réaliser un mini serveur de données clé-valeur très basique et le plus simple possible de manière à voir comment créer un serveur puis un client, mais aussi comment manipuler les données qui sont échangées de l'un à l'autre.

Ce qu'il faut absolument comprendre est que les données qui sont transmises d'un serveur à un client ou d'un client à un serveur sont des octets, et rien d'autre. En Python 2, cela pouvait prêter à confusion, puisque le type **`str`** de Python 2 était assimilable à des octets. Le type Python 3 **`str`** représente une chaîne de caractères en unicode et c'est le type **`bytes`** qui permet de gérer des octets.

Il faut donc particulièrement être attentif à ce point, puisque la plupart des exemples présents sur le net, par ailleurs d'excellentes qualités, sont écrits pour Python 2 et génèrent une confusion sur le type de données réellement envoyées.

On va créer un serveur qui s'attend à recevoir un nombre et qui renvoie un code en fonction du fait que ce nombre soit premier ou non.

Voici le code ce cette fonction :

```
def isprime(n):
    '''check if integer n is a prime'''

    # negative numbers, 0 and 1 are not primes
    if n < 2:
        return False

    # 2 is the only even prime number
    if n == 2:
        return True

    # all other even numbers are not primes
    if not n & 1:
        return False

    # range starts with 3 and only needs to go up the squareroot
    # of n for all odd numbers
    for x in range(3, int(n**0.5)+1, 2):
        if n % x == 0:
            return False
    return True
```

Voici donc le code du serveur (disponible en téléchargement) :

```
#!/usr/bin/python3

import socket

params = ('127.0.0.1', 8809)
BUFFER_SIZE = 1024 # default

donnees = {}

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)#Internet, TCP
s.bind(params)
s.listen(1)
```

Le serveur est lancé et il écoute sur le port 8809. la méthode accept attend qu'une connexion se présente et dès qu'elle arrive, elle renvoie les éléments nécessaires pour que la connexion puisse être traitée. Ce qui est réalisé dans une boucle sans fin. Le principe est que tant que l'échange client serveur dure, il est traité.

```
conn, addr = s.accept()
print('Connexion acceptée: %s' % str(addr))

while True:
        data = conn.recv(BUFFER_SIZE)
        if not data:
            break
```

© Editions ENI - All rights reserved

```
        print('Donnée reçue: %s' % data)
        try : number = int(data)
        except:
            response = b'E'
            phrase = "'%s' n'est pas un entier" % data
        else:
            if isprime(number):
                phrase = "'%s' est un nombre premier" % number
                response = b'T'
            else:
                phrase = "'%s' n'est pas est un nombre premier" % number
                response = b'F'
        finally:
            print(response)
            conn.send(response)
conn.close()
```

La partie complexe n'est pas tant la gestion de la connexion et des transferts réseau que le traitement des données. Ce qui compte est de savoir quoi faire des données et comment mettre en place un dialogue entre client et serveur, sachant que les seuls éléments échangeables sont des octets.

Dans cet exemple, on a choisi la convention suivante : si on ne comprend pas la donnée transmise par le client, on renvoie un code E (pour erreur). Si le nombre est premier, on renvoie un code T (pour True), sinon on renvoie un code F (pour False).

Les choix effectués dans cet exemple sont assez basiques et limités : on cherche à transmettre le moins de données possible, mais serveur et client doivent s'entendre pour se comprendre entre eux : le client devra connaître l'ensemble des codes susceptibles d'être envoyés par le serveur et les gérer.

Cela montre les problématiques typiques qui se posent lorsque l'on cherche à manipuler des données en utilisant des couches de bas niveau.

Voici le code du client, qu'il faut lire en faisant le lien avec celui du serveur :

```
#!/usr/bin/python3

import socket

params = ('127.0.0.1', 8809)
BUFFER_SIZE = 1024 # default

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(params)
```

Le parti a été pris d'envoyer une succession de messages de manière à couvrir tous les cas possibles de traitement côté serveur.

```
chiffres = [4j, 4, 5, -5, 17, 29, 2**50, 2**50-1]
```

En face de chaque message, qui est en réalité une sorte d'instruction donnée au serveur, est mis en commentaire le comportement attendu.
Voici le code qui effectue les requêtes vers le serveur et qui traite les réponses :

```
for chiffre in chiffres:
    s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(params)
    print("Envoi d'un message %s" % chiffre)
    s.send(('%s\n' % chiffre).encode('utf8'))
    data = s.recv(BUFFER_SIZE)
    if len(data) == 0:
        print("\tPas de réponses")
    elif data == b'E\n':
        print("\tUne erreur est détectée par le serveur")
    elif data == b'T\n':
        print("\tLe nombre est bien un nombre premier")
    elif data == b'F\n':
        print("\tLe nombre n'est pas un nombre premier")
    else:
        print("\tLa donnée reçue n'est pas comprise : %s" % data)
    s.close()
```

TODO : tant que l'on est dans la boucle, on communique avec le serveur et donc le serveur reste dans la boucle infinie de communication. Le fait de fermer la connexion côté client envoie une trame vide qui va permettre de passer dans le break côté serveur (**`if not data: break`**) et ainsi libérer le serveur.
S'il y avait une boucle autour de la méthode **`accept`**, le serveur continuerait à tourner en attendant de nouvelles données qu'une nouvelle connexion lui parvienne et tournerait ainsi à l'infini, mais en l'occurrence, il n'y a pas cette boucle et il se stoppe donc.
Cet exemple ne permet donc que de parler à un seul client, puis d'arrêter le serveur. Il existe, dans la pratique, des cas d'utilisation tels que celui-ci.
Il existe, par exemple, un excellent outil nommé woof qui s'installe via le gestionnaire de paquet et se lance en ligne de commande :

```
$ sudo aptitude install woof
```

```
$ python path/to/woof.py path/to/filename
```

Un serveur est lancé et propose une URL qu'il suffit de donner à quelqu'un par IRC ou un autre moyen pour que cette personne, en cliquant dessus, télécharge le fichier. À la fin du téléchargement, le serveur se ferme. On peut donc transmettre un document par le réseau rapidement.

© Editions ENI - All rights reserved

1.2 Utilisation d'une socket UDP

Comme TCP, UDP appartient à la couche transport du modèle OSI, mais contrairement à lui, il ne réalise pas de connexion préliminaire, de synchronisation et ne garantit pas la bonne réception des données. Par contre, il intègre un système qui assure l'intégrité de la donnée transmise, comme TCP.

UDP est moins fiable que TCP, car il peut perdre des paquets, mais il est beaucoup plus rapide car il nécessite beaucoup moins d'allers-retours. Il est donc préférable pour des technologies pour lesquelles la fiabilité n'est pas le principal besoin par rapport à la rapidité, comme la voix sur IP, le streaming ou les jeux en réseau.

Ceci est vrai dans la mesure où les applications concernées sont capables soit de se passer d'une petite partie des données qui auraient été perdues, soit de les reconstruire (à partir du moment où cela est plus rapide que les redemander), soit de les substituer.

D'autre part, TCP est utilisé pour établir un dialogue entre un client et un serveur alors qu'UDP est utilisé pour envoyer des données du client vers le serveur sans que le client attende un retour.

Là encore, on va utiliser les sockets, mais comme les protocoles TCP et UDP différent dans leur fonctionnement, cette différence est visible dans la manière d'écrire un serveur et un client.

Ainsi, le serveur n'attend pas de synchroniser une connexion avec un client et le client ne se connecte pas au serveur avant de lui envoyer des données.

En dehors de cela, la manière de traiter les données est identique à ce que l'on a vu pour TCP. Des exemples plus simples que ceux montrés dans cet ouvrage permettent de mettre en évidence les différences entre les protocoles TCP (http://wiki.python.org/moin/TcpCommunication) et UDP (http://wiki.python.org/moin/UdpCommunication). Dans les deux cas, les serveurs affichent la donnée reçue et pour TCP, la donnée est également renvoyée au client.

L'exemple suivant est dans la lignée de celui pour TCP, le client donnant des informations au serveur. Les lignes inutiles sont laissées en commentaire afin de mettre en évidence les différences avec TCP.

```
#!/usr/bin/python3

import socket

params = ('127.0.0.1', 8808)
BUFFER_SIZE = 1024 # default

donnees = {}

s=socket.socket(socket.AF_INET, socket.SOCK_DGRAM) #SOCK_STREAM
s.bind(params)
# Les lignes suivantes n'ont pas de sens en UDP:
#s.listen(1)
#conn, addr = s.accept()
#print('Connexion acceptée: %s' % str(addr))
```

On voit clairement dans la première partie qu'il n'est pas nécessaire de se mettre à l'écoute et d'initier une connexion.
Dans la partie qui suit, la différence de sémantique entre **recv** et **recvfrom** est révélatrice du fait que la méthode **recvfrom** reçoit à la fois la donnée et l'adresse du client.

```
while True:
        #data = conn.recv(BUFFER_SIZE)
        data, addr = s.recvfrom(BUFFER_SIZE)
        print('Connexion reçue: %s' % str(addr))
        print('Donnée reçue: %s' % data)
        if not data:
            break
        print('Donnée reçue: %s' % data)
        try:
            number = int(data)
        except:
            response = b'E'
            phrase = "'%s' n'est pas un entier" % data
        else:
            if isprime(number):
                phrase = "'%s' est un nombre premier" % number
                response = b'T'
            else:
                phrase = "'%s' n'est pas est un nombre premier" % number
                response = b'F'
        finally:
            print(response)
#conn.close()
```

À noter que si pour TCP il faut une boucle pour gérer chaque connexion et une autre pour gérer les échanges au sein de la connexion, en UDP une seule boucle suffit. Pour que l'exemple puisse se terminer proprement, il a été rajouté un cas d'utilisation particulier permettant de sortir de la boucle.
Voici la partie cliente correspondant :

```
#!/usr/bin/python3

import socket
params = ('127.0.0.1', 8808)
BUFFER_SIZE = 1024 # default

s=socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
#s.connect(params)
```

Là encore, pas besoin de connexion. C'est au moment où l'on envoie les données que l'on donne les informations sur le serveur à atteindre.

```
chiffres = [4j, 4, 5, -5, 17, 29, 2**50, 2**50-1]
```

© Editions ENI - All rights reserved

Les messages envoyés sont de nature différente, et le dernier message permet d'interrompre le serveur proprement. Voici la boucle et là encore, il y a une différence de sémantique entre **send** et **sendto** qui est là pour mettre en évidence la différence entre les deux protocoles et par conséquent des paramètres attendus.

```
for chiffre in chiffres:
    print("Envoi d'un message %s" % chiffre)
    #s.send(('%s\n' % chiffre).encode('utf8'))
    s.sendto(('%s\n' % chiffre).encode('utf8'), params)
    data = s.recv(BUFFER_SIZE)
    if len(data) == 0:
        print("\tPas de réponses")
    elif data == b'E\n':
        print("\tUne erreur est détectée par le serveur")
    elif data == b'T\n':
        print("\tLe nombre est bien un nombre premier")
    elif data == b'F\n':
        print("\tLe nombre n'est pas un nombre premier")
    else:
        print("\tLa donnée reçue n'est pas comprise : %s" % data)
#s.close()
```

Il faut alors lancer le serveur, puis le client, dans deux consoles différentes pour voir le résultat obtenu.

Comme on peut le voir, à peu de choses près, lesquelles s'expliquent par la différence entre les deux protocoles, une bonne partie du code est identique entre TCP et UDP. Pour être plus précis, le code métier est identique.

Pour des besoins standards, il existe une couche légèrement moins bas niveau, présentée dans les deux sections suivantes, laquelle permet d'aller encore plus loin dans l'uniformisation de l'utilisation des protocoles réseau.

1.3 Création d'un serveur TCP

Comme nous l'avons vu précédemment, la gestion d'un serveur TCP est une succession d'étapes classiques et reproductibles quel que soit le type de serveur que l'on veut monter. La partie la plus complexe réside en la détermination de la manière de traiter les données que l'on reçoit afin de réaliser d'éventuelles actions et de renvoyer une réponse adaptée.

Ceci a été modélisé dans le module **socketserver** (**SocketServer** pour la branche 2.x de Python) qui offre deux classes assignées à cette problématique : **TCPServer** qui permet de gérer toute la problématique de communication avec le client et **StreamRequestHandler** qui permet de gérer la problématique de gestion des requêtes, c'est-à-dire comment traiter les données qui arrivent.

Ainsi, pour reproduire le même travail que précédemment, le code serveur ressemblerait à celui-ci :

```
#!/usr/bin/python3

import socketserver
```

```
params = ('127.0.0.1', 8809)

donnees = {}

class PrimeTCPHandler(socketserver.StreamRequestHandler):
    def handle(self):
        data = self.rfile.readline().strip()
        try:
            number = int(data)
        except:
            response = b'E'
            phrase = "'%s' n'est pas un entier" % data
        else:
            if isprime(number):
                phrase = "'%s' est un nombre premier" % number
                response = b'T'
            else:
                phrase = "'%s' n'est pas est un nombre premier" % number
                response = b'F'
        finally:
            print(response)
            with open("prime_server.log", "a") as f:
                f.write(phrase +'\n')
            return self.wfile.write(response + b'\n')

if __name__ == '__main__':
    server = socketserver.TCPServer(params, PrimeTCPHandler)
    server.serve_forever()
```

On met donc clairement en évidence la classe permettant de traiter la requête qui est bien séparée des deux simples lignes qui gèrent le serveur TCP.
Cette mise en évidence facilite l'accessibilité du code, sa lisibilité mais laisse également des portes ouvertes pour les concepteurs en leur permettant de réaliser simplement une petite architecture pour gérer correctement leurs handlers, lesquels contiennent le code métier.
Il est à noter l'utilisation de l'attribut rfile pour lire les données qui arrivent et wfile pour écrire celles qui sont renvoyées. Il est à noter que l'ensemble de l'échange avec le client est traité dans un handler, ce qui comprend la mise en place de la connexion, l'échange des données, de la réponse et la fin de la connexion.
Ceci signifie donc que le client DOIT prendre ces changements en compte et recommencer une nouvelle connexion à chaque nouvelle requête envoyée :

```
!/usr/bin/python3

import socket

params = ('127.0.0.1', 8809)
BUFFER_SIZE = 1024 # default

chiffres = [4j, 4, 5, 17, 29, 2**50, 2**50-1]
```

© Editions ENI - All rights reserved

```
for chiffre in chiffres:
    t0 = time()
    s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(params)
    print("Envoi d'un message %s" % chiffre)
    s.send(('%s\n' % chiffre).encode('utf8'))
    data = s.recv(BUFFER_SIZE)
    if len(data) == 0:
        print("\tPas de réponses")
    elif data == b'E\n':
        print("\tUne erreur est détectée par le serveur")
    elif data == b'T\n':
        print("\tLe nombre est bien un nombre premier")
    elif data == b'F\n':
        print("\tLe nombre n'est pas un nombre premier")
    else:
        print("\tLa donnée reçue n'est pas comprise : %s" % data)
    s.close()
    print("\tTemps passé : %s" % (time()-t0))
```

En effet, le socket est recréé et la connexion réinitialisée à chaque fois qu'il y a besoin d'envoyer un nouveau message, ce qui peut paraître représenter du travail supplémentaire.
Il existe également des classes plus basiques qui sont plus proches du bas niveau et qui établissent un pont entre ce que l'on a vu précédemment et ces deux dernières classes (http://docs.python.org/py3k/library/socketserver.html).

1.4 Création d'un serveur UDP

De la même manière qu'il existe deux classes **`TCPServer`** et **`StreamRequestHandler`** pour TCP, il existe deux classes **`UDPServer`** et **`DatagramRequestHandler`** pour UDP qui ont les mêmes objectifs.
Et c'est là que l'avantage d'utiliser ces handlers devient évident : ils masquent complètement la complexité de ce qui se passe réellement au niveau du réseau pour permettre une manière de coder similaire.
Ainsi, si l'on prend le code précédent, que l'on remplace les lettres TCP par UDP et l'héritage vers StreamRequestHandler vers DatagramRequestHandler, le serveur UDP obtenu offre exactement le même service que celui écrit pour TCP.
Les codes sont similaires et pourtant les processus mis en œuvre sont très différents.
Pour mettre ceci en évidence, il convient de réutiliser le client UDP réalisé précédemment.
On peut ainsi transmettre des données de la même manière en TCP et en UDP bien que les deux protocoles aient des buts différents et des modes de fonctionnement réellement différentes.
Là encore, on reste bas niveau, mais on a déjà une bonne abstraction des couches réseau et l'on peut répondre à beaucoup de problématiques classiques. Pour utiliser des niveaux d'abstraction encore plus hauts, il faut utiliser les frameworks.

1.5 Un peu plus loin sur le sujet

Lorsque le serveur et le client sont sur la même machine, il est dommage de perdre du temps dans la couche réseau à négocier une connexion alors que le dialogue peut s'instaurer beaucoup plus rapidement.
C'est le rôle des sockets IPC ou sockets du domaine UNIX. La communication a ainsi lieu directement au cœur du noyau et les processus peuvent s'échanger directement des données.
Avant d'établir une connexion, il est important de veiller à ce que le fichier utilisé pour référencer le socket n'existe pas :

```
import os, os.path
if os.path.exists("/tmp/mycustomsock"):
    os.remove("/tmp/myustomsock")
```

Une fois ceci effectué, on peut créer le serveur qui va l'utiliser :

```
import socket
server = socket.socket(socket.AF_UNIX, socket.SOCK_DGRAM)
server.bind("/tmp/myustomsock")
```

Et le client :

```
client = socket.socket(socket.AF_UNIX, socket.SOCK_DGRAM)
client.connect("/tmp/myustomsock")
```

Pour le reste, le fonctionnement reste similaire à ce qui a été vu précédemment.
Il est à noter que MySQL et PostgreSQL, entre autres, utilisent ce système et que nombre de serveurs web se privent de cette possibilité qui est relativement utile.
Les possibilités offertes par un serveur dépendent également du système d'exploitation. Pour qu'il écoute autre chose que la boucle locale, il faut utiliser des outils annexes :

```
>>> import os
>>> kernel, hostname, release, version, hardware = os.uname()
>>> print('Hostname: %s' % hostname)
Hostname: srv.example.com
```

À partir de ces données, il est possible d'obtenir un certain nombre de renseignements :

```
>>> import socket
>>> address = socket.gethostbyname(hostname)
>>> print(address)
127.0.1.1
>>> socket.gethostbyaddr(address)
('srv.example.com', [], ['127.0.1.1'])
```

Ou plus directement :

```
>>> socket.gethostbyname_ex(hostname)
('srv.example.com', [], ['127.0.1.1'])
```

Il faut alors utiliser ces caractéristiques lors de la création du serveur TCP ou UDP pour que le serveur écoute plus largement que la simple boucle locale, c'est-à-dire la machine elle-même.

© Editions ENI - All rights reserved

Il est à noter que Python 3.3 a introduit **socket.sendmsg** qui est une méthode de plus haut niveau permettant d'envoyer des données complexes plus simplement (cette méthode a été améliorée par Python 3.5). Python 3.5 a aussi créé **socket.sendfile** qui évite de faire un **file.read** et un **socket.send** et qui permet aussi que l'entière opération soit réalisée au niveau du noyau, donc avec de meilleures performances.

2. Utiliser un protocole standard

2.1 Client HTTP

L'idée ici est de s'adresser à un serveur, via le protocole HTTP et de récupérer des informations (page web ou autre).
Pour cela, il faut connaître le protocole HTTP afin de savoir de quelle manière s'adresser au serveur. Voici un exemple simple. On se connecte au serveur :

```
>>> conn = http.client.HTTPConnection('www.google.fr')
```

On utilise une des méthodes HTTP afin d'aller chercher les données souhaitées :

```
>>> conn.request("GET", "/search?hl=fr&q=python")
```

On récupère la réponse du serveur :

```
>>> response=conn.getresponse()
```

On vérifie que tout s'est bien passé et pour cela, il faut connaître les codes d'erreurs HTTP qui sont tous parfaitement expliqués dans la documentation officielle :
http://docs.python.org/py3k/library/http.client.html

```
>>> print(response.status, response.reason)
200 OK
```

Comme tout s'est bien passé, il suffit de récupérer la page HTML renvoyée par le serveur :

```
>>> html=response.read()
```

Voyons voir à quoi cela peut ressembler :

```
>>> len(html)
85558
>>> html[:80]
b'<!doctype html><head><title>python - Recherche
Google</title><script>window.goog'
```

Il suffit maintenant de traiter le HTML pour en extraire les données que l'on souhaite et cela peut se faire aisément avec un parser tel que BeautifulSoup et un peu de recherche pour comprendre la structure de la page et être capable d'en tirer les bonnes informations.
Évidemment, une telle utilisation est archaïque lorsque l'on dispose de services web, mais il n'est parfois pas possible de faire autrement (site ancien ou payant).
L'écriture de tels scripts nécessite bien sûr une veille attentive pour déceler tout changement dans la structure de la page qui mettrait en péril la bonne lecture des données. Aussi est-il important selon le besoin de garder des sauvegardes des pages entières pour relancer un traitement correctif a posteriori.
La documentation officielle donne un exemple pour simuler le POST d'un formulaire.

Dans la dernière partie, nous créerons une application console qui entre des données dans une application web via l'émission de requêtes POST.

2.2 Serveur HTTP

Un serveur HTTP est un serveur TCP qui utilise le protocole HTTP. De la même manière que ce que l'on a pu voir précédemment, Python fournit des outils spécifiques pour gérer aisément les couches basses du traitement des échanges réseau. Aussi, il fournit deux classes qui sont dans la droite lignée de TCP :

```
>>> import http.server
>>> type.mro(http.server.HTTPServer)
[<class 'http.server.HTTPServer'>, <class 'socketserver.TCPServer'>,
<class 'socketserver.BaseServer'>, <class 'object'>]
>>> type.mro(http.server.BaseHTTPRequestHandler)
[<class 'http.server.BaseHTTPRequestHandler'>,
<class 'socketserver.StreamRequestHandler'>,
<class 'socketserver.BaseRequestHandler'>, <class 'object'>]
```

Ceci permet de créer rapidement un petit serveur pour renvoyer du contenu par HTTP. Pour cela, le concepteur doit décrire le comportement attendu du serveur lorsqu'il reçoit une requête, ce qui se fait en implémentant une sous-classe de **`BaseHTTPRequestHandler`**.

Il existe cependant quelques différences entre TCP et HTTP pour le traitement de la requête, car la classe effectue déjà un premier travail pour récupérer la méthode utilisée par le client et abstrait une partie significative liée au protocole HTTP.

Il faut donc non plus surcharger une méthode **`handle`**, mais surcharger une méthode par méthode HTTP qui suit la règle de nommage suivante : do_METHOD.

```
#!/usr/bin/python3
from http.server import BaseHTTPRequestHandler, HTTPServer

params = '127.0.0.1', 8016

class HelloHandler(BaseHTTPRequestHandler):
    def do_HEAD(self):
        self.send_response(200)
        self.senf_header('Content-type', 'text.html')
        self.end_headers()
    def do_GET(self):
        self.do_HEAD()
        self.wfile.write("""<html><head><title>Hello
World</title></head><body><p>Hello World</p></body></html>""")
   do_POST = do_GET

server = HTTPServer(params, HelloHandler)
server.serve_forever()
```

© Editions ENI - All rights reserved

Une fois le serveur lancé en ligne de commande, il n'y a plus qu'à tester le résultat :

```
>>> import http.client
>>> conn = http.client.HTTPConnection('127.0.0.1', 8016)
>>> conn.request('GET', '/url/de/test?a=42&b=oui')
>>> response = conn.getresponse()
>>> response.status, response.reason
(200, 'OK')
>>> response.read()
b'<html><head><title>Hello World</title></head><body><p>Hello
World</p></body></html>'
```

Avec un POST, on obtiendrait exactement la même chose, puisque les deux méthodes sont identiques. Par contre, avec une DELETE, non déclaré :

```
>>> conn.request('DELETE', '/url/de/test?a=42&b=oui')
>>> response = conn.getresponse()
>>> response.status, response.reason
(501, "Unsupported method ('DELETE')")
```

À noter que le serveur écrit dans la console ce qui se passe lorsqu'il est sollicité :

```
localhost - "GET /url/de/test?a=42&b=oui HTTP/1.1" 200 -
localhost - "POST /url/de/test?a=42&b=oui HTTP/1.1" 200 -
localhost - code 501, message Unsupported method ('DELETE')
localhost - "DELETE /url/de/test?a=42&b=oui HTTP/1.1" 501 -
```

Maintenant que l'on a vu comment gérer côté serveur l'envoi de données, il faut aborder la manière de traiter celles qui nous sont envoyées par le client et celles de l'environnement.
D'une part, certaines informations sont présentes directement dans des attributs de classes, d'autre part, il faut aller s'aider de librairies annexes pour décortiquer certaines informations, telle **`urllib`** :

```
#!/usr/bin/python3

from http.server import BaseHTTPRequestHandler, HTTPServer

from urllib.parse import urlparse

params = '127.0.0.1', 8016

class HelloHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        infos = []
        infos.append('client_adress: %s' % str(self.client_address))
        infos.append('address_string: %s' % self.address_string())
        infos.append('command: %s' % self.command)
        infos.append('unparsed path: %s' % self.path)
        parsed = urlparse(self.path)
        infos.append('parsed path %s' % parsed.path)
        infos.append('query: %s' % parsed.query)
```

```
        infos.append('request_version: %s' % self.request_version)
        infos.append('server_version: %s' % self.server_version)
        infos.append('sys_version: %s' % self.sys_version)
        infos.append('protocol_version: %s' % self.protocol_version)
        for k, v in self.headers.items():
            infos.append('HEADER %s: %s' % (k, v.strip()))
        infos = b'<ul><li>' + b'</li><li>'.join([bytes(i,
'utf-8') for i in infos]) + b'</li></ul>';
        self.send_response(200)
        self.send_header(b'Content-type', b'text.html')
        self.end_headers()
        self.wfile.write(b"""<html><head><title>Hello
World</title></head><body><p>Hello World</p>""" + infos +
b"""</body></html>""")

server = HTTPServer(params, HelloHandler)
server.serve_forever()
```

Il suffit de recommencer à tester en créant un client dans une console Python :

```
>>> conn = http.client.HTTPConnection('127.0.0.1', 8016)
>>> conn.request('GET', '/url/de/test?a=42&b=oui')
>>> response = conn.getresponse()
>>> response.status, response.reason
(200, 'OK')
>>> response.read()
```

Et de mettre en forme le dernier résultat, pour obtenir :

- client_adress: ('127.0.0.1', 34522)
- address_string: localhost
- command: GET
- unparsed path: /url/de/test?a=42&b=oui
- parsed path /url/de/test
- query: a=42&b=oui
- request_version: HTTP/1.1
- server_version: BaseHTTP/0.6
- sys_version: Python/3.2
- protocol_version: HTTP/1.0
- HEADER Host: 127.0.0.1:8016
- HEADER Accept-Encoding: identity

© Editions ENI - All rights reserved

À partir de là, il faut savoir exploiter les URL (http://docs.python.org/py3k/library/urllib.parse.html?highlight=urlparse#urllib.parse.urlparse) et éventuellement les cookies (http://docs.python.org/py3k/library/http.cookies.html). Il est à noter que deux implémentations particulières existent, permettant toutes deux de servir l'arborescence de fichiers disponibles à partir de la racine de l'application. La première travaille directement, l'autre par l'utilisation de CGI :

```
>>> type.mro(http.server.SimpleHTTPRequestHandler)
[<class 'http.server.SimpleHTTPRequestHandler'>,
<class 'http.server.BaseHTTPRequestHandler'>,
<class 'socketserver.StreamRequestHandler'>,
<class 'socketserver.BaseRequestHandler'>, <class 'object'>]
>>> type.mro(http.server.CGIHTTPRequestHandler)
[<class 'http.server.CGIHTTPRequestHandler'>,
<class 'http.server.SimpleHTTPRequestHandler'>,
<class 'http.server.BaseHTTPRequestHandler'>,
<class 'socketserver.StreamRequestHandler'>,
<class 'socketserver.BaseRequestHandler'>, <class 'object'>]
```

À noter qu'une telle mise à disposition d'une arborescence de fichier peut s'effectuer simplement en allant dans le répertoire voulu en console et en tapant :

```
python -m http.server 8000
```

2.3 Proxy

Un besoin courant est de pouvoir accéder à une ressource qui se trouve derrière un proxy. Les systèmes d'exploitation modernes permettent de configurer ce proxy directement au niveau du système d'exploitation et les applications doivent savoir aller chercher l'information et se configurer seules. Python sait faire cela.
Par contre, lorsque l'on n'a pas configuré le système, que l'on ne le peut pas ou que l'on souhaite utiliser un proxy alternatif pour un script, il faut alors configurer son application d'une manière particulière. Et Python ne déroge pas à la règle.
Voici la procédure dans le cas où l'on n'utilise pas d'authentification :

```
>>> import urllib2
>>> proxy_info = {'host' : 'proxy.mondomaine.org', 'port' : 8080}
```

À partir de ces informations sur le proxy, on construit l'URL d'accès :

```
>>> proxies = {"http": "http://%(host)s:%(port)d" % proxy_info}
```

En réalité, on a une URL par protocole, ce qui signifie qu'il faut un dictionnaire avec une clé par protocole qui nécessite le proxy.

```
>>> proxy_support = urllib2.ProxyHandler(proxies)
>>> opener = urllib2.build_opener(proxy_support)
>>> urllib2.install_opener(opener)
```

À partir de cet instant, le proxy peut être traversé :

```
html = urllib2.urlopen("http://python.org/").read()
```

Si l'on veut utiliser une authentification, il faut effectuer de petites modifications :

```
>>> proxy_info = {
    'host' : 'proxy.mondomaine.org', 'port' : 8080,
    'user' : 'sch', 'pass' : 'secret',}
>>> proxies = {
    "http":
        "http://%(user)s:%(pass)s@%(host)s:%(port)d" % proxy_info}
```

2.4 Cookies

Une problématique bien connue du web consiste en l'utilisation des cookies. L'utilisateur s'identifie et doit recevoir un cookie :

```
import cookielib, urllib, urllib2
login, password = 'sch@mondomaine.org', 'secret'
cookiejar = cookielib.CookieJar()
urlOpener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cookiejar))
data = urllib.urlencode({'login':login, 'password':password })
request = urllib2.Request("http://mondomaine.com/login", data)
url = urlOpener.open(request)
```

Le cookie est mis à jour avec les informations de connexion.
Il faut vérifier la validité du cookie (qui sera refourni à chaque **`urlOpener.open`**) :

```
if not 'id' in [cookie.name for cookie in cookiejar]:
    raise ValueError, "Echec de connexion"
```

2.5 FTP et SFTP

FTP est l'acronyme de *File Transfer Protocol* et comme son nom l'indique il s'agit d'un protocole de communication utilisé pour transférer des fichiers d'un serveur vers plusieurs clients ou d'un client vers le serveur.
Il est décrit par plusieurs RFC et la communication entre client et serveur est régie par des règles précises qu'il faut connaître pour en tirer parti.
Avec un peu de connaissances, la console Python est un très bon client FTP. Pour commencer, il faut connaître un FTP libre d'accès pour tester les fonctionnalités, ou en installer un en local pour tester les subtilités, l'authentification et l'upload.
Nous prendrons comme exemple ce site FTP :

```
>>> url = 'ftp.debian.org'
```

À partir de là, créer une connexion est un jeu d'enfant, avec le mot-clé **`with`** :

```
>>> from ftplib import FTP
>>> with FTP(url) as conn:
```

Il faut maintenant s'authentifier en donnant, en clair, son login et mot de passe ou rien pour une connexion anonyme (la réponse qui correspond à la ligne et qui est obtenue à la fin de la boucle **`with`** est affichée ici pour la compréhension) :

```
...     conn.login()
'230 Login successful.'
```

© Editions ENI - All rights reserved

Il est possible de se déplacer dans l'arborescence :

```
...     conn.cwd('/debian')
'250 Directory successfully changed.'
```

De lister les fichiers :

```
...     conn.retrlines('LIST')
[...]
-rw-r--r--  1 1176  1176  78596 Oct 01 13:52 README.mirrors.txt
[...]
'226 Directory send OK.'
```

Et de télécharger un fichier :

```
...     with open('test.txt', 'wb') as f:
...         conn.retrbinary('RETR README.mirrors.txt', f.write)
...
'226 Transfer complete.'
```

On a utilisé respectivement les commandes FTP **`LIST`** et **`RETR`** avec un callback.
Ne pas oublier le caractère **`b`** lorsque l'on ouvre le fichier, puisque l'on récupère des bytes et non pas une chaîne de caractères unicode (**`str`**).
Voici le résultat :

```
$ head test.txt
                              Debian worldwide mirror sites[...]
```

Enfin, la déconnexion est réalisée par la fin de la boucle with (sinon par **`quit`**) :

```
'221 Goodbye.'
```

Parfois, il arrive d'obtenir au cours de la manipulation un : **`421 Timeout`**.
Quelques autres cas d'utilisation courants, pour une URL qui n'existe pas ou ne dispose pas de service FTP :

```
socket.error: [Errno 101] Network is unreachable
```

Lorsque l'authentification est obligatoire, qu'il n'existe pas de compte anonyme :

```
ftplib.error_perm: 530 Please login with USER and PASS.
```

Il est à noter qu'il existe d'autres méthodes intéressantes à explorer. Ainsi, il est possible d'obtenir la liste des fichiers d'un répertoire plus facilement :

```
...     conn.nlst()
['README', 'README.CD-manufacture', 'README.html', 'README.mirrors.html',
'README.mirrors.txt', 'dists', 'doc', 'indices', 'ls-lR.gz',
'ls-lR.patch.gz', 'pool', 'project', 'tools']
```

De savoir dans quel répertoire on se trouve actuellement :

```
...     conn.pwd()
'/debian'
```

Il est aussi possible de récupérer le message de présentation :

```
...     conn.getwelcome()
'220 ftp.debian.org FTP server'
```

Une fois que l'on connaît tout ceci et que l'on est à l'aise avec ces fonctionnalités, tout le reste n'est que connaissance du protocole FTP.
Ainsi, si l'on sait télécharger un fichier, on sait aussi en déposer sur un serveur, car il s'agit du même procédé, mais pas la même méthode. Au lieu d'utiliser la commande **RETR** (pour retrieve), on utilise la commande **STOR** pour (store).
Bien entendu, il faut adapter le callback pour pointer vers une méthode de lecture :

```
...     with open('test.txt', 'rb') as f:
...         conn.storbinary('STOR test_on_ftp.txt', f.read)
...
'226 Transfer complete.'
```

FTP n'est pas un protocole sécurisé, ce qui peut réellement poser des problèmes.
Aussi est-il important d'utiliser un protocole plus sécurisé sans que cela remette tout en cause (utilisable de la même manière).
Pour cela, le plus simple est de s'installer en local un serveur FTP sécurisé :

```
>>> from ftplib import FTP_TLS
>>> with FTP_TLS(url) as conn:
...     conn.login()
...     conn.prot_p()
[...]
```

À part quatre nouvelles méthodes, tout le reste fonctionne de la même manière :

```
>>> type.mro(FTP_TLS)
[<class 'ftplib.FTP_TLS'>, <class 'ftplib.FTP'>, <class 'object'>]
```

À noter que l'utilisation du mot-clé **with** est récente (Python 3.2).

2.6 SSH

SSH est un protocole de communication sécurisé par un échange de clés de chiffrement en début de communications. Il s'agit des clés publiques de chaque entité communicante.
Ces clés n'étant pas symétriques, seul celui qui a la clé de déchiffrement peut décrypter un message crypté avec la clé de chiffrement liée. Toute la sécurité du protocole réside dans la génération de ces paires de clés et à leur unicité ainsi qu'au fait que la possession de la clé publique ne permet pas de retrouver la clé privée.
SSH est aujourd'hui un protocole de référence et il est lié à beaucoup d'autres technologies, car il est utilisé au-dessus d'autres protocoles, en tant que couche de chiffrement pour apporter une sécurisation (TLS est également un protocole utilisé à cet effet), par exemple pour HTTPS ou SFTP.
La différence essentielle avec TLS est que ce dernier repose sur des certificats émis ou assumés par une autorité de certification.
SSH est également un logiciel en ligne de commande permettant d'utiliser ce protocole à différentes fins (accès à un système de fichier distant, création de tunnels SSH...).

© Editions ENI - All rights reserved

Concernant Python, il existe différents moyens d'utiliser SSH (http://wiki.python.org/moin/SecureShell), sans compter twisted dont nous parlerons dans un chapitre ultérieur, ce dernier étant plutôt un framework qu'une bibliothèque.
Pour les besoins de ce chapitre, nous allons créer un nouvel utilisateur sur la machine locale qui se nomme **testeur** et a pour mot de passe **passe** :

```
$ sudo adduser testeur
Ajout de l'utilisateur `testeur' ...
Ajout du nouveau groupe `testeur' (1001) ...
Ajout du nouvel utilisateur `testeur' (1001) avec le groupe `testeur' ...
Création du répertoire personnel `/home/testeur'...
Copie des fichiers depuis `/etc/skel'...
Entrez le nouveau mot de passe UNIX :
Retapez le nouveau mot de passe UNIX :
passwd : le mot de passe a été mis à jour avec succès
Modification des informations relatives à l'utilisateur testeur
Entrez la nouvelle valeur ou « Entrée » pour conserver la valeur proposée
   Nom complet []: Compte de test
   N° de bureau []:
   Téléphone professionnel []:
   Téléphone personnel []:
   Autre []:
Ces informations sont-elles correctes ? [O/n] O
```

Puis nous allons installer un serveur SSH :

```
$ sudo aptitude install ssh
```

Ceci dans le cas où l'on n'aurait pas de serveur disponible.
La bibliothèque paramiko est disponible pour Python 2 et 3 et dispose d'une documentation claire. Elle s'installe comme suit :

```
pip-3.2 install paramiko
```

Pour créer un client ssh, il suffit de quelques lignes :

```
>>> client = paramiko.SSHClient()
>>> client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
>>> client.connect('127.0.0.1', username='testeur', password='passe')
```

On commence par créer le client en mettant en place une politique permettant de déterminer ce que l'on fait lorsque l'on se connecte à un serveur pour la première fois (dans notre cas, on le rajoute automatiquement à notre liste de serveurs de confiance). Puis on se connecte en passant en paramètres le serveur, le nom d'utilisateur et le mot de passe.
Voici comment passer une commande et récupérer le résultat :

```
>>> i, o, e = client.exec_command('pwd')
```

On récupère un flux d'entrée, un flux de sortie et un flux d'erreur. Il faut alors fermer le flux d'entrée, lire le flux d'erreur pour vérifier que tout s'est bien passé (il est alors vide) et enfin, exploiter le flux de sortie pour récupérer le résultat :

```
>>> i.close()
>>> e.readlines()
```

```
[]
>>> o.readlines()
['/home/testeur\n']
```

Attention lorsque l'on souhaite effectuer plusieurs commandes, dans le cas où celles-ci sont liées. Chaque exécution lance une nouvelle connexion :

```
>>> i, o, e = client.exec_command('cd /var')
>>> i, o, e = client.exec_command('ls -l')
>>> i.close()
>>> e.readlines()
[]
>>> o.readlines()
['total 4\n', '-rw-r--r-- 1 testeur testeur 179 2011-10-07 17:25
examples.desktop\n']
```

Il faut donc exécuter les commandes dans le même pipe :

```
>>> i, o, e = client.exec_command('cd /var; ls -l')
>>> i.close()
>>> e.readlines()
[]
>>> o.readlines()
['total 24\n', 'drwxr-xr-x  2 root root  4096 2011-10-05 08:47 backups\n',
'drwxr-xr-x 23 root root  4096 2011-09-27 11:52 cache\n',
'drwxrwxrwt  2 root root     6 2011-04-26 01:07 crash\n',
'drwxr-xr-x  3 root root  4096 2011-07-16 01:05 games\n',
'drwxr-xr-x 80 root root  4096 2011-09-27 17:47 lib\n',
'drwxrwsr-x  2 root staff    6 2011-04-21 18:51 local\n',
'drwxrwxrwt  3 root root    80 2011-10-07 15:29 lock\n',
'drwxr-xr-x 22 root root  4096 2011-10-07 07:55 log\n',
'drwxrwsr-x  2 root mail    17 2011-10-02 13:53 mail\n',
'drwxr-xr-x  2 root root     6 2011-04-26 00:51 opt\n',
'drwxr-xr-x 16 root root   760 2011-10-07 18:09 run\n',
'drwxr-xr-x 10 root root   129 2011-09-02 16:50 spool\n',
'drwxrwxrwt  2 root root     6 2011-10-06 15:16 tmp\n',
'drwxr-xr-x  6  root root   80 2011-09-29 16:20 www\n']
```

2.7 POP et POPS

POP est l'acronyme de *Post Office Protocol*, soit le protocole du bureau de poste et est défini dans la RFC 1939. Il permet à un client de récupérer les messages disponibles sur un serveur de messagerie électronique via TCP. Il existe aussi une possibilité de sécurisation des échanges entre serveur et client via une couche SSL décrite dans la RFC 2595.

Python propose une bibliothèque native pour gérer ces problématiques (elle est donc disponible pour la branche 3.x). Cette dernière permet la gestion des protocoles POP3 et POP3S (non sécurisé et sécurisé) via les classes **POP3** et **POP3_SSL** (http://docs.python.org/py3k/library/poplib.html).

© Editions ENI - All rights reserved

Voici comment créer et utiliser un client POP :

```
>>> client = poplib.POP3('pop.monfournisseur.com')
>>> client.user('monidentifiant')
b'+OK name is a valid mailbox'
>>> client.pass_('monmotdepasse')
b'+OK user exist with that password'
>>> client.getwelcome()
b'+OK connected to pop3 on 8304 '
```

On obtient donc des renseignements utiles au fur et à mesure que l'on avance dans le processus.
Il est possible de savoir combien de messages sont dans la boîte aux lettres électronique et quelle place ils occupent :

```
>>> client.stat()
(718, 23695518)
```

La boîte contient donc 718 messages !
Il est possible de lister l'ensemble des messages disponibles :

```
>>> client.list()
(b'+OK scan listing follows', [b'1 39358', b'2 7118', [...], b'718
4837'], 7517)
```

Il est important de voir que la règle est l'association entre le numéro du message et la place qu'il occupe, sachant que le premier message est le plus ancien.
C'est cette donnée qu'il faut fournir lorsque l'on souhaite réaliser n'importe quelle opération sur un message. Ces opérations sont **retr** pour récupérer le message, **dele** pour le supprimer, **top** pour avoir le début du message.
Lorsqu'un message est lu, un flag est positionné pour marquer le message comme lu, mais rien ne se fait dans l'immédiat. Lorsqu'un message est supprimé, il est marqué comme supprimé. C'est le fait de fermer la connexion explicitement qui valide les changements. Pour les annuler, il faut utiliser **rset**, tandis que **noop** évite le timeout.

```
>>> client.quit()
b'+OK'
```

2.8 IMAP et IMAPS

IMAP est l'acronyme de *Internet Message Access Protocol*, soit le procotole d'accès aux messages électroniques. Si POP3 est conçu pour se connecter à un serveur de courrier électronique, rapatrier les messages et quitter, IMAP est conçu pour permettre une plus grande souplesse et lire les messages sans forcément les rapatrier, mais requiert une connexion permanente (un mode hors connexion existe également, mais n'est pas le mode par défaut). Comme POP3, il n'est pas sécurisé, mais il existe une couche SSL. Les RFC 3501 et 2595 décrivent ces protocoles.
Python propose une bibliothèque native portée sous la branche 3.x qui permet de gérer ces problématiques au travers des classes **IMAP4** et **IMAP4_SSL** (http://docs.python.org/py3k/library/imaplib.html).

Voici comment créer et utiliser un client IMAP :

```
>>> import imaplib
>>> client = imaplib.IMAP4('imap.monfournisseur.com')
>>> client.login('monidentifiant', 'monmotdepasse')
('OK', [b'User logged'])
```

IMAP permet de gérer sa boîte aux lettres électronique relativement simplement et de manière assez avancée. Par exemple, on peut avoir la liste des différentes boîtes disponibles :

```
>>> client.lsub()
('OK', [b'() "." "INBOX.DRAFT"', b'() "." "INBOX.OUTBOX"', b'()
"." "INBOX.QUARANTAINE"', b'() "." "INBOX.TRASH"'])
```

On peut aussi savoir dans quel dossier de la boîte de réception l'on se trouve :

```
>>> client.namespace()
('OK', [b'(("INBOX." ".")) NIL NIL'])
```

Faire un choix (par défaut **`INBOX.`**), la réponse contient le nombre de messages :

```
>>> client.select('INBOX.OUTBOX')
('OK', [b'118'])
>>> client.check()
('OK', [b'Completed'])
```

Récupérer les informations sur les mails disponibles (via une recherche) :

```
>>> typ, data = client.search(None, 'ALL')
>>> data
[b'1 2 3 4 [...] 118']
```

La recherche peut être plus fine. Voici comment récupérer un message :

```
>>> num = data[0].split()[0]
>>> client.fetch(num, '(RFC822)')
('OK', [(b'1 (RFC822 {3320}', b'Date: [...]--\r\n'), b')'])
```

Il ne reste plus qu'à quitter proprement :

```
>>> client.close()
('OK', [b'Completed'])
>>> client.logout()
('BYE', [b'LOGOUT received'])
```

2.9 SMTP et SMPTS

SMTP est l'acronyme de *Simple Mail Transfer Protocol*, soit le protocole simple de transport de message. En d'autres termes, ce protocole est chargé d'acheminer les courriers électroniques, ou courriels, de l'expéditeur vers le destinataire.

Ce processus est le cœur du système gérant les courriels, puisque c'est lui qui se charge d'acheminer le courrier et qui, dans le cœur de ces courriels, précise qui est le destinataire, mais également qui est l'expéditeur et d'autres méta-informations.

© Editions ENI - All rights reserved

L'impossibilité d'identifier de manière certaine l'expéditeur est une des plus grandes limitations du protocole SMTP et la cause la plus importante des pourriels qui est un des problèmes majeurs à résoudre. Le très grand développement du réseau de distribution de courriels qui a commencé très tôt dans l'histoire de l'informatique, la diversité des solutions déployées et leur hétérogénéité rend la résistance au changement particulièrement importante. La résolution de ce problème s'en trouve plus ardue, même si diverses solutions émergent ou ont émergé et parmi elles, certaines ont une chance réelle de s'imposer.
Les serveurs SMTP libres les plus répandus et que vous pouvez très facilement installer sur votre machine et configurer sont **sendmail**, **exim** et **postfix**. Le plus simple au niveau de la configuration est de les configurer en tant que relais.
Python propose une bibliothèque native portée sous la branche 3.x qui permet de gérer ces problématiques au travers des classes **SMTP** et **SMTP_SSL** (http://docs.python.org/py3k/library/smtplib.html).
Voici comment créer et utiliser un client SMTP :

```
>>> import smtplib
>>> client = smtplib.SMTP('smtp.monfournisseur.com')
```

À partir de là, il faut bien connaître le protocole pour l'utiliser :

```
>>> client.helo()
(250, b'XXXinfosXXX hello [127.0.0.1], pleased to meet you')
```

Par exemple, depuis le RFC 2821, **helo** est remplacé par **ehlo**, il faut donc :

```
>>> client = smtplib.SMTP('smtp.laposte.net')
>>> client.ehlo()
(250, b'XXXinfosXXX hello [127.0.0.1], pleased to meet
you\nHELP\nAUTH LOGIN PLAIN\nSIZE
44000000\nENHANCEDSTATUSCODES\n8BITMIME\nOK')
```

On a ici plus d'informations. La réponse que l'on reçoit conditionne la suite du processus à donner. Le scénario classique de l'enchaînement des commandes est EHLO, puis MAIL_FROM, RCPT_TO, pour terminer par DATA.
L'ensemble s'opère, pour Python, en une seule fois, par l'utilisation de la méthode sendmail qui prend en paramètre l'expéditeur, le ou les destinataires et le message, mais ce message doit être construit. Ce n'est pas simplement du texte, c'est une chaîne de caractères formatée selon la RFC 5322 et qui contient entre autres l'expéditeur et le ou les destinataires. Ce sont des informations qui sont utilisées pour faire circuler le message d'un serveur à un autre jusqu'à destination.
Il ne faut pas oublier que si l'on manipule une chaîne de caractères, dans le réseau, ce sont des bytes qui circulent, d'où la meilleure cohérence de Python 3 dans ce domaine qui permet de faire la différence entre des bytes que l'on envoie ou que l'on reçoit du réseau et les chaînes de caractères que l'on peut manipuler dans un autre contexte.
Voici le template d'un message basique :

```
>>> message_template = 'From: %s\r\nTo: %s\r\n\r\n%s'
```

Voici comment envoyer un message très simple avec Python :

```
>>> sender = 'moi@monfournisseur.com'
>>> dest = 'toi@monfournisseur.com'
>>> message = 'Hello World!'
```

Une fois les paramètres mis en place, la procédure est simple :

```
>>> client = smtplib.SMTP('smtp.monfournisseur.com ')
```

Comme nous l'avons vu précédemment, il est possible d'activer un mode pour rendre les réponses plus ou moins bavardes :

```
>>> client.set_debuglevel(1)
```

Lorsque l'on envoie un mail, l'utilisation du template montre clairement le besoin de préciser à deux endroits différents les expéditeurs et destinataires :

```
>>> client.sendmail(sender, dest, message_template % (sender,
dest, message))
```

La réponse montre le protocole, qui est tel que précisé plus haut (ne sont gardés ici que les **send**, et les **reply** utiles à la compréhension) :

```
send: 'ehlo [127.0.1.1]\r\n'
[...]
send: 'mail FROM:<moi@monfournisseur.com> size=79\r\n'
[...]
send: 'rcpt TO:<toi@monfournisseur.com>\r\n'
reply: b'550 5.1.1 Authentification requise.
Authentication Required. LPN104_402 [402]\r\n'
send: 'rset\r\n'
reply: b'250 2.0.0 OK\r\n'
```

La réponse reçue est traitée par Python et déclenche une erreur logique :

```
Traceback (most recent call last):
[...]
b'5.1.1 Authentification requise. Authentication Required.
LPN104_402 [402]')}
```

On peut donc fermer proprement la connexion.

```
>>> client.quit()
send: 'quit\r\n'
reply: b'221 2.0.0 XXXinfosXXX ME closing connection\r\n'
reply: retcode (221); Msg: b'2.0.0 mwinf8511-out ME closing connection'
(221, b'2.0.0 XXXinfosXXX ME closing connection')
```

On voit clairement que le traitement des erreurs bas niveau est bien géré.
Voici maintenant la même chose, avec l'authentification :

```
>>> client = smtplib.SMTP('smtp.monfournisseur.com ')
>>> client.login('monidentifiant', 'monmotdepasse')
(235, b'2.7.0 ... authentication succeeded')
```

© Editions ENI - All rights reserved

Maintenant, on peut recommencer la procédure à l'identique :

```
>>> client.set_debuglevel(1)
>>> client.sendmail(sender, dest, message_template % (sender,
dest, message))
send: 'mail FROM:<moi@monfournisseur.com> size=79\r\n'
send: 'rcpt TO:<toi@monfournisseur.com>\r\n'
send: 'data\r\n'
send: b'From: moi@monfournisseur.com\r\nTo:
toi@monfournisseur.com\r\n\r\nHello World!\r\n.\r\n'
data: (250, b'2.0.0 hw1p1h0xxxxxxx03w2VWN mail accepted for delivery')
{}
```

Et on ferme la connexion proprement :

```
>>> client.quit()
send: 'quit\r\n'
reply: b'221 2.0.0 XXXinfosXXX ME closing connection\r\n'
reply: retcode (221); Msg: b'2.0.0 mwinf8511-out ME closing connection'
(221, b'2.0.0 XXXinfosXXX ME closing connection')
```

On a donc vu comment envoyer un mail, de manière simple.
Il peut se produire plusieurs erreurs, dont l'adresse d'un destinataire qui n'est pas correcte ou le nom de domaine qui n'existe pas.
Le premier cas n'est pas détecté, c'est le fournisseur réceptionnant le mail qui renvoie un message à l'envoyeur pour lui signaler l'erreur. Le second cas génère une erreur :

```
smtplib.SMTPRecipientsRefused:
{'existepas@existepassinoncestpasdechance.com': (550, b'5.1.1
Adresse d au moins un destinataire invalide. Invalid recipient.
LPN104_418 [418]')}
```

Il existe une autre commande VRFY qui permet de vérifier l'existence d'une adresse de courriel, mais elle est souvent désactivée parce que si elle est utile pour nous, elle l'est également, malheureusement, pour les envoyeurs de pourriels. Elle est utilisable par la méthode **`verify`**, mais elle n'est pas fiable, étant donné ce que l'on a dit sur les pourriels.
Pour la suite, il suffit de bien connaître le protocole pour savoir comment répondre aux différents cas d'utilisation. Par exemple, il est possible d'ajouter dans le message l'en-tête **`Subject`** pour ajouter un objet au courriel.

```
>>> message_template='From: %s\r\nTo: %s\r\nSubject: %s\r\n\r\n%s'
>>> subject = 'Test envoi de mail'
>>> client = smtplib.SMTP('smtp.fournisseur.com')
>>> client.login('monidentifiant', 'monmotdepasse')
>>> client.sendmail(sender, dest, message_template % (sender,
dest, subject, message))
>>> client.quit()
```

Jusqu'à présent, on n'a réalisé que des choses simples. Mais les choses peuvent être plus complexes et il est nécessaire de les gérer plus professionnellement.

Ainsi, il existe des RFC qui décrivent la structure de ces messages et des modules Python dédiés. Autant les utiliser :

```
>>> from email.mime.text import MIMEText
>>> msg = MIMEText(message)
>>> msg['From'] = sender
>>> msg['To'] = dest
>>> msg['Subject'] = subject
```

Du coup, la création du message est beaucoup plus claire et plus simple. Et l'envoi reste identique, à l'utilisation du message près :

```
>>> client = smtplib.SMTP('smtp.monfournisseur.com ')
>>> client.login('monidentifiant', 'monmotdepasse')
(235, b'2.7.0 ... authentication succeeded')
>>> client.sendmail(sender, dest, msg.as_string()){}
>>> client.quit()
(221, b'2.0.0 mwinf8511-out ME closing connection')
```

Cela permet alors d'ajouter des pièces jointes assez facilement, ce qui est un besoin rencontré régulièrement. Pour cela, il faut une classe adaptée :

```
>>> msg.is_multipart()
False
>>> from email.mime.multipart import MIMEMultipart
>>> msg = MIMEMultipart(message)
>>> msg.is_multipart()
True
```

Rajouter une pièce jointe devient alors d'une facilité déconcertante :

```
>>> msg['From'] = sender
>>> msg['To'] = dest
>>> msg['Subject'] = 'Test de messages avec PJ'
>>> with open('test.txt', 'r') as f:
...     msg.attach(MIMEText(f.read()))
...
```

On envoie alors le message exactement de la même manière que précédemment.

À noter qu'il existe MIMEAudio, MIMEImage, MIMEApplication, MIMEMessage et MIMEBase pour les autres types. Il est possible d'aller encore plus loin sur ce sujet en consultant la documentation officielle et en particulier la page d'exemples (https://docs.python.org/3/library/email.examples.html).

Un point important est l'envoi de mails aux formats texte et HTML et l'inclusion d'images à l'intérieur du mail HTML non pas en mettant des liens vers des sites externes, mais en les embarquant.

Pour cela, encore une fois, il faut bien connaître le fonctionnement des courriels. Il y a juste une petite marche de difficulté supplémentaire. Le point important est l'utilisation de **cid**, pour le reste, tout est conforme à ce que l'on a déjà vu.

Le cas est traité dans un exemple complet dans le fichier **courriel.py**.

© Editions ENI - All rights reserved

2.10 NNTP

NNTP est le sigle de *Network News Transfer Protocol*. Il s'agit d'un protocole réseau disposé au sein de la couche application du modèle OSI qui gère le transfert de contenu depuis ou vers des serveurs de news.
Les serveurs de news écoutent sur le port 119 lorsqu'ils ne sont pas sécurisés et 563 autrement. Là encore, le module ne peut être maîtrisé que si le protocole l'est.
Ce protocole suit des règles similaires aux autres protocoles présentés dans cette section et cette similarité se ressent côté Python.
Le module se nomme **nntplib** et il existe la classe **NNTP** et la classe **NNTP_SSL**. Il existe également une classe **NNTPError** qui gère les exceptions que l'utilisation de **NNTP** peut provoquer.
NNTP permet donc de se connecter à un serveur :

```
>>> import nntplib
>>> s = nntplib.NNTP('news.mydomain.org')
```

D'avoir des informations sur le serveur :

```
>>> s.getwelcome()
'200 news.mydomain.org NNTP server ready (posting ok)'
>>> s.getcapabilities()
```

Il permet de retrouver des informations sur un groupe :

```
>>> resp, count, first, last, name = s.group('python.myprojects')
>>> print('Le groupe %s a %s articles de %s à %s' % (name, count,
first, last))
Le groupe python.myprojects a 182 articles de 1 à 182
```

Ainsi que les dernières actualités :

```
>>> resp, overviews = s.over((last - 2, last))
>>> for id, over in overviews:
...     print(id, nntplib.decode_header(over['subject']))
...
180 Re: [Hadron] Optimisations
181 Re: [Hadron] Optimisations
182 Re: [Hadron] Optimisations
```

Il est possible de se connecter avec des identifiants en procédant comme suit :

```
>>> s = nntplib.NNTP(newsserver, user=user, password=password)
```

Voici comment récupérer le dernier article, par exemple :

```
>>> resp, overviews = s.over((last, last))
>>> for id, over in overviews:
>>>     info, article = s.article(id)
>>>     with open('article %s.txt' % over['subject'], 'w+') as f:
>>>         f.write(['%s\n' % l for l in article[2]])
```

Voici enfin comment fermer proprement la connexion :

```
>>> s.quit()
```

2.11 IRC

IRC est le sigle de *Internet Relay Chat*, soit discussion relayée par Internet et est un protocole de communication permettant à l'origine l'échange de texte. Il est utilisé pour discuter via Internet, tous les utilisateurs se connectant sur serveur IRC à l'aide de clients IRC (il en existe des gratuits sur toutes les plateformes, mais également quelques-uns qui sont libres).

Les utilisateurs peuvent participer à un ou plusieurs canaux de discussion et peuvent également parler directement entre eux (discussions privées).

IRC peut permettre d'autres échanges que la simple discussion. Il permet d'identifier un utilisateur, de gérer la connexion et déconnexion à un canal, l'expulsion, le bannissement, et l'échange de fichiers voire même plus. En effet, le client peut, au lieu de simplement parler, envoyer une chaîne qui sera comprise comme une commande et interprétée par un BOT. Ainsi, il est possible d'aller sur des canaux qui partagent des fichiers, et il est possible de donner des ordres à un BOT, qui est un robot qui traîne sur un canal et qui analyse tout ce qu'il se passe dans le canal et réagit si nécessaire.

Les exemples les plus connus sont des bots qui expulsent ceux qui écrivent des insultes, trop de majuscules ou des mots interdits. Il y en a aussi qui proposent des quizz, c'est-à-dire qui posent des questions et qui analysent les phrases pour déterminer les bonnes réponses et établir des statistiques sur les joueurs.

Python dispose d'une bibliothèque externe qui est dédiée à ce genre d'opérations. Pour Python 2, elle s'installe ainsi :

```
$ sudo aptitude install python-irclib
```

Pour Python 3, il faut utiliser **`pip`** :

```
$ sudo pip-3.2 install irc
```

Le protocole IRC est particulièrement complet et propose beaucoup de choses. Python permet de ne s'intéresser qu'à une partie et propose une classe toute prête pour gérer des bots. Celle-ci contient des hooks qu'il ne reste plus qu'à remplir pour que notre bot puisse réagir à une situation donnée.

Voici les hooks principaux :

- on_welcome : à l'arrivée sur le serveur ;
- on_join : à l'arrivée sur un canal ;
- on_pubmsg : à la réception d'un message public (sur un canal) ;
- on_privmsg : à la réception d'un message privé ;
- on_action : à la réception d'un message contenant **`/me`** ;
- on_kick : à la réception d'une instruction d'expulsion.

Ces hooks sont intimement liés à des événements, gérés par une classe dédiée.

Toutes les signatures des fonctions comportent l'objet qui permet de réaliser une action et l'événement que l'on peut interroger pour avoir plus d'informations.

© Editions ENI - All rights reserved

Par exemple, voici comment avoir les canaux ou utilisateurs concernés :

```
>>> src, dst = ev.source(), ev.target()
```

Et voici comment avoir les arguments dont potentiellement le message :

```
>>> message = ev.arguments()[0]
```

Ensuite, toute la problématique consiste à traiter correctement le message pour savoir quelle réponse il convient d'apporter. C'est de loin la partie la plus complexe, et elle est finalement connexe de la problématique purement IRC.
Une fois que l'on a déterminé la réponse à apporter, il faut utiliser l'action :
– envoyer un message public (le destinataire est un canal) :

```
>>> serv.privmsg("#channel-irc", "message")
```

– envoyer un message privé (le destinataire est un pseudo d'utilisateur) :

```
>>> serv.privmsg("utilisateur", "message")
```

– envoyer une action (privée ou publique suivant le même principe) :

```
>>> serv.action("#channel-irc", "message")
```

– joindre un canal :

```
>>> serv.join("#channel-irc")
```

– joindre un canal protégé :

```
>>> serv.join("#channel-irc", "motdepasse")
```

– quitter un canal :

```
>>> serv.part("#channel-irc")
```

– retrouver son pseudonyme actuel :

```
>>> serv.get_nickname()
```

– changer de pseudonyme :

```
>>> serv.nick("#nouveau_pseudo")
```

– envoyer une notice (même principe public/privé pour le destinataire) :

```
>>> serv.notice("#channel-irc", "message")
```

– inviter un utilisateur à rejoindre un canal :

```
>>> serv.invite("pseudonyme", "#channel-irc")
```

– se déconnecter :

```
>>> serv.disconnect("message")
```

– expulser quelqu'un (si on est opérateur) :

```
>>> serv.kick("#channel-irc", "pseudonyme", "message")
```

– utiliser les modes :

```
>>> serv.mode("#channel-irc", mode)
```

Avec tout ceci, on a tout le nécessaire pour réaliser les actions à entreprendre en réponse à une situation donnée.
Il ne manque plus qu'un détail, qui est le traitement des utilisateurs.
Lorsque l'on traite avec une source ou une cible, on se retrouve avec une donnée complexe qui est le « nick », en allant regarder source.nick.
Ceci permet de s'assurer un minimum de la provenance des messages et de faire des vérifications de manière à ne traiter des commandes critiques que lorsqu'elles sont exécutées par une source sûre. Après, tout dépend de la manière dont on contrôle la sûreté de la source.
On peut également s'amuser à détecter ceux qui ouvrent deux connexions à partir du même ordinateur et qui utilisent deux pseudonymes différents.
Voici maintenant un minuscule exemple. Il s'agit d'un bot qui traite de « boulet » tous ceux qui disent « comment » et « faire » dans la même phrase.

```
#!/usr/bin/env python3
# -*- coding: utf8 -*-


import irc.bot as ircbot

connexion = "irc.mondomaine.org", 6667

class BotBoulet(ircbot.SingleServerIRCBot):
    def __init__(self):
        ircbot.SingleServerIRCBot.__init__(self, [connexion],
 "botpseudo", "Bot réalisé en Python")
    def on_welcome(self, serv, ev)
        serv.join("#channel-irc")
    def on_pubmsg(self, serv, ev):
        pseudo = ev.source.nick
        canal = ev.target()
        message = ev.arguments()[0].lower()
        if 'comment' in message and 'faire' in message:
            serv.privmsg(canal, "%s est un boulet!" % pseudo)
        if 'boulet' in message:
            serv.kick(canal, pseudo, "Il n'y a qu'un seul
boulet ici et il vient de partir!)

if __name__ == "__main__":
    BotBoulet().start()
```

On peut également imaginer de créer une commande spéciale, que l'on identifierait par les premiers caractères du message puis par une série de données, qui déclenche l'utilisation d'un service web. Le bot permettrait alors de fournir des informations comme le programme télé, la dernière actu d'un site particulier ou autre en répondant à des informations messages du style :

```
!tv nolife-tv
!actu
```

© Editions ENI - All rights reserved

3. Wake-on-LAN

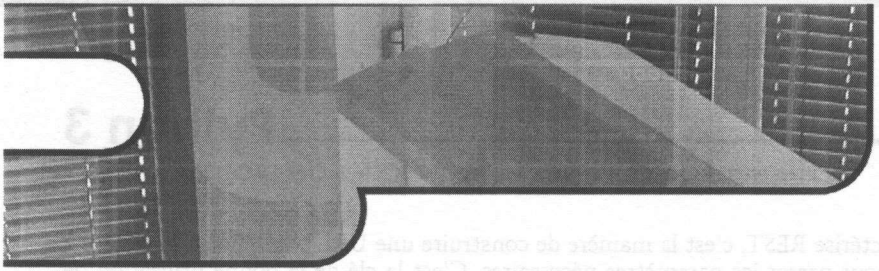
3.1 Prérequis

Wake-on-LAN est un standard réseau qui permet à un ordinateur éteint d'être démarré à distance.
Pour cela, ce matériel doit avoir une carte mère qui supporte le Wake-on-LAN et une carte réseau qui soit adaptée (actuellement, elles sont directement intégrées à la carte mère, mais sinon, il faut un connecteur supplémentaire).
Il faut également paramétrer le BIOS de l'ordinateur pour que celui-ci accepte les requêtes Wake-on-LAN, mais également le système d'exploitation (paramétrage de la carte réseau, rela-tive à la gestion de l'alimentation).
Le fonctionnement du protocole implique que le routeur de réseau associe l'adresse MAC à une adresse IP. Sans quoi, il est nécessaire de réaliser du broadcast, ce que nous allons faire.

3.2 Mise en œuvre

Le principe du Wake-on-LAN est d'émettre sur le réseau un paquet magique qui, lorsque reçu par sa cible, la réveille.
La caractéristique magique de ce paquet est qu'il contient six fois FF, suivi de seize répétitions de l'adresse MAC de la cible à réveiller.
Voici sa construction :

```
>>> adresse_mac = b'C4:DF:12:58:4E:FC'
>>> paquet_magique = b'\xFF' * 6 + b''.join([int(i, 16).
to_bytes(1, byteorder='little') for i in adresse_mac.split(b':')]) * 16
```



Chapitre 2.3
Programmation web

1. Services web

1.1 REST

Un service web est une fonctionnalité qui permet un échange de données entre applications qui peuvent être hétérogènes. Cet échange peut avoir lieu sur un intranet ou sur un Internet et est réalisé de manière synchrone.

La liste des technologies mises en œuvre est assez diverse dans plusieurs domaines.

REST est une technologie qui définit un style d'architecture applicative et est en cela également une manière de représenter une ressource susceptible d'être interrogée et de recevoir des paramètres.

Le transport est effectué via HTTP, la réponse peut prendre plusieurs formes qui sont HTML, XML ou JSON.

Le moyen le plus simple et le plus rapide pour offrir un service web renvoyant une réponse JSON est d'utiliser la bibliothèque bottle que vous installerez ainsi :

```
pip install bottle
```

Voici un exemple basique de service web que l'on peut réaliser à l'aide de cette bibliothèque :

```
from bottle import route, run
if __name__ == "__main__":
    @route('/hello/:name', method='GET')
    def get_set_of_results(name):
        return {'message': 'Hello %s' % name}
    run()
```

Pour lancer le serveur, il faut simplement aller dans le répertoire 13 des exemples et exécuter :

```
$ ./test_rest.py
```

Ce qui caractérise REST, c'est la manière de construire une URL pour choisir le service à utiliser et pour passer les paramètres nécessaires. C'est la clé de la bonne utilisation de REST et c'est relativement simple à utiliser en plus d'être finalement pythonique bien que pas conçu spécifiquement pour Python.
Il suffit d'aller ouvrir son navigateur et de taper l'URL suivante pour visualiser le résultat (fichier JSON) :

```
http://localhost/hello/quelquechose
```

Voici un client qui permet de consommer un tel service :

```
>>> from http.client import HTTPConnection
>>> conn = HTTPConnection('localhost:8080')
>>> conn.request('GET', '/hello/world')
>>> response = conn.getresponse()
>>> print(response.status)
200
>>> print(response.reason)
OK
>>> resultat = eval(response.read())
>>> resultat
{'message': 'Hello world'}
```

Cet exemple de client est un exemple qui se situe sur le bas niveau et n'utilise que les bibliothèques fournies avec Python 3. Il existe également des librairies de plus haut niveau qui gèrent les erreurs et les problématiques spécifiques à l'utilisation du REST.
La bibliothèque **bottle** est un micro-framework web qui peut servir à faire des sites web ou des services web. Cette bibliothèque est minimaliste : elle ne fait que servir du contenu web. Elle peut cependant être associée à d'autres paquets Python pour gérer les moteurs de templates (permettant de générer des pages web), comme **mako**, **jinja2** ou **cheetah** ou l'accès à une base de données via l'utilisation d'un client ou d'un ORM. Vous trouverez plus d'informations sur le site : https://bottlepy.org/docs/dev/.
Notons que tous les frameworks web qu'il s'agisse de micro-frameworks comme **Flask**, de frameworks plus complets comme **Turbogears** ou **Pyramid** ou encore de framework fullstack comme **Django** permettent d'écrire des services web.

1.2 SOAP

SOA est le sigle de *Service Oriented Architecture*, soit architecture orientée service. C'est également une technologie qui définit un style d'architecture dont le but est d'appeler une fonction comme si celle-ci faisait partie d'une bibliothèque locale via l'invocation du service web qui fournit la fonctionnalité réalisée avec SOAP.
SOAP est l'acronyme de *Simple Object Access Protocol*, qui est un protocole XML-RPC permettant la transmission de messages entre objets distants via HTTP.
Ce protocole est simplement la succession de XML-RPC, son évolution.
La bibliothèque la plus simple d'utilisation pour créer un client SOA en Python 3 est pysimplesoap. Elle s'installe ainsi :

```
$ pip-3.2 install pysimplesoap
```

© Editions ENI - All rights reserved

Elle fonctionne également pour Python 2 et s'installe toujours avec pip, en utilisant sa version liée à Python 2 (pip-2.7, par exemple).
Voici un exemple écrit pour Python 3 qui réalise la même opération que précédemment. Tout d'abord, il faut réaliser des imports :

```
from pysimplesoap.server import SoapDispatcher, SOAPHandler
from http.server import HTTPServer
```

Attention, pour Python 2, il faudra aller chercher cet objet ici :

```
from BaseHTTPServer import HTTPServer
```

On notera que la partie serveur est réalisée simplement par le module de Python 3 permettant de créer un serveur HTTP et que pysimplesoap ne s'occupe que de la partie relative au protocole SOAP lui-même.
Voici une fonction d'exemple qui est une méthode de notre serveur SOAP :

```
def hello(name):
    return {'message': 'hello %s' % name}
```

La première étape consiste à créer un dispatcher :

```
dispatcher = SoapDispatcher(
    'my_dispatcher',
    location = "http://localhost:8008/",
    action = 'http://localhost:8008/', # SOAPAction
    namespace = "http://example.com/sample.wsdl", prefix="ns0",
    trace = True,
    ns = True)
```

Une fois que ce dernier est créé, il faut procéder à l'enregistrement de la fonction que les clients pourront appeler :

```
dispatcher.register_function('Hello', hello,
    returns={'Message': str},
    args={'name': str})
```

On précise ici les noms et types des arguments attendus ainsi que celui de la réponse. Maintenant, il ne reste plus qu'à démarrer le serveur :

```
if __name__ == "__main__":
  server = HTTPServer(("", 8008), SOAPHandler)
  server.dispatcher = dispatcher
  server.serve_forever()
```

Pour lancer le serveur, il faut simplement aller dans le répertoire 14 des exemples et exécuter :

```
$ ./test_soa.py
```

Voici maintenant la partie cliente. On commence par quelques imports :

```
from pysimplesoap.client import SoapClient, SoapFault
```

Puis, on crée notre objet client en lui donnant les mêmes informations que celles données au serveur :

```
client = SoapClient(
location = "http://localhost:8008/",
action = 'http://localhost:8008/', # SOAPAction
namespace = "http://example.com/sample.wsdl",
soap_ns='soap',
ns = False)
```

On notera aussi l'existence de **rinse** (http://pythonhosted.org/rinse/) ainsi que de **spyne** (http://spyne.io) qui sont utilisables de manière plus large que le simple SOAP.
On peut maintenant utiliser la méthode partagée par le serveur en utilisant le nom des variables précisées dans le dispatcher :

```
response = client.Hello(name="world")
```

Il ne reste plus qu'à afficher le résultat en se référant au nom précisé côté serveur :

```
print(response.Message)
'hello world'
```

On voit maintenant la sémantique utilisée et la manière de créer un dispatcher, un serveur et un client. Le serveur peut potentiellement partager autant de méthodes que souhaité, lesquelles se distinguent par leurs noms au niveau du dispatcher.

1.3 Pyro

Pyro est l'acronyme de *Python Remote Objects* et est également une architecture orientée service permettant à des objets de communiquer entre eux à travers le réseau. Le but est de fournir un minimum d'efforts et que les objets distants soient utilisables exactement de la même manière que les objets locaux. La nouvelle version est la branche 4.x et elle brise la compatibilité avec les anciennes. Pour cette raison, le module s'appelle maintenant **Pyro4** et non plus **Pyro**. Les performances sont très bonnes.
Par rapport à l'utilisation de SOAP, l'inconvénient majeur est que seuls les programmes Python peuvent utiliser Pyro alors que SOAP peut permettre le dialogue entre des applications hétérogènes. L'avantage est que l'on n'est pas obligé d'encapsuler l'objet de la manière dont cela a été fait pour SOAP, c'est-à-dire renvoyer un dictionnaire qui sera utilisé par la partie cliente comme un objet classique. Pyro est en ce sens plus simple, plus naturel.
L'avantage également est que Pyro fonctionne sur Python 2.x et Python 3.x, mais également sur CPython, IronPython, Jython et Pypy. En d'autres termes, l'utilisation de Pyro permet de créer une architecture de programme souple en autorisant à avoir une partie gérant des fonctionnalités non portées sur Python 3.x écrites en Python 2.x et rendues disponibles via Pyro. Cela peut être également le cas pour des fonctionnalités utilisant des .jar écrites en Jython et servies de la même manière, par exemple.
Voici comment installer Pyro pour Python 2.x :

```
$ sudo easy_install Pyro4
```

© Editions ENI - All rights reserved

Et voici pour l'installer pour Python 3.x :

```
$ sudo easy_install3 Pyro4
```

Voici la partie serveur qui réalise la même opération que le serveur SOAP :

```
#!/usr/bin/python3
import Pyro4
if __name__ == "__main__":
    class Service:
            def hello(self, name):
                    return 'hello %s' % name
    daemon=Pyro4.Daemon()
    uri=daemon.register(Service())
    print("URI: %s" % uri)
    daemon.requestLoop()
```

Voici ce que l'on obtient en lançant le serveur (./test_pyro.py) :

```
$ ./test_pyro.py
URI:  PYRO:obj_cfdec571525a4d21b8135767e006063e@localhost:38221
```

Voici la partie cliente :

```
>>> import Pyro4
>>> service = Pyro4.Proxy(
    'PYRO:obj_cfdec571525a4d21b8135767e006063e@localhost:38221')
>>> service.hello('World')
'hello World'
```

2. Client web

Nous avons présenté dans le chapitre Programmation réseau la manière de construire un client web en utilisant le protocole HTTP. Ce chapitre présente un module de haut niveau qui est à la fois particulièrement bien adapté au protocole HTTP et qui ne vous impose pas de connaître le fonctionnement de ce protocole dans le détail.
Il s'agit du module **`requests`** qui s'installe ainsi :

```
$ pip install requests
```

Ce module a pour objectif de permettre aux non experts d'utiliser le protocole HTTP de la manière la plus simple possible.
On peut ainsi récupérer le contenu d'un service web simplement ainsi :

```
>>> import requests
>>> r = requests.get("http://127.0.0.1:8080/hello/world")
```

L'objet `r` est la réponse du serveur (on utilise ici le serveur que l'on a conçu plus haut en tant qu'exemple REST avec bottlepy).
On peut tout d'abord analyser cette réponse :

```
>>> r.status_code
200
>>> r.headers
{'Date': 'Sun, 19 May 2019 10:27:23 GMT',
```

```
 'Server': 'WSGIServer/0.2 Cpython/3.7.1',
 'Content-Type': 'application/json',
 'Content-Length': '26'}
```

Le code de statut est un élément important dans le protocole HTTP. En fonction de ce code, vous devrez traiter la réponse de manière différente. Un code 200 indique que tout s'est bien passé et que vous avez la réponse que vous souhaitez.
Un code 300 indique cependant un souci pour obtenir la ressource, que ce soit une permission, un proxy, le fait que la page a été déplacée de manière temporaire ou permanente... Vous devez agir en conséquence en fonction du code exact. Un code 400 est une erreur qui est considérée comme étant du côté client et une erreur 500 est une erreur côté serveur, sur laquelle vous ne pouvez agir.
Les en-têtes vous permettent de savoir si le contenu de votre réponse est du JSON ou du texte et c'est ce qui va déterminer comment vous allez le lire (l'élément essentiel est `Content-Type`). Dans notre cas, c'est du JSON, donc :

```
>>> r.json()
{'message': 'Hello world'}
```

Ceci renvoie un objet de type dictionnaire, directement exploitable en Python.
Voici un autre exemple :

```
>>> r = requests.get("http://127.0.0.1:8080/")
>>> r.status_code
404
>>> r.headers
{'Date': 'Sun, 19 May 2019 10:34:37 GMT',
 'Server': 'WSGIServer/0.2 Cpython/3.7.1',
 'Content-Length': '720',
 'Content-Type': 'text/html; charset=UTF-8'}
>>> print(r.text)

    <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
    <html>
        <head>
            <title>Error: 404 Not Found</title>
            <style type="text/css">
              html {background-color: #eee; font-family: sans;}
              body {background-color: #fff; border: 1px solid #ddd;
                    padding: 15px; margin: 15px;}
              pre {background-color: #eee; border: 1px solid #ddd;
                   padding: 5px;}
            </style>
        </head>
        <body>
            <h1>Error: 404 Not Found</h1>
            <p>Sorry, the requested URL
               <tt>&#039;http://127.0.0.1:8080/&#039;</tt>
               caused an error:</p>
            <pre>Not found: &#039;/&#039;</pre>
```

© Editions ENI - All rights reserved

```
        </body>
    </html>
```

Ici, le code de statut est 404 (la page demandée n'existe pas) et le contenu est de type texte. On va donc afficher du texte, qui est en fait un code HTML présentant un message d'erreur lisible pour l'utilisateur d'un navigateur.
Enfin, il se peut que le contenu soit binaire (par exemple une photographie) :

```
>>> r = requests.get(url)
>>> r.status_code
200
>>> r.headers["Content-Type"]
'image/jpeg'
```

On voit que l'on a affaire ici à une image, donc à du contenu binaire.
Voici comment l'on peut traiter un tel contenu :

```
>>> from PIL import Image
>>> from io import BytesIO>>> im = Image.open(BytesIO(r.content))
>>> im.save("test.jpg")
```

On utilise **`r.content`** pour extraire le contenu binaire (cela renvoie des octets).
La procédure pour utiliser cet outil est assez simple : faire la requête, vérifier le code de statut et les en-têtes, puis récupérer le résultat et l'exploiter ou traiter l'erreur selon le cas.
Bien entendu, il y a davantage dans le protocole HTTP que ce cas simple. On peut par exemple vouloir passer des paramètres d'authentification, exemples tirés de la documentation officielle (https://2.python-requests.org//en/master/).
D'abord, l'authentification de type Basic :

```
>>> r = requests.get(url, auth=('user', 'pass'))
```

Ce qui est un raccourci pour :

```
>>> from requests.auth import HTTPBasicAuth
>>>requests.get('https://api.github.com/user',
                 auth=HTTPBasicAuth('user', 'pass'))
```

L'authentification de type Digest :

```
>>> from requests.auth import HTTPDigestAuth
>>> requests.get(url, auth=HTTPDigestAuth('user', 'pass'))
```

D'autres types d'authentifications sont également décrits dans la documentation : https://requests-oauthlib.readthedocs.io/en/latest/oauth2_workflow.html
Un autre aspect important est la gestion des cookies. Parfois, ces cookies sont absolument nécessaires lorsque l'on enchaîne la navigation entre plusieurs pages. Il faut récupérer les cookies reçus dans une page pour les réutiliser dans la requête suivante :

```
>>> r = requests.get(url)
>>> cookies = r.cookies
>>> r = requests.get(new_url, cookies=cookies)
```

Ces cookies sont gérés dans un objet particulier, qui se comporte comme un dictionnaire et qui dispose d'interfaces supplémentaires pour faciliter sa manipulation :

```
>>> jar = requests.cookies.RequestsCookieJar()
>>> jar.set('my_cookie', 'value', domain='site.org', path='/path')
>>> jar["my_cookie"]
"value"
>>> r = requests.get(url, cookies=jar)
```

Outre les cookies, on peut également passer des paramètres dans l'URL (que l'on retrouve après le point d'interrogation). Pour ce faire, il faut procéder ainsi :

```
>>> payload = {'key1': 'value1', 'key2': 'value2'}
>>> r = requests.get(url, params=payload)
```

Il est aussi possible de passer plusieurs valeurs à une clé :

```
>>> payload = {'key1': ['value11', 'value12', 'value13'], 'key2':
'value2'}
>>> r = requests.get(url, params=payload)
```

Pour visualiser l'URL, il faut procéder ainsi :

```
>>> r.url
'https:localhost:8088?key1=value1&key1=k&key1=z&key2=value2'
```

On peut également vouloir passer ses propres en-têtes (la requête comme la réponse peuvent avoir des en-têtes spécifiques) :

```
>>> headers = {'user-agent': 'my-app',
               'Accept': 'text/plain,*/*;q=0.1',
               'Accept-Language': 'fr,fr-FR;q=0.8,en-US;q=0.5,en;q=0.3'
               'Connection':'keep-alive'}
>>> r = requests.get(url, headers=headers)
```

Enfin, il existe en HTTP plusieurs verbes autres que GET, dont POST ou PUTqui peuvent fournir des données :

```
>>> payload = {'key1': 'value1', 'key2': 'value2'}
>>> r = requests.post(url, data=payload)
```

Et il est bien entendu possible d'avoir à la fois des paramètres (qui iront dans l'URL) et des données :

```
>>> r = requests.post(url, data=data_payload, params=params_payload)
```

Il est également possible de fournir des fichiers :

```
>>> with open("exemple.txt") as f1, open("exemple.png", "b") as f2:
        r = requests.post(url, data=data_payload, params=params_payload,
                          files={"exemple.txt": f1,
                                 "exemple.png": f2})
```

Nous pouvons faire beaucoup plus que ce qui est exposé dans ces quelques lignes, mais on a déjà là de quoi répondre à la plupart des besoins.

© Editions ENI - All rights reserved

3. Web scrapping

Le web scrapping consiste à extraire du contenu actionnable depuis des sites web. Ceci est utile pour de nombreuses applications : référencement, comparaison entre sites, extraction d'informations...

En Python, on peut parfaitement faire ceci à l'aide de **`requests`** pour récupérer le contenu HTML puis de **`BeautifulSoup`** pour parser les données. Il existe cependant une alternative de plus haut niveau : Scrapy.

Cette technique est considérée par certains acteurs comme Google comme étant du pillage. Le web scrapping en lui-même ne pose pas à proprement parler de problèmes éthiques, ce qui compte vraiment est ce que vous ferez des données que vous récoltez.

Si vous récupérez des informations d'un site pour les dupliquer sur votre propre site, vous violez potentiellement le droit d'auteur (cela dépend de la licence associée à l'information que vous dupliquez), auquel cas cette utilisation est effectivement impropre.

Si vous récupérez des informations pour faire une veille technologique ou suivre l'évolution de prix sur des plateformes concurrentes, vous ne violez en principe aucune règle, car ces informations sont publiques. Il faut tout de même vérifier les conditions d'utilisation des sites en question.

Pour installer cette bibliothèque, procédez ainsi :

```
$ pip install scrapy
```

Reprenons ici l'exemple d'introduction de la documentation officielle qui est parfait et très complet. Pour commencer, on va présenter le site toscrape.com qui offre tout le nécessaire pour apprendre à scrapper, dont deux sites qui serviront de terrain d'entraînement : quotes.toscrape.com et books.toscrape.com.

Le premier est le plus simple à utiliser. Il est possible de cliquer sur un tag et ainsi réduire la liste des citations. Le tag *Humor* est parfait, car il propose deux pages, ce qui va nous permettre de voir plusieurs choses.

Pour commencer, le principe de scrappy est de créer un robot (**`Spider`**, qui dans le contexte informatique se traduit par robot et non araignée) pour définir une recherche particulière, puis de lancer ce robot :

```
from scrapy import Spider
```

Il faut maintenant définir le robot en surchargeant cette classe et en donnant les arguments principaux :

```
class QuotesSpider(Spider):
    name = 'quotes'
    start_urls = [
        'http://quotes.toscrape.com/tag/hope/',
        'http://quotes.toscrape.com/tag/humor/',
    ]
```

On définit ainsi les pages à parcourir et il peut y en avoir plusieurs. Il faut maintenant surcharger la fonction essentielle, celle qui va parcourir le résultat HTML et déterminer quelles sont les données à extraire. Pour cela, on va identifier des blocs grâce aux attributs HTML exprimés à la manière de CSS (d'où le nom de la méthode) :

```
    def parse(self, response):
        for quote in response.css('div.quote'):
            yield {
                'text': quote.css('span.text::text').get(),
                'author': quote.xpath('span/small/text()').get(),
            }
```

Cette partie extrait toutes les citations et, pour chacune d'elles, le texte et l'auteur. Mais comme il y a une pagination, on va chercher à identifier le bouton nous permettant de passer à la page suivante et ainsi aller aussi extraire cette page-là :

```
        next_page = response.css('li.next a::attr("href")').get()
        if next_page is not None:
            yield response.follow(next_page, self.parse)
```

Pour cela, on a refait un appel à la méthode **parse** pour la page suivante.
On ira, dans cet exemple précis, chercher les informations dans trois pages :

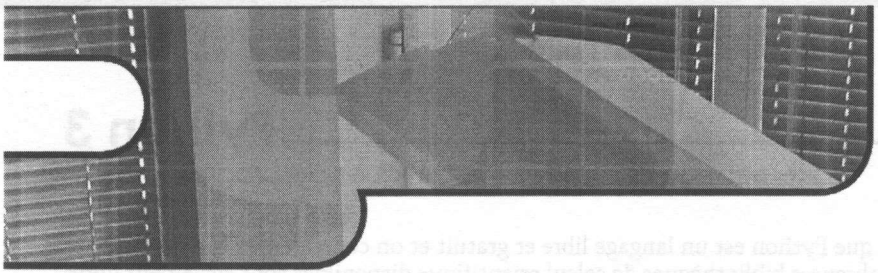
```
'http://quotes.toscrape.com/tag/hope/'
'http://quotes.toscrape.com/tag/humor/'
'http://quotes.toscrape.com/tag/humor/page/2/'
```

Une fois que le robot est défini, il faut le lancer. Pour cela, scrapy dispose d'outils de haut niveau qui vont aller chercher les robots et les exécuter dans un environnement optimisé, pour avoir le temps de traitement le plus court possible.
Voici la commande :

```
$ scrapy runspider scra.py -o result.json
```

Vous pouvez maintenant considérer le fichier obtenu pour vérifier le résultat et le comparer aux différentes pages citées plus haut que vous pourrez ouvrir dans un navigateur.
Il est à noter qu'il existe d'autres alternatives, comme Selenium (à l'origine plus dédié aux tests fonctionnels).

© Editions ENI - All rights reserved



Chapitre 2.4
Programmation scientifique

1. Calcul scientifique

1.1 Présentation

Le calcul scientifique est un domaine très particulier de l'informatique qui a ses propres exigences et nécessite des besoins très pointus. Il n'est pas pour autant uniquement destiné à des mathématiciens : http://fr.wikipedia.org/wiki/Nombre_premier_de_Mersenne

Le calcul scientifique touche des domaines aussi larges que l'ensemble des filières scientifiques (séquençage ADN, etc.), l'analyse d'image, la cartographie, l'analyse de son ou de vidéos, les mathématiques financières, le pilotage de robots, la génération de graphes, l'animation vidéo, l'analyse linguistique, la lecture de texte par un automate... qui sont tous des domaines de niche. En fait, les mathématiques sont partout et le calcul scientifique également.

N'importe quel projet informatique peut y avoir recours, y compris un projet qui n'aurait a priori rien à voir avec le domaine de la science, tel qu'un projet de gestion documentaire. En effet, les systèmes de classement et les fonctions de recherche de documents peuvent utiliser du calcul scientifique.

On peut également citer l'exemple de la retouche d'image ou du floutage vidéo en temps réel, qui sont des domaines d'application du calcul scientifique auxquels on ne pense pas forcément.

Toujours est-il que ce qui va nous intéresser dans ce chapitre, ce sont les outils spécifiquement destinés aux calculs scientifiques, quelle que soit leur finalité.

1.2 Python, une alternative libre et crédible

Pendant des années, le langage de programmation Fortran a été la référence absolue dans ce domaine et, plutôt que de réinventer la roue, Python a embarqué ses bibliothèques, ceci en combinant toutes leurs qualités et toutes celles du langage, en particulier au niveau de la syntaxe et de la lisibilité.

On rappelle que Python est un langage libre et gratuit et on constate que la quasi-totalité des très nombreuses bibliothèques de calcul scientifique disponibles sont également libres et gratuites, et c'est là un autre des points positifs puisque, contrairement à ses alternatives, vous n'avez pas besoin d'une autorisation ou d'une licence pour utiliser les programmes que vous avez écrits en Python ou même pour présenter votre travail en utilisant Python et ses bibliothèques comme support.

Un autre point très important est que ces bibliothèques sont extrêmement abouties, pointues, et n'ont rien à envier à leurs concurrentes propriétaires, fermées et payantes.

Aujourd'hui, de nombreux concours existent à destination des entreprises, doctorants ou autre sur des sujets pointus, et les deux langages utilisés pour résoudre les problématiques posées sont C++ et Python. Bien évidemment, Python se démarque par un apprentissage plus aisé, en particulier pour des non-informaticiens, mais il offre également des performances très bonnes.

Il permet un développement rapide (on peut très vite arriver à un résultat simple), incrémentiel (on peut architecturer son application de manière à y ajouter les fonctionnalités petit à petit sans devoir tout remettre en cause), permet de créer des tests et offre tout le reste de sa bibliothèque standard sans efforts supplémentaires, ce qui lui donne des armes de poids face aux outils de calcul scientifique classiques qui ont un périmètre restreint. Python apporte sa simplicité, le haut niveau, le côté dynamique, l'orientation objet, mais aussi sa portabilité (interfaçable avec C, C++ et Fortran).

Aujourd'hui, Python est enseigné dans ce domaine et est une référence pour de nombreuses entreprises.

1.3 Vue d'ensemble de quelques bibliothèques

La bibliothèque de référence généraliste Python pour le calcul scientifique est **NumPy** (Numeric Python, née en 1996, renommée en 2006), mais le nombre de ces bibliothèques est extrêmement important.

Lorsque l'on réalise du calcul scientifique, la visualisation des données est un enjeu essentiel. Il existe pour cela l'excellente bibliothèque **matplotlib**, qui permet d'afficher des restitutions 2D ou 3D, et de générer des documents (EPS, PDF...).

SciPy est un projet destiné à regrouper, unifier, standardiser, homogénéiser ces bibliothèques selon les principes qui régissent Python et qui sont décrits dans la PEP-20.

Il utilise **NumPy** et **matplotlib**, qui sont deux des piliers du calcul scientifique en Python, et cherche à reproduire un environnement de travail conforme aux concurrents propriétaires de Python de manière à faciliter la transition et la migration de l'existant.

Il contient des modules d'algèbre linéaire, de statistiques, mais également de traitement du signal, de traitement d'image qui font référence.

Aujourd'hui, toutes les fonctionnalités essentielles sont regroupées au sein d'une seule bibliothèque, **NumPy**.

```
>>> from numpy import *
```

Nous n'allons pas particulièrement nous attarder sur des concepts mathématiques comme la transformée de Fourier, par exemple, mais sur les outils de base : le tableau multidimensionnel et la matrice, ainsi que sur la génération de graphiques.

© Editions ENI - All rights reserved

2. Tableaux multidimensionnels

2.1 Création

Un tableau NumPy peut être créé à partir d'une liste Python :

```
>>> a = array([2, 3, 4])
```

Remarque

Attention : il ne faut pas confondre ce tableau à celui du module ***array****.*

Lorsque l'on affiche ce tableau, il se présente d'ailleurs sous la forme d'une liste :

```
>>> print(a)
[2, 3, 4]
```

Mais on peut visualiser le type des éléments qu'il contient :

```
>>> a.dtype
dtype('int64')
```

Ce résultat peut changer en fonction de votre plateforme (et du compilateur C qui se trouve derrière votre Python), mais vous aurez un type entier. Si un seul nombre n'est pas entier, vous aurez un type réel :

```
>>> b = array([1.2, 3.5, 5.1])
>>> b.dtype
dtype('float64')
```

On peut aussi créer très rapidement des tableaux de plusieurs dimensions à partir de listes de listes ou de listes de n-uplets :

```
>>> array( [ (1.5,2,3), (4,5,6) ] )
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
>>> array( [ ((1.5,2,3), (4,5,6)), ((1.5,2,3), (4,5,6)) ] )
array([[[ 1.5,  2. ,  3. ],
        [ 4. ,  5. ,  6. ]],

       [[ 1.5,  2. ,  3. ],
        [ 4. ,  5. ,  6. ]]])
```

Lors de l'affichage, la seconde dimension apparaît grâce au saut de ligne, la troisième grâce à deux sauts, ce qui laisse apparaître une ligne blanche.

Comme l'outil est très souple, on peut aussi tout simplement générer une liste à l'aide d'un générateur, similaire à **range** :

```
>>> arange(15)
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

Il peut également prendre un pas :

```
>>> arange( 10, 30, 5 )
array([10, 15, 20, 25])
```

Mais peut aussi fonctionner avec des nombres réels :

```
>>> arange( 0, 2, 0.3 )
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8])
```

Enfin, une fois que l'on a généré une liste de nombres, on peut la remodeler à volonté, nous permettant donc de créer des tableaux à plusieurs dimensions :

```
>>> arange(15).reshape(3, 5)
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> arange(16).reshape(2, 4, 2)
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5],
        [ 6,  7]],

       [[ 8,  9],
        [10, 11],
        [12, 13],
        [14, 15]]])
```

2.2 Déterminer la composition d'un tableau

Maintenant, imaginons que vous avez entre les mains un tableau que vous ne connaissez pas a priori. Pour cet exercice, ce sera celui-ci :

```
>>> a = arange(15).reshape(3, 5)
```

Vous pouvez vérifier qu'il s'agit bien d'un tableau multidimensionnel :

```
>>> type(a)
numpy.ndarray
```

Vous pouvez connaître le nombre de dimensions :

```
>>> a.ndim
2
```

Et sa forme, c'est-à-dire la longueur de chaque dimension :

```
>>> a.shape
(3, 5)
```

Vous pouvez également connaître le type utilisé pour stocker les nombres :

```
>>> a.dtype.name
'int64'
```

La taille totale en mémoire d'un de ces nombres :

```
>>> a.itemsize
8
```

© Editions ENI - All rights reserved

Et la taille totale du tableau (soit la multiplication entre les longueurs des dimensions) :

```
>>> a.size
15
```

2.3 Générateurs de tableaux

Nous avons déjà vu la fonction **arange** qui nous permet de générer un tableau en le remplissant de nombres séparés entre eux par un pas.
Dans la plupart des cas, nous avons simplement besoin d'un tableau vide ; et vide, en mathématiques, signifie soit que des 0 (le neutre de l'addition) :

```
>>> zeros( (3,4) )
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

soit que des 1 (le neutre de la multiplication) :

```
>>> ones( (2,3,4), dtype=int16 )
array([[[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]],

       [[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]]], dtype=int16)
```

Au passage, on a vu que l'on peut ici choisir le type à utiliser pour les nombres, on peut donc créer un tableau de 1 ou de 0 réels sans trop d'efforts.
On peut aussi générer un tableau vide, celui-ci se remplira alors de nombres extrêmement proches de 0.

```
>>> empty( (2,3) )
array([[  6.91200909e-310,   3.62814222e-316,   3.61014518e-316],
       [  3.62623315e-316,   3.61670203e-316,   3.63022125e-316]])
```

Enfin, nous faisons ici une introduction à un outil très utile lorsqu'il s'agit de dessiner des graphiques : l'espace linéaire.
La fonction qui suit va créer une série de points compris entre un minimum et un maximum, et on va indiquer combien de points on souhaite avoir :

```
>>> linspace( 0, 2, 9 )
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ,  1.25,  1.5 ,  1.75,  2.  ])
```

Ceci pourrait aussi être réalisé grâce à **arange**, mais cette écriture est plus directe lorsque l'on a à l'esprit le graphique à réaliser et le résultat attendu. Il suffit de changer le dernier paramètre pour passer d'un graphique peu précis à quelque chose de plus vivant. Personnellement, nous recommandons d'utiliser des nombres impairs pour avoir un nombre d'intervalles pair, ce qui, en général, donne des résultats visuellement plus réussis.

Une fois que l'on a son espace linéaire, voici comment écrire une fonction :

```
>>> def f(x,y):
...     return 10*x+y
```

On aurait pu également utiliser une fonction lambda nommée. Toujours est-il que l'on peut obtenir le résultat de cette fonction ainsi :

```
>>> b = fromfunction(f,(5,4),dtype=int)
```

On a ici le résultat de l'application de cette fonction, **x** variant de **0** à **5** exclus (donc **4**) et **y** variant de **0** à **4** exclus (donc **3**). Notons que **x** est la première variable, donc varie de ligne en ligne, et **y** est la seconde, donc varie de colonne en colonne :

```
>>> b
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
```

2.4 Opérations basiques

Créons deux tableaux de départ, de dimension et de forme identique :

```
>>> a = array( [20,30,40,50] )
>>> b = arange( 4 )
```

Il est possible d'effectuer les opérations d'addition, de soustraction, de multiplication et de division entre ces tableaux. Que cela signifie-t-il ? Soustraire deux tableaux, par exemple :

```
>>> a-b
array([20, 29, 38, 47])
```

Le résultat est un tableau de dimension identique au tableau de départ et dans lequel le résultat est la soustraction des éléments à la même position.
Il en est de même pour la multiplication :

```
>>> a*b
array([  0,  30,  80, 150])
```

Attention, on rappelle que l'on est là dans un tableau multidimensionnel et non pas dans une matrice. On a également une notion de produit scalaire :

```
>>> dot(a, b)
260
```

Pour des tableaux unidimensionnels, le résultat est la somme des multiplications deux à deux :
Attention, dans le cas de tableaux multidimensionnels, le résultat est différent :

```
>>> dot(arange(4).reshape(2, 2), arange(4).reshape(2, 2))
array([[ 2,  3],
       [ 6, 11]])
```

© Editions ENI - All rights reserved

Et voici ce qu'il se passe lorsque l'on multiplie un tableau horizontal par un autre vertical :

```
>>> dot(arange(4).reshape(4, 1), arange(4).reshape(1, 4))
array([[0, 0, 0, 0],
       [0, 1, 2, 3],
       [0, 2, 4, 6],
       [0, 3, 6, 9]])
```

Et voici le dernier exemple avec des tableaux aux dimensions transposées :

```
>>> dot(arange(6).reshape(2, 3), arange(6).reshape(3, 2))
array([[10, 13],
       [28, 40]])
```

Enfin, on peut ajouter, soustraire, multiplier ou diviser par un nombre :

```
>>> a*3
array([ 60,  90, 120, 150])
>>> b+6
array([6, 7, 8, 9])
```

Le nombre sera alors ajouté, soustrait, multiplié à tous les éléments du tableau ou sera le diviseur de ces éléments.
On peut également calculer la somme des éléments d'un tableau, quelle que soit sa dimension :

```
>>> b.sum()
6
>>> arange(15).reshape(3, 5).sum()
105
```

On peut récupérer le minimum et le maximum :

```
>>> b.min()
0
>>> b.max()
3
```

On peut également faire ce calcul selon une dimension particulière. Prenons un nouveau tableau :

```
>>> c = arange(12).reshape(3,4)
>>> c
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Voici la somme selon la première dimension, c'est-à-dire, dans ce cas, verticale :

```
>>> c.sum(axis=0)
array([12, 15, 18, 21])
```

Et la somme selon la seconde dimension :

```
>>> c.sum(axis=1)
array([ 6, 22, 38])
```

On peut également avoir le minimum ou le maximum par dimension :

```
>>> c.min(axis=1)
array([0, 4, 8])
```

Enfin, dernier élément important : on peut toujours accéder à l'aide à tout moment d'une fonction :

```
>>> help(c.min)
Help on built-in function min:

min(...)
    a.min(axis=None, out=None)

    Return the minimum along a given axis.

    Refer to `numpy.amin` for full documentation.

    See Also
    --------
    numpy.amin : equivalent function
```

Terminons en signalant qu'au-delà des opérations communes, on peut aussi appliquer des fonctions à l'ensemble des éléments d'un tableau :

```
>>> exp(a)
array([  4.85165195e+08,   1.06864746e+13,   2.35385267e+17,
         5.18470553e+21])
>>> sqrt(b)
array([[ 0.        ,  1.        ,  1.41421356,  1.73205081],
       [ 2.        ,  2.23606798,  2.44948974,  2.64575131],
       [ 2.82842712,  3.        ,  3.16227766,  3.31662479]])
```

2.5 Opérateur crochet

L'opérateur crochet est un bijou en Python. Il l'est aussi en Python scientifique. Commençons par définir un nouveau tableau :

```
>>> d = arange(8).reshape(2, 4)
>>> d
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

On peut accéder directement à un élément à partir de ses indices selon les différentes dimensions :

```
>>> d[1,3]
7
```

Ce qui est plus élégant que d'utiliser plusieurs opérateurs crochets. Et on peut utiliser aussi des indices négatifs :

```
>>> d[-1,-1]
7
```

© Editions ENI - All rights reserved

Et un pas :

```
>>> d[0:5, 1]
array([1, 5])
>>> d[ : ,1]
array([1, 5])
>>> d[1:3, : ]
array([[4, 5, 6, 7]])
```

On peut également itérer sur un tableau très simplement :

```
>>> for r in b:
...     print(r)
...
[0 1 2 3]
[4 5 6 7]
```

On obtient l'élément de dimension n - 1. On peut tout aussi facilement itérer sur les éléments du tableau :

```
>>> for e in d.flat:
...     print(e)
...
0
1
2
3
4
5
6
7
```

Pour obtenir une version aplatie du tableau dans une nouvelle variable, on peut utiliser ceci :

```
>>> d.ravel()
array([0, 1, 2, 3, 4, 5, 6, 7])
>>> d
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

Vous noterez que le tableau de départ n'est pas modifié. Au contraire, pour modifier le tableau courant, il suffit d'utiliser **reshape** ou de passer la nouvelle forme ainsi :

```
>>> d.shape = (4, 2)
>>> d
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

Il est également possible de transposer un tableau :

```
>>> d.transpose()
array([[0, 2, 4, 6],
       [1, 3, 5, 7]])
```

Voici maintenant comment générer des nombres au hasard (et les arrondir au nombre inférieur) :

```
>>> e = floor(10*random.random((2,2)))
>>> e
array([[ 0.,  5.],
       [ 7.,  6.]])
```

Créons avec la même méthode un autre tableau :

```
>>> f = floor(10*random.random((2,2)))
>>> f
array([[ 1.,  2.],
       [ 0.,  8.]])
```

Voici comment empiler ces tableaux :

```
>>> vstack((e, f))
array([[ 0.,  5.],
       [ 7.,  6.],
       [ 1.,  2.],
       [ 0.,  8.]])
```

On peut aussi les empiler par la droite :

```
>>> hstack((e, f))
array([[ 0.,  5.,  1.,  2.],
       [ 7.,  6.,  0.,  8.]])
```

Nous avons donc fait ici un tour relativement complet des manières simples de créer et de modifier un tableau. Tout le sel du calcul scientifique va justement consister à faire des calculs à partir de tableaux de nombres que vous aurez générés ainsi ou que vous aurez échantillonnés et que vous pourrez retravailler.

3. Matrices

Si les tableaux sont des objets malléables représentant un ensemble de données quelconque, les matrices sont des objets mathématiques aux règles strictes.
Il y a cependant quelques similarités entre les matrices et les tableaux. Toujours est-il que NumPy permet de les manipuler très simplement.
Voici comment créer une matrice, à partir d'une simple chaîne de caractères :

```
>>> A = matrix('1.0 2.0; 3.0 4.0')
```

Notez que l'espace sert à séparer les éléments d'une même ligne et que le point-virgule sert à séparer les lignes entre elles.

© Editions ENI - All rights reserved

Lorsque l'on affiche une matrice, cela se présente toujours sous la forme d'une liste de listes, mais cette fois-ci on a **matrix** au lieu de **array** :

```
>>> A
matrix([[ 1.,  2.],
        [ 3.,  4.]])
```

On peut calculer la transposée d'une matrice ainsi :

```
>>> A.T
matrix([[ 1.,  3.],
        [ 2.,  4.]])
```

Voici une matrice sur une ligne :

```
>>> X = matrix('5.0 7.0')
>>> X
matrix([[ 5.,  7.]])
```

Sa transposée sera sur une seule colonne :

```
Y = X.T
>>> Y
matrix([[ 5.],
        [ 7.]])
```

On peut multiplier une matrice carrée par une matrice colonne de la même taille :

```
>>> A*Y
matrix([[ 19.],
        [ 43.]])
```

On peut aussi calculer l'inverse d'une matrice carrée :

```
>>> A.I
matrix([[-2. ,  1. ],
        [ 1.5, -0.5]])
```

Et on peut résoudre l'équation matricielle :

```
>>> solve(A, Y)
matrix([[-3.],
        [ 4.]])
```

On a ici présenté les éléments syntaxiques des opérations essentielles pour les matrices.

4. Génération de graphiques

Python est un langage qui a de très nombreuses cordes à son arc. Il peut totalement être utilisé de manière impérative ou de manière objet, sans aucun problème.
En ce qui concerne la génération de graphiques, on peut donc soit faire appel à une syntaxe pythonique qui va utiliser l'objet, soit préférer une syntaxe à la MATLAB, ce qui est pratique pour les connaisseurs de ce langage, puisque cela leur évite d'avoir à réapprendre tout et d'avoir à maîtriser les concepts objet.
Pour visualiser un graphique, vous aurez besoin d'utiliser une console IPython ou, encore mieux, le IPython Notebook.

Pour ce dernier, la commande recommandée pour le lancer est :

```
$ ipython3 notebook --pylab inline
```

4.1 Syntaxe MATLAB

Pour dessiner un graphique, il suffit de procéder par étapes. Tout d'abord, on crée un espace linéaire :

```
>>> x = linspace(-5, 5, 11)
```

Ensuite, on crée la fonction à représenter :

```
>>> y = x ** 2
```

Puis, on crée notre espace de représentation :

```
>>> figure()
```

Suivi du graphique (on veut représenter y par rapport à x et on veut une ligne rouge) :

```
>>> plot(x, y, 'r')
```

On peut ensuite donner un nom aux axes :

```
>>> xlabel('abscisse')
>>> ylabel('ordonnee')
```

Puis donner un titre au graphique lui-même :

```
>>> title('fonction carree')
```

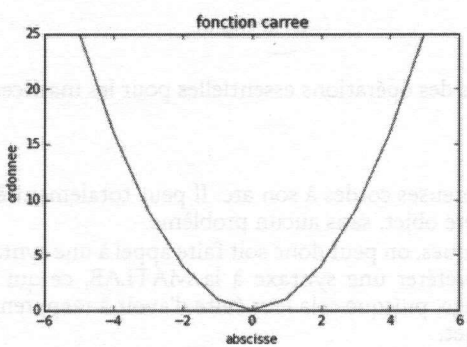
Ensuite, vous pouvez afficher votre graphique :

```
>>> show()
```

Notons que la courbe sera d'autant plus précise qu'il y aura de points. Ceci se fait en changeant l'espace linéaire de départ (et en répétant toutes les étapes) :

```
>>> x = linspace(-5, 5, 101)
>>> x = linspace(-5, 5, 1001)
```

Voici le résultat :



© Editions ENI - All rights reserved

4.2 Syntaxe objet

Nous pouvons réaliser exactement le même type d'opération à l'aide d'une syntaxe objet :

```
x = linspace(-5, 5, 11)
y = x ** 2
fig = plt.figure()
axes = fig.add_axes([0, 0, 1, 1])
axes.plot(x, y, 'r')
axes.set_xlabel('abscisse')
axes.set_ylabel('ordonnee')
axes.set_title('fonction carree')
```

Dans IPython ou IPython Notebook, ce code suffira à afficher le graphique. Vous pouvez également l'enregistrer dans un fichier :

```
fig.savefig("courbe.png")
```

Le format PNG est un excellent format d'image pour ce type de graphiques. Il n'est cependant pas possible de l'agrandir sans faire apparaître de pixels.
Pour des besoins précis, vous pourrez toujours préciser une résolution :

```
fig.savefig("courbe2.png", dpi=300)
```

Mais si vous souhaitez utiliser votre graphique quelle que soit l'échelle, il vaut mieux privilégier le format vectoriel :

```
fig.savefig("courbe.svg")
```

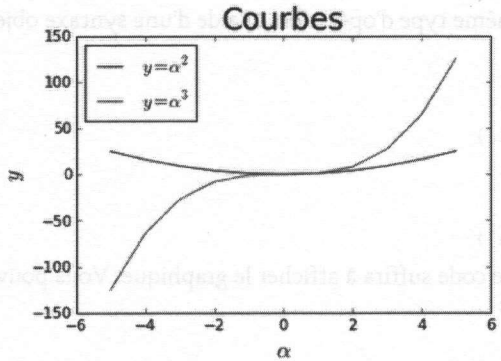
4.3 Mise en forme

Les options pour mettre en forme les graphiques sont illimitées :

```
fig, ax = subplots()

ax.plot(x, x**2, label=r"$y = \alpha^2$")
ax.plot(x, x**3, label=r"$y = \alpha^3$")
ax.set_xlabel(r'$\alpha$', fontsize=18)
ax.set_ylabel(r'$y$', fontsize=18)
ax.set_title('Courbes', fontsize=24)
ax.legend(loc=2);
```

Ici, on crée un graphique affichant deux courbes bleue et verte, en précisant la taille à utiliser pour le label et en utilisant LaTeX pour la légende.

Voici le résultat :



Il est à noter qu'il est possible de changer le style par défaut :

```
matplotlib.rcParams.update({'font.size': 18,
                            'font.family': 'serif'})
```

Pour revenir aux paramètres par défaut :

```
matplotlib.rcParams.update({'font.size': 12,
                            'font.family': 'sans'})
```

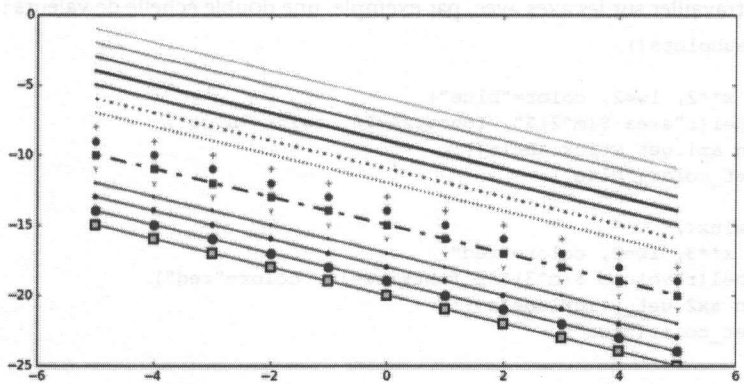
L'exemple suivant, issu de la documentation officielle, vous permettra de voir comment utiliser différentes couleurs et types de lignes et comment marquer les points ; il vous suffit de comparer le code au résultat.

```
fig, ax = subplots(figsize=(12,6))

ax.plot(x, -x-6, color="blue", linewidth=0.25)
ax.plot(x, -x-7, color="blue", linewidth=0.50)
ax.plot(x, -x-8, color="blue", linewidth=1.00)
ax.plot(x, -x-9, color="blue", linewidth=2.00)

# possible linestype options '-', '-', '-.', ':', 'steps'
ax.plot(x, -x-10, color="red", lw=2, linestyle='-')
ax.plot(x, -x-11, color="red", lw=2, ls='-.')
ax.plot(x, -x-12, color="red", lw=2, ls=':')

# custom dash
line, = ax.plot(x, -x-15, color="black", lw=1.50)
line.set_dashes([5, 10, 15, 10]) # format: line length, space length, ...

# possible marker symbols: marker = '+', 'o', '*', 's', ',', '.',
# '1', '2', '3', '4', ...
ax.plot(x, -x-13, color="green", lw=2, ls='*', marker='+')
ax.plot(x, -x-14, color="green", lw=2, ls='*', marker='o')
ax.plot(x, -x-15, color="green", lw=2, ls='*', marker='s')
```

© Editions ENI - All rights reserved

```
ax.plot(x, -x-16, color="green", lw=2, ls='*', marker='1')

# marker size and color
ax.plot(x, -x-17, color="purple", lw=1, ls='-', marker='o', markersize=2)
ax.plot(x, -x-18, color="purple", lw=1, ls='-', marker='o', markersize=4)
ax.plot(x, -x-19, color="purple", lw=1, ls='-', marker='o', markersize=8,
        markerfacecolor="red")
ax.plot(x, -x-20, color="purple", lw=1, ls='-', marker='s', markersize=8,
        markerfacecolor="yellow", markeredgewidth=2,
markeredgecolor="blue")
```

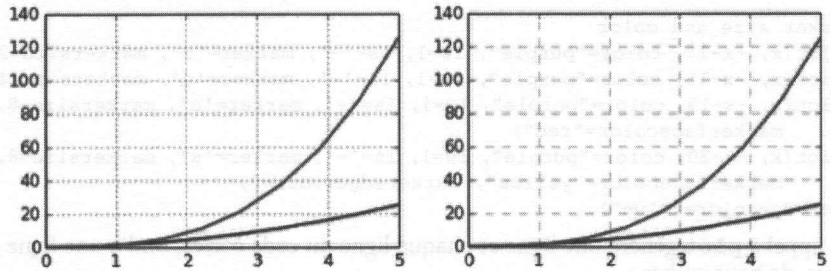
Chaque appel à **plot** génère une ligne et chaque ligne du code correspond à une ligne du graphique, de haut en bas :



On peut également faire apparaître la grille du fond :

```
fig, axes = subplots(1, 2, figsize=(10,3))

# default grid appearance
axes[0].plot(x, x**2, x, x**3, lw=2)
axes[0].grid(True)

# custom grid appearance
axes[1].plot(x, x**2, x, x**3, lw=2)
axes[1].grid(color='b', alpha=0.5, linestyle='dashed', linewidth=0.5)
```

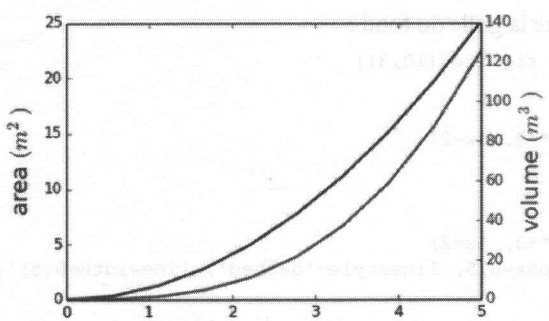
Ce qui produit le résultat suivant :



Voici comment travailler sur les axes avec, par exemple, une double échelle de valeurs :

```
fig, ax1 = subplots()

ax1.plot(x, x**2, lw=2, color="blue")
ax1.set_ylabel(r"area $(m^2)$", fontsize=18, color="blue")
for label in ax1.get_yticklabels():
    label.set_color("blue")

ax2 = ax1.twinx()
ax2.plot(x, x**3, lw=2, color="red")
ax2.set_ylabel(r"volume $(m^3)$", fontsize=18, color="red")
for label in ax2.get_yticklabels():
    label.set_color("red")
```

Soit :



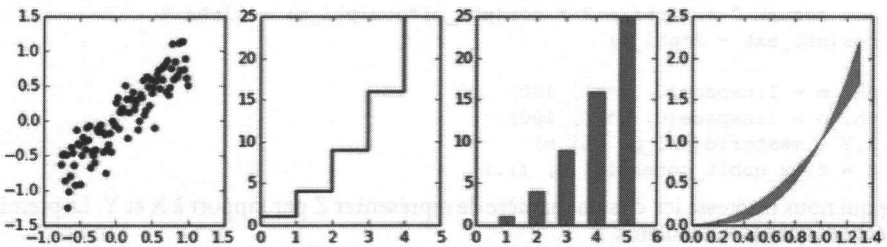
Ou travailler sur la représentation graphique :

```
n = array([0,1,2,3,4,5])

fig, axes = subplots(1, 4, figsize=(12,3))

axes[0].scatter(xx, xx + 0.25*randn(len(xx)))
```

© Editions ENI - All rights reserved

```
axes[1].step(n, n**2, lw=2)

axes[2].bar(n, n**2, align="center", width=0.5, alpha=0.5)

x2 = linspace(0, 1.3, 16)
axes[3].fill_between(x2, x2**2, x2**3, color="green", alpha=0.5)
```

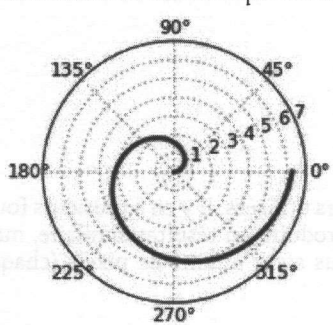
Ce qui donne :



Notez que l'on a mis deux graphiques côte à côte en créant notre objet **subplot** en mode impératif (à la MATLAB), ligne 2 (**subplots** avec 1 ligne et 4 colonnes et la taille de l'image est de 12 par 3). Cela crée automatiquement quatre axes sur lesquels on peut travailler, chacun étant un des graphiques.
On peut également utiliser la projection polaire :

```
fig = plt.figure()
ax = fig.add_axes([0.0, 0.0, .6, .6], polar=True)
t = linspace(0, 2 * pi, 100)
ax.plot(t, t, color='blue', lw=3)
```

Le résultat de cet exemple est une spirale :



4.4 Graphiques 3D

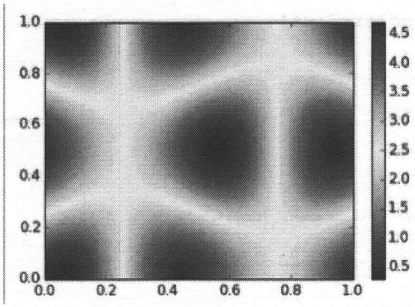
On peut également représenter des courbes 3D, c'est-à-dire une fonction qui prend deux paramètres. Voici un exemple classique, également issu de la documentation officielle :

```
alpha = 0.5
phi_ext = pi/2

def flux_qubit_potential(phi_m, phi_p):
    return 2 + alpha - 2 * cos(phi_p)*cos(phi_m) - alpha *
cos(phi_ext - 2*phi_p)

phi_m = linspace(0, 2*pi, 100)
phi_p = linspace(0, 2*pi, 100)
X,Y = meshgrid(phi_p, phi_m)
Z = flux_qubit_potential(X, Y).T
```

Ce qui nous intéresse ici, c'est la manière de représenter Z par rapport à X et Y. La première méthode utilise les couleurs :

```
fig, ax = subplots()

im = ax.pcolor(X/(2*pi), Y/(2*pi), Z, cmap=cm.RdBu,
vmin=abs(Z).min(), vmax=abs(Z).max())
fig.colorbar(im)
```

Ce qui nous donne un résultat tel que celui-ci :

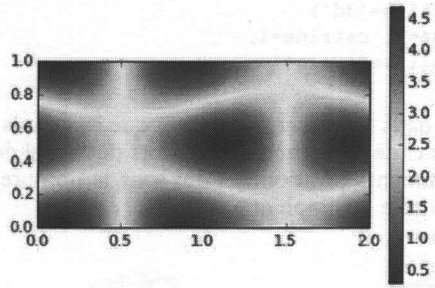


Le paramètre **cmap** permet de définir la carte des couleurs utilisées. Il y en a plusieurs fournies d'office. Voici une autre méthode qui permet de produire un résultat similaire, mais avec une interpolation bilinéaire permettant de ne plus avoir autant de pixels (chaque pixel correspond en fait à une des valeurs calculées) :

```
fig, ax = subplots()

im = imshow(Z, cmap=cm.RdBu, vmin=abs(Z).min(), vmax=abs(Z).max(),
extent=[0, 2, 0, 1])
im.set_interpolation('bilinear')

fig.colorbar(im)
```

© Editions ENI - All rights reserved

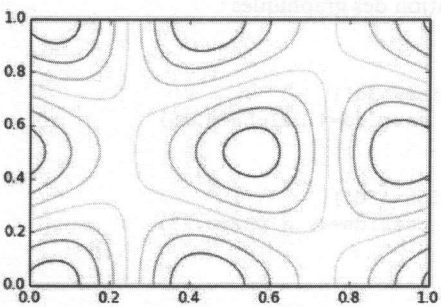
On notera que l'on a étiré le résultat initial (à l'aide du paramètre **extent**) :



Voici maintenant comment obtenir en lieu et place de ces couleurs (qui ne sont pas forcément faciles à lire) des lignes de niveau :

```
ig, ax = subplots()

contour(Z, cmap=cm.RdBu, vmin=abs(Z).min(), vmax=abs(Z).max(),
extent=[0, 1, 0, 1])
```

Ce qui donne un résultat plus lisible :



Au-delà de ces méthodes, on a la possibilité de dessiner réellement notre courbe en 3D. Pour ce faire, il faut déclarer notre axe avec un argument **projection** particulier :

```
fig = plt.figure(figsize=(14,6))

ax = fig.add_subplot(1, 3, 1, projection='3d')
```

Le code suivant va afficher un graphique sous la forme d'une nappe :

```
im = ax.plot_surface(X, Y, Z, rstride=4, cstride=4, linewidth=0)
```

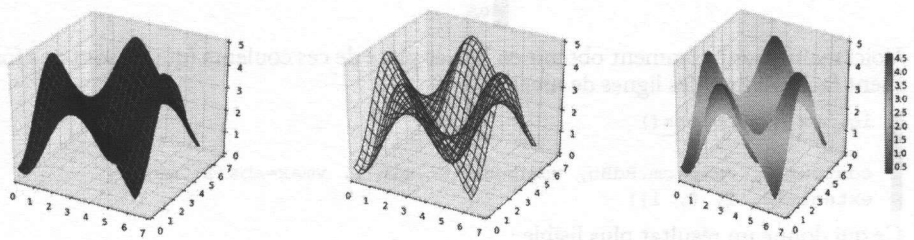
Cela générera un maillage dont les paramètres **rstride** et **cstride** servent à définir la densité. On peut d'ailleurs préférer avoir le graphique sous forme d'une grille (ou de mailles) seulement :

```
ax = fig.add_subplot(1, 3, 2, projection='3d')
im = ax.plot_wireframe(X, Y, Z, rstride=4, cstride=4)
```

Le code suivant va également utiliser le système des couleurs :

```
ax = fig.add_subplot(1, 3, 3, projection='3d')
im = ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
cmap=cm.coolwarm, linewidth=0, antialiased=False)
fig.colorbar(im, shrink=0.5)
```

Notez que l'on a changé de carte de couleurs dans le dernier exemple et que l'on a mis les graphiques en ligne par l'utilisation cette fois-ci de la méthode objet : lors de l'appel de la méthode **add_subplot**, on passe trois arguments positionnels qui sont le nombre de lignes, le nombre de colonnes et le numéro d'ordre du graphique.
Le résultat de ces trois graphiques est le suivant :



Notons qu'il est possible de changer l'orientation des graphiques :

```
fig = plt.figure(figsize=(12,6))

ax = fig.add_subplot(1,2,1, projection='3d')
ax.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.25)
ax.view_init(30, 45)

ax = fig.add_subplot(1,2,2, projection='3d')
ax.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.25)

ax.view_init(70, 30)

fig.tight_layout()
```

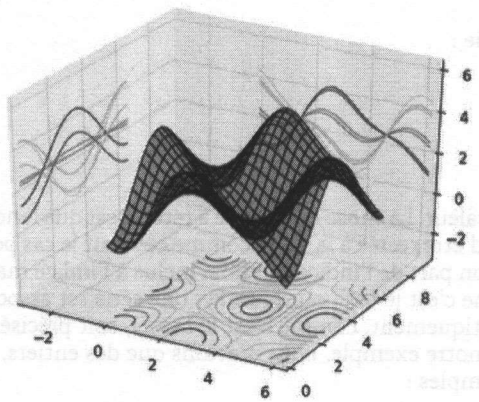
Dans cet exemple, notons l'utilisation du paramètre **alpha** qui sert à déterminer le niveau de transparence de la nappe. C'est la méthode **view_init** qui permet de choisir l'orientation du graphique.
Pour conclure, sachez que l'on peut dessiner à la fois la courbe 3D, mais également sa projection selon les trois axes :

```
fig = plt.figure(figsize=(8,6))

ax = fig.add_subplot(1,1,1, projection='3d')

ax.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.25)
cset = ax.contour(X, Y, Z, zdir='z', offset=-pi, cmap=cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='x', offset=-pi, cmap=cm.coolwarm)
```

© Editions ENI - All rights reserved

```
cset = ax.contour(X, Y, Z, zdir='y', offset=3*pi, cmap=cm.coolwarm)

ax.set_xlim3d(-pi, 2*pi)
ax.set_ylim3d(0, 3*pi)
ax.set_zlim3d(-pi, 2*pi)
```

Ce qui donne le résultat suivant :



5. Introduction à Pandas

5.1 Présentation

Pandas est une bibliothèque Python spécialisée dans le traitement, la manipulation et l'analyse de données.

Ces données sont organisées sous la forme d'un dataframe, lequel est un tableau organisé, à la manière d'un fichier CSV. Les colonnes d'un tel tableau correspondent à des variables et les lignes à des individus. Lorsque les données sont organisées sous une seule colonne, on peut utiliser la notion de série.

Pour installer Pandas, il suffit de faire :

```
$ pip install pandas
```

5.2 Séries

Une série est un dataframe n'ayant qu'une seule colonne, comparable à une liste Python. Il se crée d'ailleurs à partir d'une telle liste :

```
>>> from pandas import Series
>>> s = Series([1, 1, 2, 3, 5, 8, 13])
```

Chaque série est associée à un index qui, par défaut, est construit comme celui des listes :

```
>>> s.index
RangeIndex(start=0, stop=7, step=1)
```

On peut ainsi adresser rapidement un élément de la série à partir de son index :

```
>>> s[1]
1
>>> s[4]
5
>>> s[9]
KeyError: 9
```

On peut également extraire une sous-série :

```
>>> s[2:5]
2    2
3    3
4    5
dtype: int64
```

On voit ici à gauche l'index et à droite la valeur. La chose essentielle à retenir est que l'index associé à la valeur reste identique au lieu d'être recréé à la volée comme ce serait le cas pour une liste. Notons que, comme toujours, on part de l'indice minimal inclus à l'indice maximal exclu. Notons également que, comme c'est le cas pour NumPy, une série est associée à un type qui est soit déterminé automatiquement, comme c'est le cas ici, soit précisé au moment de la création de la série. Dans notre exemple, nous n'avions que des entiers, on a donc un type entier. Voici d'autres exemples :

```
>>> Series([1, 1, 2, 3, 5, 8, 13], dtype=float).dtype
dtype('float64')
>>> Series([1, 1., 2, 3, 5, 8, 13]).dtype
dtype('float64')
```

Dans le premier cas, le typage est explicite, dans le second cas il est implicite et dû au fait qu'il y a un **1.** qui est la représentation d'un nombre flottant.
Enfin, une des fonctionnalités essentielles de la série est qu'il est possible d'extraire des sous-séries non pas en travaillant sur l'index, mais en travaillant sur la valeur :

```
>>> s[s % 2 == 0]
2    2
5    8
dtype: int64
```

Cet exemple n'a retenu que les valeurs paires. Voici une mise en évidence de la particularité des index :

```
>>> ss = s[s % 2 == 0]
>>> ss[2]
2
>>> ss[0]
KeyError: 0
```

On peut aussi utiliser ceci :

```
>>> s.iat[5]
8
>>> s.iloc[5]
```

© Editions ENI - All rights reserved

```
8
>>> s.get(5)
8
>>> ss.iat[0]
2
>>> ss.iloc[0]
2
>>> ss.get(0)
```

Les méthodes **iat** et **iloc** vont chercher l'index reconstitué alors que le crochet et la méthode **get** vont utiliser les index initiaux, la première renvoyant une exception en cas de non existence et la seconde renvoyant **None**.
Pour terminer sur le sujet des indices, il est important de savoir qu'ils peuvent être donnés au moment de la construction de la série :

```
>>> s = Series([1, 1, 2, 3, 5, 8, 13], index=['a', 'b', 'c', 'd', 'e', 'f', 'g'])
```

Il est aussi possible de poser les indices après coup :

```
>>> s.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g'])
```

Remarque

Attention : s'il y a moins d'indices que de valeurs, alors les valeurs sans indice seront supprimées ; au contraire, s'il y a plus d'indices, ces derniers seront associés à NaN, qui est un moyen de dire qu'il n'y a pas de valeur associée à l'indice.

Voici comment ces indices peuvent être utilisés :

```
>>> s["a"]
1
>>> s["g"]
13
>>> s.iat["a"]
ValueError: iAt based indexing can only have integer indexers
>>> s.iloc["a"]
TypeError: Cannot index by location index with a non-integer key
>>> s[0]
1
>>> s[5]
8
```

Par rapport à des listes Python, il y a des différences notables à connaître. Par exemple :

```
>>> 13 in s
False
>>> 'a' in s
True
>>> 13 in s.values
True
```

On peut utiliser ces index pour classer les valeurs, tout comme on peut classer les valeurs entre elles et visualiser l'ordre des index par la suite. Pour cela, il faut utiliser :

```
>>> s.sort_index()
>>> s.sort_values()
```

Au passage, on peut aussi transformer les valeurs ainsi :

```
>>> s.apply(lambda x: x**2)
a      1
b      1
c      4
d      9
e     25
f     64
g    169
dtype: int64
```

La gestion de ces index est également très importante lorsque l'on gère plusieurs séries :

```
>>> s1, s2 = Series([1, 3, 5]), Series([4, 2, 0])
>>> s1 + s2
0    5
1    5
2    5
dtype: int64
```

Dans le cas précédent, on additionne une à une chaque valeur, en se reposant sur les index par défaut. Les séries ayant la même longueur, le traitement s'effectue sans surprise. Voici maintenant un exemple plus complexe :

```
>>> s1, s2 = Series([1, 3, 5], index=['a', 'b', 'c']), Series([4, 2, 0],
index=['c', 'd', 'b'])
>>> s1 + s2
a    NaN
b    3.0
c    9.0
d    NaN
dtype: float64
```

Ici, seuls les éléments dont les indices sont présents dans les deux listes ont une valeur. Les autres sont à NaN. Il est possible de supprimer les valeurs NaN et leurs indices correspondants :

```
>>> (s1 + s2).dropna()
b    3.0
c    9.0
dtype: float64
```

Il est aussi possible de récupérer toutes les valeurs uniques :

```
array([ 1,  2,  3,  5,  8, 13])
```

Ou encore de savoir combien d'occurrences on a pour chaque valeur :

```
>>> s.value_counts()
1     2
13    1
5     1
3     1
```

© Editions ENI - All rights reserved

```
2     1
8     1
dtype: int64
```

Pour terminer sur les séries, voici quelques éléments statistiques :

```
>>> s.describe()
count     7.000000
mean      4.714286
std       4.423961
min       1.000000
25%       1.500000
50%       3.000000
75%       6.500000
max      13.000000
dtype: float64
>>> s.count()
7
>>> s.mean()
4.714285714285714
>>> s.std()
4.423960733486292
>>> s.min()
1
>>> s.max()
13
>>> s.median()
3.0
>>> s.median(25)
>>> s.sum()
33
>>> s.mad()
3.387755102040816
```

Pour en savoir plus, n'oubliez pas qu'il existe avant tout cette simple possibilité :

```
>>> help(s.mad)
```

Cet outil est puissant, mais surtout particulièrement utile et adapté au traitement de grandes quantités de données.

5.3 Dataframes

Un dataframe est un objet qui combine les possibilités d'une liste de listes et d'une liste de dictionnaires. Chaque élément de la liste (ou ligne) est un individu et chaque valeur de l'individu dispose à la fois d'une relation d'ordre (numéro de la colonne) et d'une étiquette (titre de la colonne).

Il est possible de transformer une série en dataframe ainsi :

```
>>> d = s.to_frame()
```

Voici une construction depuis une liste de dictionnaires :

```
>>> d= DataFrame([{"u": 1, "d": 2, "t": 3}, {"u": 10, "d": 20, "t": 30}])
```

Puis depuis une liste de listes :

```
>>> from numpy import array
>>> a = array([[1, 2, 3], [10, 20, 30]]
>>> from pandas import DataFrame
>>> d= DataFrame(a, columns=["u", "d", "t"])
```

Ou encore depuis un dictionnaire de listes (chaque liste représentant alors une colonne) :

```
d= DataFrame({"u": [1, 10], "d": [2, 20], "t": [3, 30]})
```

Chacun de ces exemples mène à la création de cet objet :

```
>>> d
    u   d   t
0   1   2   3
1  10  20  30
```

On peut également voir ceci comme une liste de séries, lesquelles partageraient toutes les mêmes indices (qui sont les noms des colonnes) :

```
>>> d2 = DataFrame({'u': Series([1, 10, 100]), 'a': Series([42, 34, 31])})
```

Lorsque l'on crée des dataframes, on crée en général les indices qui vont avec :

```
>>> d= DataFrame([{"u": 1, "d": 2, "t": 3},
                  {"u": 10, "d": 20, "t": 30}], index=['a', 'b'])
>>> d
    d   t   u
a   2   3   1
b  20  30  10
```

Ou dans le cas des séries réutiliser leur index :

```
>>> d2 = DataFrame({'u': Series([1, 10, 100], index=['a', 'b', 'c']),
                    'x': Series([42, 34, 31], index=['a', 'b', 'c'])})
>>> d2
     u   x
a    1  42
b   10  34
c  100  31
```

Voici quelques éléments simples pour récupérer des informations de base sur le dataframe :

```
>>> d.shape
(2, 3)
>>> len(d)
2
>>> len(d.columns)
3
>>> d.describe()
              d          t          u
```

© Editions ENI - All rights reserved

```
count   2.000000   2.000000   2.000000
mean   11.000000  16.500000   5.500000
std    12.727922  19.091883   6.363961
min     2.000000   3.000000   1.000000
25%     6.500000   9.750000   3.250000
50%    11.000000  16.500000   5.500000
75%    15.500000  23.250000   7.750000
max    20.000000  30.000000  10.000000
>>> d.columns.values
array(['d', 't', 'u'], dtype=object)
```

On dispose aussi de **`d.head(x)`** ou **`d.tail(x)`** qui permettent de montrer les **`x`** premiers éléments d'un dataframe, **`x`** étant optionnel et valant 5 par défaut. Ces derniers sont particulièrement utiles pour visualiser le début ou la fin des données et visualiser leur structure.
On peut facilement extraire une série correspondant à une colonne :

```
>>> d["t"]
a     3
b    30
Name: t, dtype: int64
```

Ou à une ligne :

```
>>> d.loc["a"]
d    2
t    3
u    1
Name: a, dtype: int64
```

On peut récupérer des informations partielles en utilisant plusieurs techniques :

```
>>> d["t"][1:]
b    30
Name: t, dtype: int64
>>> d.loc["a"][1:2]
t    3
Name: a, dtype: int64
>>> d.loc[["a", "b"], ["t", "d"]]
    t   d
a   3   2
b  30  20
>>> d.loc[:, ["t", "d"]]
    t   d
a   3   2
b  30  20
>>> d.loc[["a", "b"],:]
    d   t   u
a   2   3   1
b  20  30  10
>>> d.loc["a", "t"]
3
```

On peut réaliser exactement la même chose en travaillant sur des indices :

```
>>> d.iloc[:,1]
a     3
b    30
Name: t, dtype: int64
>>> d.iloc[0]
d    2
t    3
u    1
Name: a, dtype: int64
>>> d.iloc[:,1][1:]
b    30
Name: t, dtype: int64
>>> d.iloc[0][1:2]
t    3
Name: a, dtype: int64
>>> d.iloc[0:1,[1, 2]]
   t  u
a  3  1
>>> d.iloc[0:2,[1, 2]]
    t   u
a   3   1
b  30  10
>>> d.iloc[0:2,[1, 0]]
    t   d
a   3   2
b  30  20
>>> d.iloc[:,[1, 0]]
    t   d
a   3   2
b  30  20
>>> d.iloc[0:2,:]
    d   t   u
a   2   3   1
b  20  30  10
```

Comme pour les séries, on peut procéder à de la recherche de données. Cette recherche se base sur des conditions impactant les valeurs associées à des colonnes :

```
>>> d[d['t'] % 2 == 0]
    d   t   u
b  20  30  10
>>> d[(d['t'] % 2 == 0) & (d['u']>16)]
Empty DataFrame
Columns: [d, t, u]
Index: []
>>> d[(d['t'] % 2 == 0) & (d['u']<16)]
    d   t   u
b  20  30  10
```

© Editions ENI - All rights reserved

Ceci permet de ne conserver que les individus correspondant aux valeurs souhaitées.
Il est également possible de travailler sur l'unicité des individus, en supprimant les doublons :

```
>>> du= DataFrame([{"u": 1, "d": 2, "t": 3},
                   {"u": 10, "d": 20, "t": 30},
                   {"u": 1, "d": 2, "t": 3}], index=['a', 'b', 'c'])
>>> du.drop_duplicates()  # Ne garde que la première occurrence du doublon
    d   t   u
a   2   3   1
b  20  30  10
>>> du.drop_duplicates(keep=False)  # Ne garde aucune occurrence du
doublon
    d   t   u
b  20  30  10
>>> du.drop_duplicates(keep="last")  # Ne garde que la dernière occurrence
du doublon
    d   t   u
b  20  30  10
c   2   3   1
```

Il est possible de modifier l'objet lui-même au lieu de renvoyer une copie modifiée :

```
>>> du.drop_duplicates(keep=False, inplace=True)
>>> du
    d   t   u
b  20  30  10
```

Il est aussi possible de concaténer plusieurs dataframes :

```
>>> from pandas import concat
>>> concat([d, du])
    d   t   u
b  20  30  10
a   2   3   1
b  20  30  10
```

Le fait que les indices soient identiques ne va pas faire disparaître une des lignes, ces index sont simplement répétés. On peut aussi faire l'inverse, c'est-à-dire travailler à indices identiques :

```
>>> concat([d, du], axis=1)
    d   t   u     d     t     u
a   2   3   1   NaN   NaN   NaN
b  20  30  10  20.0  30.0  10.0
```

Dans le cas où les colonnes ne sont pas identiques, les données sont fusionnées :

```
>>> concat([d, d2], sort=False)
      d     t     u      q
a   2.0     3     1    NaN
b  20.0    30    10    NaN
d   NaN   300   100  400.0
```

```
e   NaN  3000  1000  4000.0
>>> concat([d, d2], sort=True)
      d       q     t     u
a   2.0     NaN     3     1
b  20.0     NaN    30    10
d   NaN   400.0   300   100
e   NaN  4000.0  3000  1000
```

Si on ne souhaite conserver que les valeurs partagées par les deux dataframes, on peut faire ceci :

```
>>> concat([d, d2], sort=True, join='inner')
     t     u
a    3     1
b   30    10
d  300   100
e 3000  1000
```

Comme pour les séries, on peut réaliser quelques opérations essentielles sur les dataframes, à la manière des arrays NumPy :

```
>>> d+du
      d     t     u
a   NaN   NaN   NaN
b  40.0  60.0  20.0
>>> d * 2
    d   t   u
a   4   6   2
b  40  60  20
>>> d * 2 + 42
    d    t   u
a  46   48  44
b  82  102  62
>>> 1 / d
      d         t    u
a  0.50  0.333333  1.0
b  0.05  0.033333  0.1
>>> d ** 3
      d      t     u
a     8     27     1
b  8000  27000  1000
```

Toujours comme les séries, on peut faire des opérations de tri, mais celles-ci sont plus complexes :

```
>>> d
    d   t   u
a   2   3   1
b  20  30  10
>>> d.sort_index(axis=0, ascending=False)
    d   t   u
```

© Editions ENI - All rights reserved

```
b  20  30  10
a   2   3   1
>>> d.sort_index(axis=1, ascending=False)
    u   t   d
a   1   3   2
b  10  30  20
>>> d.sort_values(by="t")
    d   t   u
a   2   3   1
b  20  30  10
>>> d.sort_values(by=["t", 'u'], ascending=False)
    d   t   u
b  20  30  10
a   2   3   1
>>> d.sort_values(by=["t", 'u'], ascending=False, inplace=True)
>>> d
    d   t   u
b  20  30  10
a   2   3   1
```

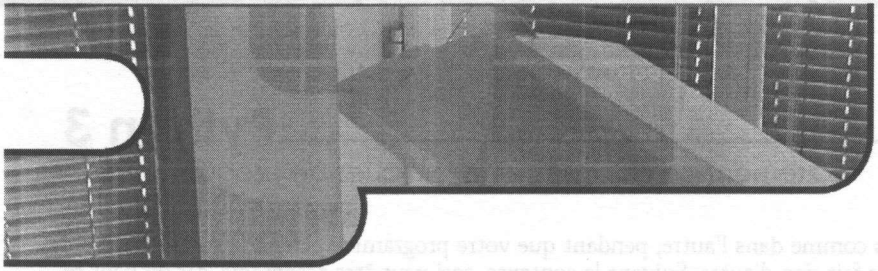
Enfin, voici comment regrouper des données et faire des opérations sur ces groupes :

```
>>> d= DataFrame([{"a": 1, "b": 2, "c": 1},
                  {"a": 2, "b": 20, "c": 1},
                  {"a": 1, "b": 12, "c": 4},
                  {"a": 2, "b": 3, "c": 6},
                  {"a": 3, "b": 7, "c": 9},
                  {"a": 2, "b": 9, "c": 2}],
                 index=['a', 'b', 'c', 'd', 'e', 'f'])
>>> d
   a   b  c
a  1   2  1
b  2  20  1
c  1  12  4
d  2   3  6
e  3   7  9
f  2   9  2
>>> d.groupby('a').sum()
    b  c
a
1  14  5
2  32  9
3   7  9
>>> d.groupby('a').mean()
           b    c
a
1   7.000000  2.5
2  10.666667  3.0
3   7.000000  9.0
```

```
>>> d.groupby('a').median().reset_index()  # Remet 'a' comme une colonne
   a    b    c
0  1  7.0  2.5
1  2  9.0  2.0
2  3  7.0  9.0
>>> d.groupby('a').std().reset_index()
   a         b         c
0  1  7.071068  2.121320
1  2  8.621678  2.645751
2  3       NaN       NaN
>>> d.groupby('a').prod().reset_index()
   a    b   c
0  1   24   4
1  2  540  12
2  3    7   9
>>> d.groupby('a').quantile(q=0.1).reset_index()
0.1  a    b    c
0    1  3.0  1.3
1    2  4.2  1.2
2    3  7.0  9.0
>>> d.groupby('a').agg(min).reset_index()
   a  b  c
0  1  2  1
1  2  3  1
2  3  7  9
>>> d.groupby('a').agg("mean").reset_index()
   a          b    c
0  1   7.000000  2.5
1  2  10.666667  3.0
2  3   7.000000  9.0
>>> d.groupby('a').agg(["mad", max]).reset_index()
   a         b         c
           mad max  mad max
0  1  5.000000  12  1.5   4
1  2  6.222222  20  2.0   6
2  3  0.000000   7  0.0   9
>>> d.groupby('a').filter(lambda x: x["b"].min() > 5)
   a  b  c
e  3  7  9
```

Ce dernier exemple permet de filtrer les données par rapport à un calcul appliqué sur le groupe. Il s'agit d'un exemple à la fois complexe et élégant de ce que l'on peut faire avec Pandas. Cela fait appel à des principes de programmation fonctionnelle classiques avec Python, mais appliqués d'une manière différente.

© Editions ENI - All rights reserved



Partie 3 : Programmation concurrente

Chapitre 3.1 Initiation à la programmation concurrente

1. Notion de performance

Avant de parler de performance, il faut d'abord savoir analyser quels sont les paradigmes utilisés, quelle est la complexité du programme et enfin, quelle est l'architecture matérielle qui va faire tourner le programme.

En effet, il y a énormément de détails sur lesquels le programme n'a aucune maîtrise. On peut citer, par exemple, la nature du système d'exploitation qui le fait tourner, la quantité et la consommation des autres programmes qui tournent à coté, la quantité de mémoire, le processeur ou encore le débit réseau.

Sachez qu'on ne parle pas, dans cette partie, de la performance dans le sens optimisation au niveau algorithmique du code, mais de solutions au niveau de l'architecture de l'application elle-même.

1.1 Programmation bloquante

Tout programme passe par des phases où il est en pause. Ceci peut être dû au fait qu'il attende quelque chose de l'utilisateur :

```
data = read("Saisissez une donnée")
```

Ou parce qu'il attend après une ressource quelconque : I/O, disque dur, périphérique, ou encore réseau :

```
import requests
response = requests.get('http://inspyration.org')
```

Dans cet exemple, le programme est en attente de la résolution de nom (DNS), puis de la réponse du serveur qui va potentiellement mettre la requête en file d'attente puis travailler de son côté pendant un petit moment avant de commencer à envoyer sa réponse.

Dans un cas comme dans l'autre, pendant que votre programme attend la ressource, il est bloqué : il ne fait rien d'autre. Suivant le contexte, ceci peut être acceptable, car on peut se permettre de perdre jusqu'à 300, voire 500 millisecondes sans que l'utilisateur n'en ressente la gêne.
Cependant, dans d'autres contextes, ceci est plus ennuyeux :

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(response.content, "html.parser")
image_urls = [img.get("src") for img in soup.find_all("img")]
image_contents = [requests.get("http://inspyration.org/{}".format(url))
                  for url in image_urls]
```

Le programme va parcourir le code HTML et extraire toutes les images présentes. Il y a ensuite les télécharger l'une après l'autre. On vivra ici des attentes dues au réseau et si on subit une attente d'une demi seconde pour chaque image, cela donne au total 5 secondes d'inactivité du programme pour 10 images, ce qui est cette fois-ci du domaine de l'inacceptable.
Améliorer sa performance peut donc se faire en diminuant les temps d'attente. Par exemple dans ce cas en lançant tous les téléchargements en même temps et en récoltant les données au fur et à mesure qu'elles arrivent.
En procédant ainsi, on perd un peu de sécurité de programmation, car on ne sait pas dans quel ordre les données vont arriver. Comme on le verra, ce n'est pas un souci, car Python est fait de telle manière qu'il garantit la sécurité de programmation autrement.
On va télécharger ici plusieurs fichiers et répéter le temps d'attente autant de fois qu'il y a de fichiers. Le programme va donc potentiellement passer un gros pourcentage du temps à attendre, à être bloqué.
Transformer le programme en utilisant plusieurs fils d'exécution ou processus n'améliorera pas fondamentalement les choses car, si l'on réduit les temps d'attente, on payera les coûts de création de ces tâches ou processus et les coûts d'ordonnancement.
La solution dans ce cas est d'utiliser la programmation asynchrone.

1.2 Utilisation de ressources CPU

Dans le cas d'une application continuellement active faisant tourner de lourds algorithmes, par exemple un algorithme de retouche d'image, il est quasiment impossible de gagner significativement du temps en réduisant les temps d'attente, car il n'y en a pas. Le souci n'est pas l'I/O, mais le fait que l'on utilise les capacités de la machine qui fait tourner votre programme au maximum de ce qu'elle peut lui offrir.
En effet, lorsque vous lancez votre programme, le système crée un processus qui va démarrer la machine virtuelle Python et c'est cette dernière qui va exécuter votre programme. Et chaque processus a, par défaut, un seul fil d'exécution.
L'idée générale est d'augmenter la capacité de traitement de l'information en augmentant la quantité de temps CPU que le programme peut utiliser. S'il est actif tout le temps ou quasiment tout le temps, c'est-à-dire jamais ou quasiment jamais bloqué par l'I/O, alors augmenter la quantité de CPU attribué à ce programme est le seul moyen d'améliorer ses performances.
Et pour cela, il existe plusieurs méthodes.

© Editions ENI - All rights reserved

De nos jours, nous disposons assez couramment d'ordinateurs multi-cœurs et, selon l'environnement, vous avez peut-être accès à plusieurs processeurs. Suivant les autres applications qui tournent sur cette machine, il est raisonnable de penser que l'on pourrait utiliser plusieurs processus ou plusieurs fils d'exécution. Ainsi, au lieu de n'avoir qu'un seul cœur d'un seul processeur qui exécuterait le programme, il pourrait y avoir plusieurs cœurs de plusieurs processeurs (savoir lesquels est une considération système qu'on laisse à la bonne gestion du système d'exploitation, ce n'est pas de notre ressort et ce n'est pas important : on sait que le système d'exploitation fera au mieux).

Un autre moyen assez malin est d'utiliser le processeur de la carte graphique. En effet, ces processeurs sont peut-être moins rapides que les processeurs centraux, mais ils ont en général beaucoup plus de cœurs et peuvent par conséquent traiter beaucoup de processus en parallèle.

Si l'application est hébergée dans un centre de serveurs, il est possible de séparer l'application sur plusieurs serveurs et d'accéder aux différents cœurs des différents CPU de chaque serveur, ce qui permet d'augmenter d'autant la capacité de traitement. On parle de l'utilisation d'une grappe de serveurs.

Vous pouvez même procéder à ce genre de chose chez vous et pour pas trop cher en achetant plusieurs RaspberryPI et en déployant votre application sur chacun d'entre eux. Là encore, leurs processeurs ne sont peut-être pas très rapides, mais ils possèdent plusieurs cœurs et si l'on sait en tirer parti, ces outils de la technologie moderne feront des merveilles.

Nous avons vu ici que face à un problème donné il existe de nombreuses solutions. Mais ces solutions ont un coût. À la fois sur l'aspect financier, puisque payer un serveur n'a pas la même conséquence qu'en payer une grappe et à la fois sur le développement puisque chaque solution fait appel à des paradigmes différents et rajoute de nouvelles difficultés.

1.3 Organisation de l'application

Un autre axe d'optimisation consiste à séparer ce que fait l'application en tâches unitaires et à séparer ce qui impacte l'utilisateur, ce qui doit être fait de suite et ce qui peut attendre.

Voici un exemple simple : un utilisateur se trouve sur une page web qui est le formulaire de contact du site d'une société. Il remplit le formulaire et le valide. L'application web va alors créer une entrée dans la base de données pour enregistrer ce que l'utilisateur a saisi, puis va renvoyer une page signalant une erreur ou informant l'utilisateur que son message a bien été enregistré. Mais elle va également envoyer un mail au responsable commercial de la société pour l'informer qu'il a reçu une demande d'un contact.

Or, cette dernière tâche d'envoi de mail n'est pas indispensable pour l'utilisateur qui remplit le formulaire. Si elle prend une seconde, c'est une seconde de temps d'attente supplémentaire pour l'utilisateur. Et cela peut compter pour l'image de la société derrière le site. Par contre, si le responsable commercial reçoit le mail tout de suite ou une seconde plus tard ou même une minute plus tard, cela n'a pas d'impact fondamental pour lui, ni même pour l'utilisateur qui a rempli le formulaire de contact.

L'idée alors serait de ne pas envoyer le mail tout de suite, mais de prendre en compte cette tâche et de l'exécuter plus tard, à un moment où le serveur qui héberge l'application a un peu de temps de disponible.

Ainsi, le fait d'identifier toutes les tâches exécutées par l'application, qui ne sont pas essentielles dans l'instant présent et peuvent être remises à plus tard (sachant que plus tard veut dire quelques secondes plus tard), peut permettre à cette application de se concentrer sur les aspects essentiels et sur le fait de rendre la main à l'utilisateur au plus tôt. Et dès qu'elle est en attente d'une action de l'utilisateur, elle peut en profiter pour jouer ces tâches mises en attente.

L'idée serait que l'application elle-même gère le moment de jouer les tâches en attente.

On peut aussi concevoir son application en isolant des parties critiques et en la séparant en plusieurs parties. Ainsi, on peut imaginer une partie dédiée à l'interface avec l'utilisateur, qui serait toujours active et rendrait la main très vite, et une autre partie faisant le travail en sous-main et informant la première partie de ses avancées.

Suivant ce que vous développez (un petit script, une application complexe isolée ou une application complexe interagissant avec de nombreuses autres applications ou services ou I/O de manière générale), vous aurez divers outils à votre disposition pour augmenter la capacité de traitement de votre application.

2. Terminologie

On va présenter dans cette partie quelques notions techniques qu'il est nécessaire de comprendre pour faire ses choix.

2.1 Processus

Un processus est l'exécution d'un ensemble d'instructions par l'utilisation de ressources physiques, les deux principales étant la mémoire vive dans laquelle il stocke son environnement d'exécution et le processeur qui lui est affecté.

Les opérations de création, gestion et destruction d'un processus sont gérées par le système d'exploitation, pas directement par Python. Python propose des outils de haut niveau pour créer, gérer et détruire ces processus qui restent relativement simples, mais il utilise en réalité le système d'exploitation qui fait l'essentiel du travail.

Et heureusement, car ce dernier est extrêmement complexe et fait intervenir des notions de programmation système très pointues. Le processus est géré par un ordonnanceur qui est dépendant du système d'exploitation. Ce dernier est chargé de mettre à disposition les ressources (mémoire, temps processeurs...) et de veiller à ce que chaque processus accède à ces ressources de manière équitable. S'il y a plusieurs processeurs, les processus sont également distribués entre eux de manière équitable.

Un programme lancé par un utilisateur peut correspondre à un seul processus ou à plusieurs. Par exemple, au lancement, Apache crée six processus :

```
$ ps ax | grep apache
 1569 ?        Ss     0:00 /usr/sbin/apache2 -k start
 1584 ?        S      0:00 /usr/sbin/apache2 -k start
 1585 ?        S      0:00 /usr/sbin/apache2 -k start
 1586 ?        S      0:00 /usr/sbin/apache2 -k start
 1587 ?        S      0:00 /usr/sbin/apache2 -k start
 1588 ?        S      0:00 /usr/sbin/apache2 -k start
 3067 pts/1    S+     0:00 grep apache
```

© Editions ENI - All rights reserved

De la même manière, chaque commande en ligne exécutée dans le terminal lance un processus. Ainsi, dans l'exemple précédent, deux processus ont été invoqués : la commande **ps** en a lancé un et la commande **grep** un autre.
Ces deux programmes ont communiqué entre eux à l'aide d'un pipe, qui est un moyen d'échanger de la donnée entre processus sans passer par un fichier temporaire ou quoi que ce soit d'autre, simplement en partageant de la mémoire.

Remarque

Pour plus d'informations sur le fonctionnement du pipe du terminal :
https://brandonwamboldt.ca/how-linux-pipes-work-under-the-hood-1518/

Chaque processus dispose de son propre environnement d'exécution et est indépendant. Le coût de création d'un processus est relativement élevé, il faut donc veiller à ce que ce coût ne surpasse pas les gains éventuels. Dans beaucoup de cas d'utilisation, un processus peut faire son travail de manière autonome. Cependant, parfois, il est nécessaire de faire communiquer les processus entre eux.
Lorsque plusieurs processus doivent échanger de l'information, il faut mettre en place une architecture relativement complexe. Il faut donc prendre le temps de concevoir la manière dont on va créer ses processus et les outils à mettre en place pour les faire travailler ensemble.
Un programme est dit multi-processus dès lors qu'il crée plusieurs processus. Ce genre de technologie est idéale lorsque l'on dispose de plusieurs processeurs ou de plusieurs cœurs.
Le module Python **multiprocessing** fournit tous les outils de haut niveau pour gérer les processus.

2.2 Fil d'exécution

Chaque processus contient par défaut un fil d'exécution. Mais il est possible d'en créer plusieurs. Présentant les mêmes caractéristiques qu'un processus d'un point de vue externe au système, un fil d'exécution permet, comme son nom l'indique, d'exécuter des instructions. La différence notable est que lorsque l'on a plusieurs fils d'exécution, tous utilisent le même contexte d'exécution, celui du processus auquel ils appartiennent. Cependant, ils possèdent leur propre pile d'appel, liée à l'exécution de leurs instructions.
Entre fils d'exécution, que certains appellent processus légers, le parallélisme n'est pas réel en tant que tel. Par contre, il est possible de concevoir une délégation de travail vers des fils d'exécution secondaires pour fluidifier le fil d'exécution principal et de gérer au mieux les temps d'attente éventuels. Eux aussi sont gérés par les systèmes d'exploitation.
Par rapport au processus, le fil d'exécution est moins coûteux, mais plus dépendant.
Le fil d'exécution se nomme *thread* en anglais et le module Python **threading** contient tous les outils de haut niveau pour le gérer.
Ce module propose volontairement une interface très similaire à celle de **multiprocessing** pour que l'on puisse utiliser les fils d'exécution ou les processus de manière identique (du point de vue du développeur Python, car ce n'est, bien entendu, pas la même chose à bas niveau).

2.3 Programmation non bloquante

Python propose la notion de greenlet et un module **gevent** qui propose une interface similaire à celle des fils d'exécution ou des processus, sauf qu'il s'agit ici de remplacer les outils classiques de Python par d'autres outils non bloquants. On va alors proposer une solution qui, sur un plan technique, est de l'asynchrone, mais qui, sur un plan du code, ressemble à l'utilisation de fils d'exécution ou de processus. Il existe également les notions de pool et de file d'attente.

L'avantage principal de cet outil consiste à écrire du code similaire à ce que l'on fait en programmation parallèle tout en étant moins coûteux que les fils d'exécution et donc encore moins que les processus, puisque techniquement, aucun nouveau fil d'exécution n'est créé (ni aucun processus).

2.4 Notion de GIL

Cela peut être le bon moment pour parler de la GIL (*Global Interpreter Lock*, soit verrou global de l'interpréteur).

Lorsque l'on crée plusieurs fils d'exécution (ou *thread* en anglais), on va utiliser le système d'exploitation et lui confier la gestion de ces derniers. Ils peuvent alors s'exécuter en parallèle, dans différents cœurs. Pour Python, c'est également le cas.

Par contre, en Python, tout comme en Ruby, PHP ou JavaScript, il est implémenté un GIL : un verrou sur les données du programme, absolument nécessaire pour qu'un fil d'exécution puisse s'exécuter. En conséquence, cela permet de s'assurer qu'un seul fil d'exécution s'exécute à tout instant : celui qui possède l'accès aux données du programme. Ce système apporte de la sécurité de programmation et simplifie grandement la tâche de la machine virtuelle Python, mais l'inconvénient est qu'au lieu de se reposer sur le système, elle doit ordonnancer elle-même ces fils d'exécution et cela coûte à Python des performances. En effet, plus il y a de fils d'exécution et plus le GIL coûte de la performance à la machine virtuelle Python.

Un des gros chantiers pour le passage à Python 3 a été de travailler sur cet aspect pour limiter les contraintes qu'il impose, mais de nombreux développeurs pensent aujourd'hui que ce GIL doit être purement et simplement supprimé. Ceci est à mettre en balance avec la sécurité de programmation que le GIL apporte et le fait que Python a énormément progressé dans le domaine de l'asynchrone et qu'il peut aussi travailler avec des processus.

Les progrès sur l'aspect asynchrone ne suffiront-ils pas à résoudre la plupart des problèmes de performance ? D'autant plus que l'utilisation des fils d'exécution a aussi un autre inconvénient sur lequel le développeur n'a pas la main : on ne sait jamais quand il va se mettre en pause.

Vu ainsi, on peut tourner la question autrement : d'autres outils à disposition des développeurs ne peuvent-ils pas avantageusement remplacer l'utilisation de plusieurs fils d'exécution ?

Est-ce que le risque de perdre de la sécurité de programmation vaut les gains des performances espérés par la suppression du GIL ? Si un tel verrou est nécessaire, c'est que la manière dont Python gère la mémoire n'est pas thread-safe, ce qui oblige à gérer un mécanisme de synchronisation qui lui-même force à la mise en place du GIL. D'un autre côté, un langage comme le C n'a pas besoin d'un GIL tout comme d'autres implémentations de Python telles que CPython.

© Editions ENI - All rights reserved

Saura-t-on améliorer le problème d'origine ? En d'autres termes, saura-t-on rendre la gestion de la mémoire de Python thread-safe ?
Autant de questions qui font débat au sein de la communauté des contributeurs Python.

3. Présentation des paradigmes

3.1 Programmation asynchrone

Lorsque la problématique principale est le temps I/O, la solution idéale est de programmer en asynchrone. L'idée est que l'on continue à faire tourner notre programme par un processus ne contenant qu'un fil d'exécution, on ne s'embête pas à gérer des tâches ou des processus, qui sont un outil relativement lourd à mettre en place.
La programmation asynchrone consiste à déclarer qu'une des tâches que le programme doit exécuter peut être potentiellement bloquante (être en attente d'une I/O et n'avoir rien à faire pendant ce temps) et le programme pourra alors continuer à exécuter la suite du code en attendant.
C'est, par exemple, un des points forts de JavaScript, un langage qui est conçu avec des objectifs particuliers : il est exécuté par un navigateur et c'est le navigateur qui gère les fils d'exécution et processus. JavaScript aura donc toujours, dans ce contexte, un seul fil d'exécution et un seul processus et étant donné qu'il exécute des instructions dans un contexte réseau, il y a forcément énormément d'attente I/O.
C'est donc pour cela que le levier asynchrone est pour ce langage la meilleure option pour améliorer la performance et c'est pour cette raison qu'il y excelle, tous les efforts étant portés vers cela.
Ainsi, lorsque l'on vante les mérites de Node.js, on ne fait que vanter la bonne utilisation des capacités du langage lui-même. Node.js fait simplement une bonne utilisation de l'asynchrone et sait en tirer parti. C'est aussi ce que d'autres serveurs Python très anciens et très connus font parfaitement depuis des années, tels que Tornado ou Twisted, mais chacun à sa manière.
Un des axes d'amélioration de la branche 3.x de Python est justement le développement de ces capacités de gestion de l'asynchrone. Le chapitre suivant est dédié à la présentation de ces fonctionnalités et présente quelques exemples concrets.

3.2 Programmation parallèle

La programmation parallèle est le fait de rajouter de la puissance du CPU de la même machine en créant plusieurs fils d'exécution ou plusieurs processus pour répartir la charge de travail.
Utiliser la programmation parallèle nécessite un peu de travail, mais ce n'est pas, en Python, une tâche insurmontable. Cependant, cela n'apportera pas autant que souhaité si le vrai problème est l'I/O. Dans ce cas- là, il sera toujours préférable de se tourner vers une solution asynchrone qui demandera à la fois moins de travail au développeur et qui apportera une résolution plus efficace à la problématique de la performance.

Une fois ceci exposé, il faut définir la charge de travail et la manière de la répartir. Lorsque l'on parle de programmation parallèle, on parle essentiellement d'un travail à reproduire plusieurs fois. Par exemple, on a un répertoire contenant beaucoup d'images et pour chacune, il faut appliquer des filtres, créer un cadre, rajouter un texte et rajouter un code stéganographique.

On pourra alors créer une fonction permettant de faire toutes ces opérations sur une seule image et créer plusieurs processus, par exemple. Imaginons que nous avons 500 images à traiter, il est possible de créer 4 processus et de leur donner 125 images à chacun.

Encore mieux, grâce à des interfaces de haut niveau, il est possible de créer un pool d'exécution de 4 processus et de lui dire qu'il y a 500 photos à traiter. Lorsque chaque processus aura terminé de traiter une image, il ira chercher la suivante, jusqu'à ce qu'il n'y en ait plus. À la fin, un des processus pourra n'en avoir traité que 80, alors qu'un autre en aura fait 150, suivant la taille de ces images et leur temps de traitement individuel. On optimisera ainsi le temps de traitement global. L'avantage de cette méthode est que l'on ne crée pas un processus pour une seule tâche, on réutilise plusieurs fois les mêmes 4 processus créés au tout début. Il s'agit ici de répartir les tâches au fur et à mesure plutôt que de le faire a priori.

Selon la charge de travail que l'on demande au processeur, il faudra choisir entre utiliser plusieurs fils d'exécution et plusieurs processus. En effet, le coût de création d'un processus est assez important et si le travail qu'on lui confie est trop faible au regard de ce coût, cela pourrait même s'avérer contre-performant. De plus, les processus ont une mémoire qui leur est propre et, bien qu'il soit possible de faire communiquer deux processus entre eux, on préférera l'éviter dans la mesure du possible. Si le fait de partager la mémoire entre deux processus est vraiment indispensable, peut-être utiliser des fils d'exécution sera plus simple à mettre en œuvre.

Une fois ces considérations exposées, ce qui est appréciable en Python est que les deux modules **`multiprocessing`** et **`threading`** partagent une interface similaire, c'est-à-dire que l'on peut facilement remplacer l'un par l'autre, du point de vue du développeur. Les traitements en arrière-plan seront réalisés de manière transparente.

De plus, on verra aussi un module de haut niveau permettant de répartir la charge à la volée.

3.3 Programmation distribuée

La programmation distribuée est un type de programmation concurrente. Il s'agit d'utiliser un outil permettant la distribution des tâches. C'est cet outil qui s'occupera de gérer l'ordonnancement optimal de ces dernières.

On peut distinguer deux cas. Dans le premier cas, l'outil est sur la même machine et il va optimiser l'utilisation des CPU pour distribuer les tâches intelligemment. Dans le second cas, on va utiliser une grappe de machines et la distribution se fera sur l'ensemble de ses serveurs, ce qui laisse la possibilité d'ajuster la charge en rajoutant ou supprimant des serveurs dans la grappe.

Il existe aujourd'hui la notion de Big Data qui utilise énormément la programmation distribuée pour rechercher de la performance lors du traitement de gros volumes de données, mais on retrouve également cette solution lorsqu'il s'agit de déléguer des tâches.

On peut citer par exemple l'outil Celery qui permet, par exemple, de reporter des tâches annexes telles que l'envoi d'un mail après la réalisation d'une action donnée ou encore de reporter les opérations lourdes pour les exécuter en arrière-plan pendant que l'opérateur est informé que ce qu'il a demandé a bien été pris en compte et qu'il peut faire autre chose.

© Editions ENI - All rights reserved

3.4 Programmation concurrente

Avant de parler de programmation concurrente, il convient de parler de concurrence. La concurrence consiste à faire plusieurs choses en même temps. En clair, utiliser le module **`threading`**, c'est faire de la concurrence avec des fils d'exécution tandis qu'utiliser le module **`multiprocessing`**, c'est faire de la concurrence avec des processus. Il y a aussi la programmation asynchrone qui consiste à faire de la concurrence avec un seul processus et un seul fil d'exécution, mais en changeant de tâche aux moments optimaux. La notion de programmation distribuée est aussi une forme de concurrence, mais qui porte cette fois-ci sur plusieurs applications ou plusieurs serveurs.

En effet, à partir du moment où un processus a deux fils d'exécution, ces deux fils sont en concurrence pour le CPU ainsi que la zone mémoire qu'ils partagent. De même, si deux processus sont assignés au même processeur, ils sont également en concurrence pour l'accès à ce processeur. Même lorsque deux applications travaillent ensemble ou encore deux serveurs, ils peuvent être en concurrence pour l'accès à des ressources externes.

Dans ce contexte, la notion de programmation concurrente est associée dans certaines documentations à toutes ces formes de concurrence, elle recouvre donc, dans ce contexte, les paradigmes asynchrones, parallèles et distribués.

Dans d'autres documentations, la programmation concurrente désigne spécifiquement les solutions que l'on peut mettre en place pour permettre à différentes parties d'un programme de collaborer pour se partager les ressources de manière optimale. Ceci mène naturellement aux solutions que l'on peut mettre en place pour construire son application de manière concurrente.

Prenons donc un exemple : on peut imaginer une application qui est un serveur web proposant une application métier, du type Django. Cette application pourrait être associée à un autre processus qui serait chargé de recevoir des mails et de créer des objets en base en fonction de ce qu'il reçoit, ainsi qu'à un troisième chargé de surveiller la base de données et d'envoyer des notifications lorsqu'un déclencheur est activé.

Dans ce contexte très spécifique, les trois processus s'exécutent de manière concurrente et leur ressource partagée est la base de données, ce qui tombe particulièrement bien, car les bases de données sont conçues pour être hautement concurrentes.

Dans cette situation, il n'y a pas de soucis particuliers et cela est même relativement facile à gérer : on construit trois applications indépendantes qui accèdent à la même base de données.

Là où cela se complique, c'est lorsque l'on a des processus qui doivent partager du CPU, une zone mémoire ou des I/O identiques.

Dans la notion de programmation parallèle, nous avons présenté un exemple consistant à appliquer des filtres, créer un cadre, rajouter un texte et rajouter un code stéganographique sur une grande quantité d'images. On rajoute maintenant la condition que, pour chaque image, au cas par cas, il faudrait ne faire qu'une partie des opérations.

On imagine que sur la première image, il faudrait seulement appliquer les filtres et mettre du texte, tandis que pour la seconde il faudrait réaliser toutes les opérations, pour la troisième seulement l'opération stéganographique, et ainsi de suite.

Utiliser la programmation concurrente consisterait à créer un processus spécialisé pour chaque étape de la chaîne de traitement et faire passer les images d'un processus à un autre en fonction de ce qui est requis.

Pour comparer avec une situation quotidienne, imaginez que vous avez une équipe de 4 ouvriers qui doivent construire 20 maisons. Soit vous parallélisez la tâche en demandant à chaque ouvrier de construire 5 maisons, soit vous assignez un rôle à chaque ouvrier : un qui monte les murs et le toit, un autre pour l'électricité, un autre pour la plomberie et le dernier pour la peinture et l'équipement.

Dans ce dernier cas, chaque ouvrier travaillera dans chaque maison en y faisant quelque chose de spécifique.

Par contre, comme les ouvriers ne gèrent pas toute la chaîne de bout en bout, ils devront communiquer entre eux. C'est aussi la problématique essentielle de la programmation concurrente.

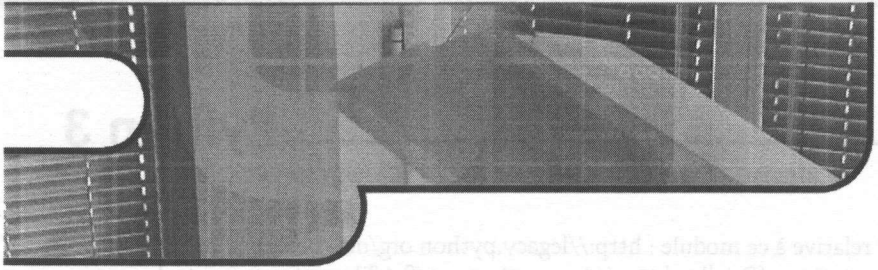
En effet, faire communiquer des processus entre eux n'est pas une mince affaire et leur faire utiliser des ressources partagées non plus. Il faut mettre en place des mécanismes pour s'assurer que l'un de ces problèmes ne se pose pas, tels que les verrous, l'exclusion mutuelle ou les sémaphores.

Dans notre exemple, on aurait un processus qui gérerait uniquement l'application de filtres, un autre la création du cadre, un autre le rajout du texte et un dernier pour le rajout d'un code stéganographique. Chaque processus est totalement indépendant et gère seul sa partie. Mais chaque processus doit communiquer avec les autres pour transmettre des données. Et cette communication peut être complexe.

La programmation concurrente consiste à se baser sur toutes les techniques issues de la programmation asynchrone, parallèle ou distribuée afin de concevoir une application ultra-concurrente en choisissant pour chaque problématique le bon type de programmation et en assurant l'utilisation optimale des ressources.

Dans les chapitres suivants, nous aborderons la programmation asynchrone, la manière de l'utiliser pour réduire les temps I/O et pour faire communiquer différents programmes ensemble. Nous aborderons ensuite la programmation parallèle afin d'optimiser les opérations qui demandent beaucoup de calculs et nous détaillerons les différentes manières de faire communiquer des tâches entre elles ou des processus entre eux et de protéger l'accès à des ressources. Enfin, nous aborderons la programmation distribuée avec quelques solutions impliquant des architectures complexes, mais faciles à mettre en place et permettant d'économiser le processus principal de l'application.

© Editions ENI - All rights reserved



Chapitre 3.2
Programmation asynchrone : initiation

1. Utilité de la programmation asynchrone

La recherche de performance dans certains domaines est un point crucial. Comme on l'a vu précédemment, la programmation parallèle, c'est-à-dire l'utilisation de plusieurs tâches ou de plusieurs processus, permet d'améliorer les performances notablement. Mais elles ne sont pas toujours les solutions les plus adaptées et, surtout, elles font intervenir des notions système et augmentent la complexité de l'application. Dans certains cas, lorsque les causes du manque de performance des applications sont dues à des problématiques I/O, il existe une autre solution, laquelle consiste à faire de la programmation asynchrone. La programmation asynchrone permet d'éviter qu'une tâche soit bloquée par une opération lente dépendant d'un facteur externe, telle que la récupération de données depuis le réseau ou le disque dur (lecture de fichiers, requêtes bases de données, requête Internet) ou l'écriture sur le réseau ou le disque dur ou encore l'utilisation d'un périphérique.

L'inconvénient de la programmation asynchrone est que le code n'est plus exécuté de manière parfaitement prédictible, ce qui rend la détection d'erreurs un peu plus complexe.

L'asynchrone est particulièrement efficace dans de nombreux domaines, en particulier dans celui du Web, où il est devenu très à la mode avec l'émergence, entre autres, de Node.js, qui permet d'avoir de très bonnes performances non pas grâce à la qualité du langage ou de la bibliothèque elle-même, mais tout simplement grâce au fait qu'elle fonctionne en mode asynchrone et que dans ce domaine, les temps d'attente sont légions.

Python avait jusque-là quelques solutions, mais non idéales, telles que **`asyncore`**, ou plus ou moins complexes à mettre en place, telles que **`Twisted`** (un serveur internet – et non pas simplement un serveur web – qui est une merveille absolue, mais reste particulièrement difficile à appréhender) ou encore **`gevent`** ou **`Tornado`**.

Avec Python 3.4 est arrivé le module **`asyncio`** qui fait des miracles et qui permet d'avoir de bonnes performances tout en profitant des autres avantages naturels du langage Python.

Voici la PEP relative à ce module : http://legacy.python.org/dev/peps/pep-3156/
Et la documentation officielle : http://docs.python.org/3.4/library/asyncio.html
Notons qu'il a été rétroporté pour les versions plus anciennes de Python 3 : https://github.com/python/asyncio
Avec Python 3.5, l'asynchrone devient naturel avec l'adjonction de quelques mots-clés qui, bien utilisés, nous permettent de basculer d'un code synchrone vers un code asynchrone avec extrêmement peu d'efforts.
La PEP relative à ces ajouts est : https://www.python.org/dev/peps/pep-0492/
Notons pour être complet que ces mots-clés ne deviennent des mots réservés (dédiés uniquement à cette utilisation) qu'à partir de Python 3.7.

2. Introduction à l'asynchrone

2.1 Notion de coroutine

Remarque

Pour ce chapitre, le discours sera basé sur la version la plus récente de Python 3, à savoir Python 3.7 au moment de la rédaction de cet article. Le chapitre suivant reviendra sur la syntaxe à utiliser pour les versions antérieures.

Pour utiliser la programmation asynchrone, il faut identifier dans son code une partie qui est bloquante, c'est-à-dire qui passe du temps à attendre après des I/O.
Il faut donc sortir cette partie du code linéaire et en faire une **coroutine**.
Cette notion de coroutine est à la base de la programmation asynchrone.
Voici l'exemple le plus basique, adapté de la documentation officielle :

```
>>> import asyncio
>>> import time
>>> async def main():
...     print(f"hello {time.strftime('%X')}")
...     await asyncio.sleep(1)
...     print(f"world {time.strftime('%X')}")
...
```

La première ligne permet d'importer le module **asyncio**, qui est dédié aux problématiques de programmation asynchrone et la deuxième le module **time** qui va nous permettre de visualiser le temps pris réellement par les algorithmes.
La troisième ligne ressemble à la définition d'une fonction, sauf que le mot-clé **def** est précédé du mot-clé **async**. C'est ce simple détail qui fait de la fonction une coroutine.
Pour être plus précis, dans cet exemple, **main** est une **fonction** et l'**appel** de cette fonction **main()** va nous donner la **coroutine**.

```
>>> main
<function main at 0x7f030738ec80>
>>> main()
<coroutine object main at 0x7f03073e99c8>
```

© Editions ENI - All rights reserved

Nous allons maintenant essayer une coroutine avec des arguments :

```
>>> async def say_after(delay, what):
...     print(f"before {what} {time.strftime('%X')}")
...     await asyncio.sleep(delay)
...     print(f"after  {what} {time.strftime('%X')}")
...
```

Cet exemple, également adapté de la documentation officielle, montre qu'au niveau syntaxique il n'y a rien de surprenant. On rajoute les mots-clés **async** et **await**, sur lequel on donnera les explications plus tard, le premier devant la définition de la fonction pour en faire une coroutine et le second aux endroits où il faut attendre le résultat d'une fonctionnalité non bloquante.
Encore une fois, on peut constater que **say_after** est une fonction, mais qu'un appel à cette fonction ne va pas renvoyer un résultat, mais une coroutine :

```
>>> say_after
<function say_after at 0x7f03073491e0>
>>> say_after(1, "hello")
<coroutine object say_after at 0x7f03072e92c8>
```

Enfin, il faut savoir qu'une coroutine peut renvoyer un résultat :

```
>>> async def get_the_response():
...     return 42
...
```

Ce résultat peut être récupéré lorsque l'on appelle la coroutine (pas la fonction, mais bien la coroutine). Comme précédemment, l'appel de la fonction génère la coroutine :

```
>>> get_the_response()
<coroutine object get_the_response at 0x7f554ed157c8>
```

Maintenant que nous savons comment écrire des coroutines, voyons comment les exécuter.

2.2 Exécution d'une coroutine

On peut commencer par montrer ce que l'on ne peut pas faire. En effet, une coroutine n'est pas un callable :

```
>>> get_the_response()()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    get_the_response()()
TypeError: 'coroutine' object is not callable
```

Pour appeler une coroutine, il faut procéder ainsi :

```
>>> asyncio.run(get_the_response())
42
```

Lorsque l'on écrit du code, on écrit par défaut du code synchrone. C'est cette instruction qui va permettre d'appeler du code asynchrone et d'attendre que toutes les instructions de ce dernier soient terminées avant de reprendre la suite des instructions synchrones.

Bien évidemment, l'appel de fonction asynchrone, pour être efficace, devra faire plusieurs choses et tenter de les paralléliser, nous verrons comment plus tard.
Ce qui compte pour l'instant est de comprendre que c'est cette instruction **run** qui va permettre de lancer les tâches asynchrones et qu'à la fin de l'instruction, on revient en synchrone, c'est-à-dire à du code Python classique.
Testons maintenant avec la coroutine **main** que nous avons écrite plus haut :

```
>>> >>> asyncio.run(main())
hello 11:58:55
world 11:58:56
!!!!! 11:58:56
```

Si on exécute cela dans un terminal, on verra s'afficher la première ligne, puis une seconde après la deuxième ligne suivie immédiatement par la troisième ligne.
Il y a plusieurs choses à dire ici. Tout d'abord, on voit que l'argument passé à **run** est **main()** et non **main**, c'est-à-dire la **coroutine** et non la **fonction**.
Ensuite, on peut noter que l'appel à **async.sleep(1)** va faire attendre 1 seconde. Cette fonction a, du point de vue de la coroutine, un effet similaire à **time.sleep(1)**, sauf que cette dernière est et restera pour des raisons de compatibilité une fonction synchrone. C'est-à-dire que pendant l'exécution de cette fonction, rien d'autre ne peut être exécuté. Il s'agit donc d'une fonction bloquante et si on l'avait utilisée, créer une coroutine n'aurait servi à rien.
Par contre, la fonction **async.sleep(1)** est une fonctionnalité non bloquante. Il s'agit d'une coroutine que l'on exécute et dont on attend qu'elle soit terminée par l'utilisation du mot-clé **await** qui signifie littéralement **attendre** en anglais.
Ce dernier va à la fois **bloquer l'exécution de la coroutine courante**, en attendant la fin de l'appel, mais également permettre de **faire en sorte que, pendant ce temps, d'autres puissent être exécutées**.

Remarque

On ne peut pas prévoir quelles autres coroutines seront appelées, ni dans quel ordre, mais ce que l'on sait d'une manière certaine, c'est que notre coroutine ne reprendra qu'après que la coroutine **sleep** *ait terminé.*

2.3 Précisions sur le mot-clé await

Il est impossible d'utiliser **await** en dehors d'une fonction asynchrone :

```
>>> await get_the_response()
  File "<input>", line 1
SyntaxError: 'await' outside function
>>> def test():
...     await get_the_response()
  File "<input>", line 2
SyntaxError: 'await' outside async function
```

Si on appelle une fonction asynchrone sans l'attendre, il ne se passe rien. Voici un exemple de ce qu'il ne faut pas faire :

```
>>> async def main():
```

© Editions ENI - All rights reserved

```
...     print("test appel 1", get_the_response())
...     print("test appel 2", await get_the_response())
...
```

Dans le premier affichage, on n'utilise pas le mot-clé **await**, on va donc créer la coroutine et ne rien en faire.
Dans le second affichage, on utilise le mot-clé **await**, on va donc créer la coroutine et l'exécuter.
Au passage, le rajout du mot-clé **await** n'empêche pas de mettre cette instruction à l'intérieur d'un appel de fonction.

```
>>> asyncio.run(main())
test appel 1 <coroutine object get_the_response at 0x7f554eda3148>
RuntimeWarning: coroutine 'get_the_response' was never awaited
RuntimeWarning: Enable tracemalloc to get the object allocation traceback
test appel 2 42
```

Il est à noter que l'on voit l'affichage d'un avertissement nous expliquant que nous n'avons pas attendu une coroutine et qu'il y a probablement quelque chose à revoir dans notre code. Ce qui permet de détecter une anomalie qui pourrait facilement nous échapper.
Dans le cas où il serait difficile de localiser la source de l'erreur, un second message nous recommande l'utilisation de tracemalloc pour, si on le juge utile, avoir plus d'informations et détecter la source de l'erreur.
À noter que ces avertissements ne s'affichent qu'une seule fois :

```
>>> asyncio.run(main())
test appel 1 <coroutine object get_the_response at 0x7f554ed0e948>
test appel 2 42
```

Remarque

Sans aller trop loin dans les détails de l'implémentation, il faut savoir que l'asynchrone repose sur une boucle événementielle et que c'est le langage Python qui va ordonner les différentes coroutines pour gérer laquelle exécuter et quand passer de l'une à l'autre.

Remarque

Le but de cette organisation est que le programme ne reste jamais inactif et aille chercher parmi les coroutines en cours d'exécution celle qu'il peut faire avancer.

Regardons maintenant comment exécuter plusieurs coroutines.

2.4 Exécuter plusieurs coroutines en séries

Lorsque l'on a un code qui a des parties bloquantes, nous avons vu qu'il faut identifier ces parties, les isoler en créant des coroutines utilisant des fonctionnalités non bloquantes. Ce que nous souhaitons faire est d'exécuter plusieurs coroutines en même temps pour faire en sorte qu'il y en ait toujours au moins une qui tourne.

Pour cela, on va écrire une coroutine **main** qui va elle-même appeler nos coroutines qui exécutent les tâches qui nous intéressent et c'est cette dernière coroutine **main** que l'on va passer à la fonction **run**.

Dans la coroutine **main**, écrite précédemment, nous avions utilisé le mot-clé **await** pour appeler une coroutine **asyncio.sleep**. Dans cet exemple, nous allons écrire une coroutine **main** qui ira appeler plusieurs coroutines **say_after** en utilisant plusieurs instructions **await**. Ces coroutines sont exécutées en séries, c'est-à-dire l'une après l'autre.

Voici un exemple appelant la coroutine **say_after** déclarée précédemment :

```
>>> async def main_say_after():
...     print(f"started at {time.strftime('%X')}")
...     await say_after(1, 'hello')
...     await say_after(2, 'world')
...     await say_after(1, '!!!!!!')
...     print(f"finished at {time.strftime('%X')}")
...
```

Voici maintenant le résultat de l'exécution de cette coroutine :

```
>>> asyncio.run(main_say_after())
started at 09:46:00
before hello 09:46:00
after  hello 09:46:01
before world 09:46:01
after  world 09:46:03
before !!!!! 09:46:03
after  !!!!! 09:46:04
finished at 09:46:04
```

Comme on peut le voir, le mot-clé **await** signifie réellement **attendre**. Une coroutine n'est exécutée que lorsque la précédente est terminée. Ce code prend donc 4 secondes à s'exécuter.

S'il est important de passer par là pour la compréhension, ce n'est pas forcément ce que l'on veut voir, puisque l'intérêt d'utiliser des coroutines est de les exécuter en parallèle.

2.5 Tâches asynchrones

Il est possible de paralléliser l'exécution des coroutines par la création de tâches asynchrones :

```
>>> async def main_say_after():
...     task1 = asyncio.create_task(
...         say_after(1, 'hello'))
...
...     task2 = asyncio.create_task(
...         say_after(2, 'world'))
...
...     task3 = asyncio.create_task(
...         say_after(1, '!!!!!!'))
...
...     print(f"started at {time.strftime('%X')}")
```

© Editions ENI - All rights reserved

```
...
...     print(f"await task 1 {time.strftime('%X')}")
...     await task1
...     print(f"await task 2 {time.strftime('%X')}")
...     await task2
...     print(f"await task 3 {time.strftime('%X')}")
...     await task3
...
...     print(f"finished at {time.strftime('%X')}")
...
```

Si on fait abstraction des nombreux affichages présents dans ce code pour illustrer le propos, le principe est relativement simple : on crée une tâche pour chaque appel coroutine à exécuter (soit chaque appel de fonction asynchrone), puis on lance les tâches et on attend que chaque tâche soit terminée.
Le point clé est que les tâches sont toutes lancées en même temps par le premier **await**.
Voyons voir ce que cela donne en exécutant cette coroutine :

```
>>> asyncio.run(main_say_after())
started at 10:19:26
await task 1 10:19:26
before hello 10:19:26
before world 10:19:26
before !!!!! 10:19:26
after  hello 10:19:27
await task 2 10:19:27
after  !!!!! 10:19:27
after  world 10:19:28
await task 3 10:19:28
finished at 10:19:28
```

Si on analyse la première partie, on voit que rien ne se passe avant le premier **await**.
Ensuite, toutes les tâches sont lancées ensemble. Puis, la première tâche se termine et on passe à l'instruction **await** suivante, qui attend la tâche 2. Or, la tâche 3 se termine, puis la tâche 2. On passe alors à l'instruction qui attend la tâche 3, mais comme elle est déjà terminée, on passe immédiatement à l'instruction suivante.
Ce que l'on peut retenir de ce code est que pour paralléliser des traitements, il faut créer des tâches asynchrones puis les attendre toutes. L'ordre d'attente importe peu, car au final, si la tâche est déjà terminée, l'instruction ne prend qu'un temps infiniment court, puisqu'elle n'a rien à faire.

Remarque

Si on oublie d'attendre l'une des tâches, alors elle est bien lancée, mais ne se termine pas forcément. Si elle dure moins longtemps que la plus longue qui est attendue, alors on verra l'affichage. Sinon, on ne verra pas l'affichage.

2.6 Exécuter les tâches asynchrones de manière concurrente

Maintenant que nous avons vu comment créer manuellement des tâches et les attendre, il est important de savoir que Python propose une méthode de haut niveau pour nous économiser un peu de travail. En effet, rien que pour trois tâches, il y avait déjà trop de répétitions de code quasi identique et le pythoniste n'aime pas cela.
Voici un code qui fait exactement la même chose que le code précédent :

```
>>> async def main_say_after():
...     print(f"started at {time.strftime('%X')}")
...     await asyncio.gather(
...         say_after(1, 'hello'),
...         say_after(2, 'world'),
...         say_after(1, '!!!!!!'),
...     )
...     print(f"finished at {time.strftime('%X')}")
...
```

Cette technique est à la fois élégante et suffisamment claire pour que vous deviniez, après avoir lu ce qui précède, les avantages de cette technique.
En réalité, il n'est pas conseillé de créer soi-même des tâches, mais plutôt d'utiliser cette fonction.
Tous les arguments passés à la fonction **`gather`** doivent être de type awaitable.

2.7 Notion d'awaitable

Un objet awaitable est simplement un objet sur lequel on peut utiliser le mot-clé **`await`**.

```
>>> async def main():
...     await print("truc")
...
>>> asyncio.run(main())
truc
Traceback (most recent call last): [...]
TypeError: object NoneType can't be used in 'await' expression
```

Si on utilise mal cette instruction, on obtient une erreur de type TypeError qui est explicite.
Alors quels sont les objets awaitables ? On peut déjà répondre les coroutines, et les tâches asynchrones, puisqu'on vient de le voir. Il faut rajouter à cette liste les futures dont on parlera plus tard.

2.8 Précisions sur asyncio.sleep

La coroutine **`asyncio.sleep`** renvoie par défaut **`None`**, mais elle peut renvoyer, en fait, ce que l'on souhaite. Ainsi, nous pourrions remplacer avantageusement ceci :

```
>>> async def compute_value()
...     return 42
...
```

© Editions ENI - All rights reserved

par ceci :

```
>>> compute_value = asyncio.sleep(1, result=42)
```

Cette astuce permet de se faciliter la tâche lorsque l'on teste l'asynchrone.

3. Éléments de grammaire

3.1 Gestionnaire de contexte asynchrone

Le gestionnaire de contexte permet de gérer proprement une ressource :

```
>>> with open("path/to/file") as f:
...     content = f.read()
...
```

En procédant ainsi, on sait que quoi qu'il se passe, notre fichier sera bien fermé. Et ceci est applicable à n'importe quelle ressource, en particulier des ressources réseau.
Il en est de même pour le gestionnaire de contexte asynchrone, sauf que la ressource est traitée de manière asynchrone, par des outils asynchrones.
En d'autres termes, elle nécessite un traitement particulier qui est différent de celui du gestionnaire de contexte classique, mais au final, cela est relativement transparent pour le développeur qui n'a qu'à rajouter le mot-clé **async** devant le mot-clé **with** (voir le chapitre Déclarations pour plus de détails).
Voici un exemple avec le module **aiofiles** (https://github.com/Tinche/aiofiles) :

```
>>> async with aiofiles.open("path/to/file") as f:
...     content = await f.read()
...
```

Voici un exemple avec le module **aiohttp**, qui permet de gérer des ressources HTTP de manière asynchrone, comme son nom l'indique :

```
>>> import asyncio
>>> import aiohttp
>>> async def download_json():
...     loop = asyncio.get_running_loop()
...     async with aiohttp.ClientSession(loop=loop) as session:
...         async with session.get('http://site.com/file.json') as resp:
...             print(resp.status)
...             print(await resp.json())
...             print(data)
...
>>> asyncio.run(download_json())
```

3.2 Générateurs asynchrones

Un générateur asynchrone est un générateur qui permet de faire autre chose en attendant le prochain élément. Comme les générateurs standard, il renvoie un élément dès qu'il l'a calculé, évitant de devoir récupérer et stocker toute la liste pour permettre à celui qui l'a appelé d'avancer.

```
>>> async def gen(limit):
...     for i in range(limit):
...         yield i
...         asyncio.sleep(0.001)
...
```

Ces générateurs sont particulièrement utiles et sont utilisables uniquement dans une itération asynchrone.

3.3 Compréhensions asynchrones

Ils existent en compréhension :
Voici deux exemples tirés de la PEP 525 :

```
result = [i async for i in aiter() if i % 2]
result = [await fun() for fun in funcs if await condition()]
```

3.4 Itération asynchrone

L'itération asynchrone est un outil qui permet d'itérer sur un élément tout en permettant de faire autre chose si l'itérateur est en attente du prochain résultat.
Voici un exemple, en utilisant **`aiofiles`** :

```
>>> async with aiofiles.open("path/to/file") as f:
...     async for line in f:
...         print(line)
...
```

Ce type d'itération est à privilégier à chaque fois que vous travaillerez avec de multiples données.

4. Notions avancées

Remarque

Ce chapitre est destiné aux lecteurs avancés.

4.1 Introspection

Voyons dans un premier temps ce que nous pouvons trouver par nous-mêmes pour comprendre ce qu'est une tâche asynchrone :

```
>>> async def introspect():
...     task = asyncio.current_task()
...     print(task)
...     print(type(task), type.mro(type(task)))
```

© Editions ENI - All rights reserved

```
...     print(dir(task))
...
```

Pour récupérer la tâche asynchrone qui est en train d'être exécutée, on peut faire appel à **asyncio.current_task**. Bien évidemment, ceci nous renverra la coroutine **introspect**.
On la lance usuellement et on peut constater le premier point : la tâche courante, au sein de notre tâche, est bien notre tâche :

```
>>> asyncio.run(introspect())
<Task pending coro=<introspect()>
```

Ensuite, si nous regardons l'arbre d'héritage, nous voyons ceci :

```
[<class '_asyncio.Task'>, <class '_asyncio.Future'>, <class 'object'>]
```

Notre coroutine est un objet du type _asyncio.task et elle est un type particulier de futures, un des trois types d'awaitables (avec la tâche et la coroutine).
Voyons maintenant quels sont les attributs de la classe :

```
['__await__', '__class__', '__del__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__iter__', '__le__',
'__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'_asyncio_future_blocking', '_callbacks', '_coro', '_exception',
'_fut_waiter', '_log_destroy_pending', '_log_traceback', '_loop',
'_must_cancel', '_repr_info', '_result', '_source_traceback', '_state',
'add_done_callback', 'all_tasks', 'cancel', 'cancelled', 'current_task',
'done', 'exception', 'get_loop', 'get_stack', 'print_stack',
'remove_done_callback', 'result', 'set_exception', 'set_result']
```

Comme d'habitude, on se concentre sur les éléments non spéciaux et non privés. On oublie également **current_task** et **all_tasks**, car ils sont dépréciés et n'existeront plus en Python 3.9, remplacés respectivement par **asyncio.current_task** et **asyncio.all_tasks**. Nous allons présenter un cas d'utilisation pour chacun d'entre eux.
Commençons par montrer la manière de récupérer toutes les tâches :

```
>>> async def introspect():
...     for task in asyncio.all_tasks():
...         print(task)
...
>>> async def main_say_after():
...     print(f"started at {time.strftime('%X')}")
...     await asyncio.gather(
...         say_after(1, 'hello'),
...         say_after(2, 'world'),
...         introspect(),
...     )
...     print(f"finished at {time.strftime('%X')}")
...
>>> asyncio.run(main_say_after())
```

```
started at 15:47:29
before hello 15:47:29
before world 15:47:29
<Task pending coro=<say_after()>
<Task pending coro=<introspect()>
<Task pending coro=<main_say_after()>
<Task pending coro=<say_after()>
after  hello 15:47:30
after  world 15:47:31
finished at 15:47:31
```

On voit bien la tâche **introspect**, les deux **say_after** ainsi que la **main_say_after**. Ce que l'on ne met pas en évidence ici est qu'en réalité, **all_tasks** ne retourne que les tâches non terminées.
Enfin, pour chaque tâche, il est possible de savoir si elle est terminée ou non. Pour cela, on modifie la coroutine **introspect** de cette manière :

```
>>> async def introspect(delay=None):
...     if delay:
...         await asyncio.sleep(delay)
...     for task in asyncio.all_tasks():
...         print(task, task.done())
...
```

On va ensuite utiliser cette coroutine pour voir ce qu'il se passe à l'état initial, après que la première tâche **say_after** soit terminée puis à l'état final :

```
>>> async def main_say_after():
...     print(f"started at {time.strftime('%X')}")
...     await asyncio.gather(
...         say_after(1, 'hello'),
...         say_after(3, 'world'),
...         introspect(),
...         introspect(2),
...         introspect(4),
...     )
...     print(f"finished at {time.strftime('%X')}")
...
```

On peut maintenant analyser ce qu'il se passe à chaque étape :

```
>>> asyncio.run(main_say_after())
started at 16:07:40
before hello 16:07:40
before world 16:07:40
<Task pending coro=introspect()> False
<Task pending coro=introspect()> False
<Task pending coro=introspect()> False
<Task pending coro=main_say_after()> False
<Task pending coro=say_after()> False
<Task pending coro=say_after()> False
```

© Editions ENI - All rights reserved

Comme on peut s'y attendre, aucune tâche n'est terminée. Voici maintenant ce que l'on peut observer à mi-chemin :

```
after  hello 16:07:41
<Task pending coro=<say_after()> False
<Task pending coro=<introspect()> False
<Task pending coro=<main_say_after()> False
<Task pending coro=<introspect()> False
```

On voit que les tâches terminées ne sont tout simplement plus dans la liste des tâches récupérées via **all_tasks**.

Remarque

Ces fonctionnalités sont plutôt destinées à être utilisées avec des futures, des objets de plus bas niveau que les tâches. Il semblait cependant intéressant de montrer qu'elles étaient néanmoins utilisables dans ce contexte. On présentera à nouveau ces fonctionnalités dans la partie dédiée aux futures.

```
after  world 16:07:43
<Task pending coro=<introspect()> False
finished at 16:07:44
```

Ceci se confirme, puisqu'il ne reste ici que la tâche courante.
On peut cependant retourner à notre exemple bas niveau pour avoir une validation de la manière dont **done** fonctionne :

```
>>> async def main_say_after():
...     task1 = asyncio.create_task(
...         say_after(1, 'hello'))
...
...     task2 = asyncio.create_task(
...         say_after(4, 'world'))
...
...     task3 = asyncio.create_task(
...         say_after(2, '!!!!!!'))
...
...     print(f"started at {time.strftime('%X')}")
...
...     print(f"await task 1 {time.strftime('%X')}", task1.done(),
task2.done(), task3.done())
...     await task1
...     print(f"await task 2 {time.strftime('%X')}", task1.done(),
task2.done(), task3.done())
...     await task2
...     print(f"await task 3 {time.strftime('%X')}", task1.done(),
task2.done(), task3.done())
...     await task3
...
...     print(f"finished at {time.strftime('%X')}")
...
```

```
>>> asyncio.run(main_say_after())
started at 16:17:49
await task 1 16:17:49 False False False
before hello 16:17:49
before world 16:17:49
before !!!!! 16:17:49
after  hello 16:17:50
await task 2 16:17:50 True False False
after  !!!!! 16:17:51
after  world 16:17:53
await task 3 16:17:53 True True True
finished at 16:17:53
```

On voit qu'avant d'attendre la fin de la tâche 1, il n'y en a aucune de terminée. Avant la fin de la tâche 2, seule la première tâche est terminée et avant d'attendre la fin de la tâche 3, toutes les tâches sont en réalité terminées.

4.2 Gestion du résultat ou de l'exception

Chaque tâche, une fois terminée, se voit affecter son résultat :

```
>>> async def main():
...     task = asyncio.create_task(get_the_response())
...     print(task.done())
...     await task
...     print(task.done(), task.result())
...
>>> asyncio.run(main())
False
True 42
```

Il est assez rare que l'on ait à l'utiliser, mais ceci peut être utile à savoir dans certains contextes. De la même manière, une tâche se voit associer une exception. Si la coroutine se termine normalement, alors **`task.exception()`** renvoie **`None`**. Si la coroutine a levé une exception, alors **`task.exception()`** renvoie cette exception.

Au niveau de la gestion des exceptions, il faut savoir que par défaut toute exception se produisant dans une coroutine va se propager jusqu'au fil du programme principal :

```
>>> async def mistake(delay):
...     await asyncio.sleep(delay)
...     raise Exception("This is a mistake")
...
>>> async def main_say_after():
...     print(f"started at {time.strftime('%X')}")
...     await asyncio.gather(
...         say_after(1, 'hello'),
...         say_after(3, 'world'),
...         mistake(2),
...     )
...     print(f"finished at {time.strftime('%X')}")
...
```

© Editions ENI - All rights reserved

```
>>> asyncio.run(main_say_after())
started at 20:28:41
before hello 20:28:41
before world 20:28:41
after  hello 20:28:42
Traceback (most recent call last):
Exception: This is a mistake
```

Il est cependant possible de faire en sorte qu'une exception soit traitée comme un retour, de manière à ne pas impacter les autres tâches :

```
>>> async def main_say_after():
...     print(f"started at {time.strftime('%X')}")
...     await asyncio.gather(
...         say_after(1, 'hello'),
...         say_after(3, 'world'),
...         mistake(2),
...         return_exceptions=True,
...     )
...     print(f"finished at {time.strftime('%X')}")
...
>>> asyncio.run(main_say_after())
started at 20:30:28
before hello 20:30:28
before world 20:30:28
after  world 20:30:31
finished at 20:30:31
```

À ce moment-là, l'exception sera visible via la méthode **`exception`** de la tâche, précédemment exposée.

4.3 Annuler une tâche

Il est possible d'annuler une tâche. Pour illustrer ceci, on va reprendre l'exemple précédent dans lequel nous déclarions trois tâches et les attendions manuellement :

```
>>> async def main_say_after():
...     task1 = asyncio.create_task(
...         say_after(1, 'hello'))
...
...     task2 = asyncio.create_task(
...         say_after(4, 'world'))
...
...     task3 = asyncio.create_task(
...         say_after(2, '!!!!!!'))
...
...     print(f"started at {time.strftime('%X')}")
...
...     print(f"await task 1 {time.strftime('%X')}")
...     await task1
...     print(f"await 1 sec  {time.strftime('%X')}")
```

```
...     await asyncio.sleep(1)
...     task2.cancel()
...     print(f"await task 2 {time.strftime('%X')}")
...     try:
...         await task2
...     except asyncio.CancelledError:
...         print("You can't await a cancelled task")
...     print(f"await task 3 {time.strftime('%X')}")
...     await task3
...
...     print(f"finished at {time.strftime('%X')}")
...
```

Tout se déroule correctement jusqu'à l'annulation de la tâche 2. Ensuite, si on essaie d'attendre la tâche annulée, alors on obtient une exception ; cet exemple montre comment la capturer :

```
>>> asyncio.run(main_say_after())
started at 11:31:55
await task 1 11:31:55
before hello 11:31:55
before world 11:31:55
before !!!!! 11:31:55
after  hello 11:31:56
await 1 sec  11:31:56
after  !!!!! 11:31:57
await task 2 11:31:57
You can't await a cancelled task
await task 3 11:31:57
finished at 11:31:57
```

Pour la tâche 3, il n'y a pas de souci, le fait que la tâche 2 ait été annulée ne change rien à son comportement et le script se termine correctement.

Cependant, si on se rappelle bien, créer les tâches par soi-même n'est pas conseillé. Si on reprend l'exemple correct et que l'on modifie à la marge la coroutine **say_after**, on peut mettre en exergue le fait que si la coroutine **asyncio.gather** n'utilise pas **return_exceptions** à **True** et que l'exception d'annulation n'est pas capturée, alors cette exception se propage et entraîne l'annulation de toutes les tâches non terminées : l'annulation est répercutée : si une des tâches est annulée, alors toutes le sont par contamination :

```
>>> async def say_after(delay, what, cancel_after=False):
...     print(f"before {what} {time.strftime('%X')}")
...     await asyncio.sleep(delay)
...     print(f"after  {what} {time.strftime('%X')}")
...     if cancel_after:
...         task = asyncio.current_task()
...         task.cancel()
...         print(f"cancel {what} {time.strftime('%X')}")
...
>>> async def main_say_after():
```

© Editions ENI - All rights reserved

```
...     print(f"started at {time.strftime('%X')}")
...     await asyncio.gather(
...         say_after(1, 'hello'),
...         say_after(2, 'world'),
...         say_after(1, '!!!!!!', True),
...     )
...     print(f"finished at {time.strftime('%X')}")
...
>>> asyncio.run(main_say_after())
started at 15:36:14
before hello 15:36:14
before world 15:36:14
before !!!!! 15:36:14
after  hello 15:36:15
after  !!!!! 15:36:15
cancel !!!!! 15:36:15
Traceback (most recent call last):
concurrent.futures._base.CancelledError
```

On voit que l'annulation n'est pas immédiate, car le **print** après le **cancel** a le temps de se faire. C'est normal, car le processus d'annulation nécessite ce laps de temps.
De plus, on peut signaler qu'il existe une méthode de la tâche nommée **cancelled** qui permet de savoir si la tâche en question a été annulée ou non. Celle-ci est plutôt utile lorsque l'on manipule les futures autant que les tâches.
Par contre, si l'annulation est bien capturée ou si l'on utilise **return_exceptions** à **True**, alors l'annulation d'une tâche ne change rien pour les autres. Par contre, dans tous les cas, l'annulation de la coroutine qui contient l'appel à **asyncio.gather** entraîne l'annulation de toutes les coroutines passées en paramètre de ce **asyncio.gather**.

4.4 Se prémunir d'une annulation

Il est possible de se prémunir d'une annulation par la simple utilisation d'**asyncio.shield** :

```
>>> asyncio def will_be_cancelled():
...     try:
...         res = await asyncio.sheild(should_pursue())
...     except CancelledError:
...         res = None
...
```

Dans cet exemple, si la coroutine **will_be_cancelled** est annulée, la coroutine **should_pursue** continuera. Par contre, lorsqu'elle sera terminée, l'instruction **await** lancera une exception du type **CancelledError**. C'est pour cela que son code est dans une boucle d'exception.
Par contre, si la tâche **should_pursue** elle-même est annulée, elle est alors réellement annulée et **asyncio.shield** n'intervient pas.

4.5 Timeouts

On peut poser un timeout sur un appel asynchrone. Pour cela, il faut utiliser **asyncio.wait_for**. Si ce dernier ne répond pas à temps, alors une exception de type **asyncio.TimeoutError** sera lancée.
Voici un exemple qui vient directement de la documentation officielle :

```
>>> async def eternity():
...     # Sleep for one hour
...     await asyncio.sleep(3600)
...     print('yay!')
...
>>> async def main():
...     # Wait for at most 1 second
...     try:
...         await asyncio.wait_for(eternity(), timeout=1.0)
...     except asyncio.TimeoutError:
...         print('timeout!')
...
>>> asyncio.run(main())
timeout!
```

La tâche exécutée par **asyncio.wait_for** est alors annulée. Pour être exact, la tâche est annulée et **asyncio.wait_for** attend que cette annulation soit effective avant de lancer l'exception **asyncio.TimeoutError**.

Remarque

Comme pas mal de fonctionnalités présentées dans ce chapitre, wait_for est surtout conçue pour les futures, mais elle est particulièrement adaptée aux tâches également. C'est la solution idéale pour appliquer un timeout sur une coroutine.

4.6 Gestion globale fine de l'attente

On se place dans le cas où l'on a plusieurs tâches concurrentes et que l'on souhaite agir après que la première d'entre elles soit terminée (peu importe laquelle). Pour cela, on peut procéder ainsi :

```
>>> async def main_say_after():
...     task1 = asyncio.create_task(
...         say_after(1, 'hello'))
...
...     task2 = asyncio.create_task(
...         say_after(4, 'world'))
...
...     task3 = asyncio.create_task(
...         say_after(2, '!!!!!!'))
...
...     tasks = {task1, task2, task3}
...
...     print(f"started at {time.strftime('%X')}")
```

© Editions ENI - All rights reserved

```
...     done, pending = await asyncio.wait(tasks,
return_when=asyncio.FIRST_COMPLETED)
...     print(f"FIRST_COMPLETED: {done}\nNOT_YET_COMPLETED: {pending}")
...     done, pending = await asyncio.wait(tasks,
return_when=asyncio.ALL_COMPLETED)
...     print(f"ALL_COMPLETED: {done}\nNOT_YET_COMPLETED: {pending}")
...     print(f"finished at {time.strftime('%X')}")
...
>>> asyncio.run(main_say_after())
started at 18:04:47
before hello 18:04:47
before world 18:04:47
before !!!!! 18:04:47
after  hello 18:04:48
FIRST_COMPLETED: {<Task finished coro=<say_after() done>}
NOT_YET_COMPLETED: {<Task pending coro=<say_after() running>,
                    <Task pending coro=<say_after() running>}
after  !!!!! 18:04:49
after  world 18:04:51
ALL_COMPLETED: {<Task finished coro=<say_after() done>,
                <Task finished coro=<say_after() done>,
                <Task finished coro=<say_after() done>}
NOT_YET_COMPLETED: set()
finished at 18:04:51
```

Cette technique permet ainsi de n'attendre que la première tâche. On peut également intervenir après la fin de toutes les tâches ou encore après la première exception, en utilisant **`asyncio.FIRST_EXCEPTION`**.

Enfin, sachez qu'il est aussi possible de positionner un **timeout** dans cette fonction.

Par contre, en utilisant **`wait`**, on ne peut cependant pas intervenir après la fin de chaque tâche et gérer le résultat des tâches au fur et à mesure. Si on souhaite faire ceci, il faut procéder autrement :

```
>>> async def say_after(delay, what):
...     print(f"before {what} {time.strftime('%X')}")
...     await asyncio.sleep(delay)
...     print(f"after {what} {time.strftime('%X')}")
...     return delay
...
>>> async def main_say_after():
...     task1 = asyncio.create_task(
...         say_after(1, 'hello'))
...
...     task2 = asyncio.create_task(
...         say_after(4, 'world'))
...
...     task3 = asyncio.create_task(
...         say_after(2, '!!!!!'))
...
```

```
...     tasks = {task1, task2, task3}
...
...     print(f"started at {time.strftime('%X')}")
...     for task in asyncio.as_completed(tasks):
...         result = await task
...         print(f"task completes: {task}, result={result}")
...     print(f"finished at {time.strftime('%X')}")
...
>>> asyncio.run(main_say_after())
started at 11:52:29
before hello 11:52:29
before world 11:52:29
before !!!!! 11:52:29
after hello 11:52:30
task completes: <coroutine object as_completed>, result=1
after !!!!! 11:52:31
task completes: <coroutine object as_completed>, result=2
after world 11:52:33
task completes: <coroutine object as_completed>, result=4
finished at 11:52:33
```

À noter que pour avoir un résultat, nous avons modifié pour cet exemple notre fonction asynchrone de base **say_after**. Cette technique peut aussi être utilisée dans une compréhension de liste :

```
>>> async def main_say_after():
...     task1 = asyncio.create_task(
...         say_after(1, 'hello'))
...
...     task2 = asyncio.create_task(
...         say_after(4, 'world'))
...
...     task3 = asyncio.create_task(
...         say_after(2, '!!!!!!'))
...
...     tasks = {task1, task2, task3}
...
...     print(f"started at {time.strftime('%X')}")
...     results = [await task for task in asyncio.as_completed(tasks)]
...     print(f"results: {results}")
...     print(f"finished at {time.strftime('%X')}")
...
>>> asyncio.run(main_say_after())
started at 11:57:55
before hello 11:57:55
before world 11:57:55
before !!!!! 11:57:55
after hello 11:57:56
after !!!!! 11:57:57
```

© Editions ENI - All rights reserved

```
after world 11:57:59
results: [1, 5, 1]
finished at 11:57:59
```

Cette technique est relativement simple à mettre en œuvre, mais en utilisant ici une compréhension de liste, on perd un peu l'intérêt de l'utilisation de **as_completed** si ce n'est que l'on conserve l'ordre d'arrivée des résultats.

4.7 Module contextvars

Une des nouveautés de Python 3.7 est le module **contextvars**. Pour l'instant assez limité dans son utilisation, il est possible que son usage se répande à d'autres fonctionnalités asynchrones dans les versions futures.
Un contexte est un conteneur de variables et l'idée derrière ce module est de permettre de partager ce contexte entre différentes fonctions sans que l'on soit obligé de passer des variables ou des dictionnaires de variables de fonction en fonction.
Commençons par la notion de variable :

```
>>> from contextvars import ContextVar
>>> nb = ContextVar("How many ?", default=0)
>>> nb.set(0)
```

On vient de créer une nouvelle variable, en mettant une description et une valeur par défaut. À partir de ce moment-là, on peut créer un contexte :

```
>>> from contextvars import copy_context
>>> ctx = copy_context()
>>> print(list(ctx.items()))
[(<ContextVar name='How many ?' default=0 at 0x7fa3ac2cc048>, 0)]
```

Il est alors possible de lire la valeur de notre variable de deux manières différentes :

```
>>> print(nb.get(), ctx[nb])
0 0
```

Le contexte contient notre variable. Par contre, si la variable n'est pas mise à jour avec **set**, alors le contexte ne contient pas cette variable, même si elle a une valeur par défaut :

```
>>> from contextvars import ContextVar, copy_context
>>> nb = ContextVar("How many ?", default=0)
>>> ctx = copy_context()
>>> print(list(ctx.items()))
[]
>>> nb.set(0)
>>> print(list(ctx.items()))
[]
```

Il est possible de faire suivre l'évolution des variables et de revenir à une valeur précédente :

```
>>> nb.set(0)
>>> token_1 = nb.set(1)
>>> token_2 = nb.set(2)
```

Le token est un objet qui représente une transition entre une valeur et la suivante. Ce token a deux attributs :

```
>>> token_1.var
<ContextVar name='How many ?' default=0 at 0x7f1204cbb0f8>
>>> token_1.old_value
0
```

Ce token permet de revenir vers une ancienne valeur :

```
>>> nb.reset(token_1)
>>> nb.get()
0
```

On peut le faire quel que soit l'ordre d'appel des tokens :

```
>>> nb.reset(token_2)
>>> nb.get()
1
```

Par contre, chaque token ne peut être appelé qu'une seule fois :

```
>>> nb.reset(token_2)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    nb.reset(token_2)
RuntimeError: <Token used var=<ContextVar name='How many ?' default=0 at
0x7f1204cbb0f8> at 0
x7f1204d14870> has already been used once
```

Voici maintenant un exemple complet utilisant ces fonctionnalités :

```
import asyncio
import logging
import coloredlogs
import sys
import signal
from functools import partial
from contextvars import ContextVar, copy_context


main_logger = logging.getLogger("main")
logger_format = "%(asctime)s,%(msecs)03d %(levelname)s %(message)s"
coloredlogs.install(fmt=logger_format, level="DEBUG",
logger=main_logger)


nb = ContextVar("How many ?", default=0)
nb.set(0)
ctx = copy_context()


async def aside_value():
```

© Editions ENI - All rights reserved

```
    main_logger.info(f"aside_value starting with    {nb.get()!r}")
    nb.set(await asyncio.sleep(1, result=42))
    main_logger.info(f"aside_value ending with      {nb.get()!r}")

def used_value():
    main_logger.info(f"used_value starting with     {nb.get()!r}")
    nb.set(10)
    main_logger.info(f"used_value ending with       {nb.get()!r}")

async def testing_tokens():
    await asyncio.sleep(0.1)
    main_logger.info(f"using token starting with: {nb.get()!r}")
    token_1 = nb.set(await asyncio.sleep(0.01, result=1))
    main_logger.info(f"token 1: from {token_1.old_value} > {nb.get()!r}")
    token_2 = nb.set(await asyncio.sleep(0.01, result=2))
    main_logger.info(f"token 2: from {token_2.old_value} > {nb.get()!r}")
    token_3 = nb.set(await asyncio.sleep(0.01, result=3))
    main_logger.info(f"token 3: from {token_3.old_value} > {nb.get()!r}")
    nb.reset(token_3)
    main_logger.info(f"using token 3: {nb.get()!r}")
    nb.reset(token_1)
    main_logger.info(f"using token 1: {nb.get()!r}")
    nb.reset(token_2)
    main_logger.info(f"using token 1: {nb.get()!r}")
    nb.reset(token_1)  # this will cause RuntimeError

def print_callback(crut):
    main_logger.info(f"Callback said               {nb.get()!r}")


async def main():
    event_loop = asyncio.get_running_loop()
    task = asyncio.create_task(aside_value())
    task.add_done_callback(print_callback, context=ctx)

    ctx.run(used_value)
    await task

    task = asyncio.create_task(testing_tokens())
    await task


if __name__ == "__main__":
    asyncio.run(main())
```

Voici maintenant ce que cela donne lorsqu'on exécute ce programme :

```
$ python test_context_vars.py
2019-03-08 16:25:11,385 INFO used_value starting with    0
2019-03-08 16:25:11,385 INFO used_value ending with      10
2019-03-08 16:25:11,385 INFO aside_value starting with   0
2019-03-08 16:25:12,386 INFO aside_value ending with     42
2019-03-08 16:25:12,387 INFO Callback said               10
2019-03-08 16:25:12,488 INFO using token starting with: 0
2019-03-08 16:25:12,499 INFO token 1: from 0 > 1
2019-03-08 16:25:12,510 INFO token 2: from 1 > 2
2019-03-08 16:25:12,521 INFO token 3: from 2 > 3
2019-03-08 16:25:12,521 INFO using token 3: 2
2019-03-08 16:25:12,521 INFO using token 1: 0
2019-03-08 16:25:12,522 INFO using token 1: 1
Traceback (most recent call last):
[...]
RuntimeError: <Token used var=<ContextVar name='How many ?' default=0 at
0x7f13882bc678> at 0x7f1387833dc8> has already been used once
```

Comme on peut le voir, la tâche ne partage pas le contexte. Elle a donc son propre contexte et ne partage pas la valeur des variables avec le flux principal.

Remarque

Ce script est disponible dans asyncio/test_context_vars.py

À noter la fonction **`ctx.run`** qui permet de faire tourner une fonction sans que l'on puisse passer de paramètres (sauf utilisation de **`functools.partial`**, comme déjà vu).

5. Boucle événementielle asynchrone

5.1 Gestion de la boucle

Toute tâche asynchrone est exécutée par ce que l'on nomme la boucle événementielle. Si la programmation parallèle repose sur la gestion des fils d'exécution ou des processus réalisée par la machine virtuelle et par le système, la programmation asynchrone repose sur la boucle événementielle fournie par **`asyncio`** pour gérer de la manière la plus efficace possible la gestion des I/O, des événements système et des changements de contextes applicatifs.

L'application que l'on développe va interagir avec la boucle événementielle pour enregistrer du code à exécuter et laisser prendre les décisions appropriées pour ordonner l'exécution de ce code aux moments les plus appropriés, lorsque les ressources demandées sont accessibles. Chaque tâche, une fois qu'on lui a donné la main, va pouvoir avancer puis rendre la main à la boucle dès qu'elle se met en attente.

Dans tout ce que l'on a vu jusqu'à présent, la présence de la boucle événementielle est totalement invisible dans le code, mais à chaque fois que l'on utilise le mot-clé **`await`**, on retourne dans cette boucle et celle-ci va choisir une nouvelle tâche asynchrone à exécuter.

© Editions ENI - All rights reserved

Pour obtenir la boucle asynchrone courante, on peut faire :

```
>>> asyncio.get_running_loop()
```

Attention, la boucle courante n'est disponible que lorsque l'on est dans une fonction asynchrone, exécutée par exemple par **`asyncio.run`**.
S'il n'y a pas de boucle asynchrone, alors on obtient une exception :

```
>>> asyncio.get_running_loop()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    asyncio.get_running_loop()
RuntimeError: no running event loop
```

Cette boucle courante est à utiliser de manière préférentielle, elle est gérée automatiquement par **`asyncio`**.
Sachez qu'il est aussi possible d'utiliser :

```
>>> loop = asyncio.get_event_loop()
```

Cette dernière commande peut fonctionner en dehors d'une fonction asynchrone et est très utilisée. Vous la verrez donc souvent dans les exemples.
Il est possible de créer une nouvelle boucle :

```
>>> loop = asyncio.new_event_loop()
```

Et on peut déclarer cette boucle comme étant la boucle courante ainsi :

```
>>> asyncio.set_event_loop(loop)
```

À noter que si l'on ne souhaite plus avoir de boucle courante, on peut procéder ainsi :

```
>>> asyncio.set_event_loop(None)
```

On peut, si on le souhaite, utiliser plusieurs boucles asynchrones, dans des cas très particuliers où l'on voudrait séparer des tâches, mais ce n'est pas quelque chose de répandu. Par défaut, la boucle asynchrone fonctionne selon une politique prédéfinie qui est explicitée dans la classe **`asyncio.DefaultEventLoopPolicy`**.
On peut également créer sa propre politique en créant sa propre classe qui héritera de **`asyncio.AbstractEventLoopPolicy`** (https://docs.python.org/3/library/asyncio-policy.html#asyncio-policies). Ce sont là des concepts très avancés que l'on a rarement besoin de manipuler.

5.2 Débogage

Avant de commencer, il paraît important de vous donner les clés essentielles sur les moyens que vous pouvez mettre en œuvre pour vous aider à détecter les erreurs dues à une mauvaise gestion de l'asynchrone.
Tout d'abord, la boucle événementielle a son propre système de log qui vous permet de voir ce qu'il se passe en arrière-plan et surtout de détecter si quelque chose se passe mal. Je vous conseille de l'activer sur des scripts que vous maîtrisez pour voir ce qu'il se passe et prendre en main ces messages avant de l'utiliser pour déboguer un script qui poserait problème.

Pour l'activer :

```
>>> event_loop.set_debig(True)
```

Avec ceci, il faudra aussi configurer le logging :

```
logging.basicConfig(
    level=logging.DEBUG,
    format="%(levelname)7s: %(message)s",
    stream=sys.stderr,
)
```

Bien entendu, si on utilise **False**, on désactive ces messages d'informations (qui sont désactivés par défaut) pour ne les activer que lorsque l'on en a besoin et éviter d'être envahi par une trop grande quantité de messages.

Lorsque l'application quitte, il est possible que certaines coroutines, futures ou tâches puissent être encore en suspend. L'application ne quitte donc pas proprement et cela peut signifier que des ressources utilisées par ces coroutines pourraient être mal fermées et ainsi se retrouver dans un état instable.

Pour prévenir cela, on peut écrire ce qui suit :

```
>>> warnings.simplefilter('always', ResourceWarning)
```

Asyncio utilise également **RuntimeWarning**, mais ces derniers sont automatiquement affichés par défaut.

Ce code permet de reporter toutes les parties du code qui ne se terminent pas proprement.

Enfin, la problématique principale de la programmation asynchrone est l'utilisation d'un code qui ne l'est pas. Pour détecter ces bouts de code, on peut faire ceci :

```
>>> event_loop.slow_callback_duration = 0.001
```

Cette instruction permet de spécifier que l'on considère un callback comme étant lent dès qu'il dépasse une milliseconde (par défaut, c'est un dixième de seconde). Cette instruction n'est utile que si l'on a configuré le debug.

Remarque

Il est livré un fichier asyncio/test_debug qui permet de tester différentes variations de ces choix de configuration et observer leur effet. Ci-dessous un exemple d'utilisation.

```
$ python test_debug.py -di -s 0.001
2019-03-07 13:02:34,671 Using selector: EpollSelector
2019-03-07 13:02:34,671 Python 3.7 style
2019-03-07 13:02:34,671 debug is enabled
2019-03-07 13:02:34,671 warning disabled
2019-03-07 13:02:34,671 setting slow_callback_duration to 0.001
2019-03-07 13:02:34,671 main 3.7-style coroutine starting
2019-03-07 13:02:34,672 task very slow starting
2019-03-07 13:02:34,773 task very slow completed
2019-03-07 13:02:34,773 Executing <Task finished coro=<coroutine_ve-
ry_slow() done, defined at test_debug.py:30> result=None created at /usr/
lib/python3.7/asyncio/tasks.py:325> took 0.101 seconds
2019-03-07 13:02:34,774 task quite slow starting
```

© Editions ENI - All rights reserved

```
2019-03-07 13:02:34,864 task quite slow completed
2019-03-07 13:02:34,865 Executing <Task finished coro=<corou-
tine_quite_slow() done, defined at test_debug.py:25> result=None created
at /usr/lib/python3.7/asyncio/tasks.py:325> took 0.091 seconds
2019-03-07 13:02:34,865 main 3.7-style coroutine completed
2019-03-07 13:02:34,866 Close <_UnixSelectorEventLoop running=False
closed=False debug=True>
```

5.3 Notion de future

La notion de future fait référence à une technique de synchronisation utilisée par les langages concurrents. Il s'agit d'une structure de donnée qui permet de représenter le résultat issu d'une tâche non encore réalisée permettant à une partie de l'application d'attendre qu'une autre partie ait terminé son travail.

Comme on l'a déjà dit, ce concept a été introduit initialement dans le langage Python avec **concurrent.futures** et est décliné maintenant avec l'asynchrone. Le principe consiste à permettre d'agir sur un objet qui n'est pas encore calculé.

Tout ce que l'on a vu jusqu'à présent est une manière de faire de l'asynchrone qualifiée de haut niveau. La notion de future est utilisée en sous-main, mais nous n'y touchons pas directement : on travaille sur des coroutines et on crée des tâches.

Mais, si on le souhaite, on peut utiliser directement les futures. En effet, il existe une autre manière de faire de l'asynchrone, plus bas niveau, mais parfois relativement pratique dans sa mise en œuvre, impliquant la manipulation des futures.

Contrairement à ce que l'on a vu précédemment, il ne s'agit plus seulement de créer quelques coroutines par l'utilisation des mots-clés **async** et **await** puis de les exécuter, mais de vraiment utiliser de l'algorithmique spécifique à l'asynchrone.

Les futures asynchrones sont conçues à l'image du module **concurrent.futures** qui est plus ancien et très apprécié des développeurs. Il s'agit d'un pont entre un code bas niveau basé sur les **fonctions de retour (callback)**, la base de la programmation événementielle et le mécanisme de haut niveau **async/await**.

Précisons tout de suite que **concurrent.futures.Future** et **asyncio.Future** ne sont pas compatibles et pas faits pour être interchangeables, la première n'étant pas awaitable.

Si leur manière de fonctionner est volontairement similaire, les éléments gérés sont totalement différents. Par contre, dans certains cas, on peut utiliser les deux ensemble.

Il est à nouveau temps de faire un peu d'introspection :

```
>>> async def main():
...     loop = asyncio.get_running_loop()
...     future = loop.create_future()
...     print(future, type.mro(type(future)))
...     print(dir(future))
...
>>> asyncio.run(main())
<Future pending> [<class '_asyncio.Future'>, <class 'object'>]
['__await__', '__class__', '__del__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
```

```
'__hash__', '__init__', '__init_subclass__', '__iter__', '__le__',
'__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '_asyncio_fu-
ture_blocking', '_callbacks', '_exception', '_log_traceback', '_loop',
'_repr_info', '_result', '_source_traceback', '_state', 'add_done_callba-
ck', 'cancel', 'cancelled', 'done', 'exception', 'get_loop', 'remo-
ve_done_callback', 'result', 'set_exception', 'set_result']
```

Voici un exemple d'utilisation d'une future. Tout d'abord, on crée la fonction asynchrone qui simule une attente et renvoie une valeur. Cette fonction prend en paramètre une future et utilise **`future.set_result`** pour positionner le résultat.

```
>>> from random import randint
>>> async def compute_value(future):
...     await asyncio.sleep(1)
...     value = randint(100, 999)
...     future.set_result(value)
...
```

Nous allons ensuite utiliser une fonction asynchrone pour :
- récupérer la boucle courante,
- créer une future,
- créer une tâche (la fonction asynchrone précédente),
- attendre notre tâche et afficher son résultat.

Voici ce que cela donne :

```
>>> async def main():
...     loop = asyncio.get_running_loop()
...     future = loop.create_future()
...     loop.create_task(compute_value(future))
...     print(await future)
...
>>> asyncio.run(main())
437
```

Lorsque l'on travaille à bas niveau, il s'agit ici de la manière recommandée par la documentation pour créer une future et une tâche. Si vous lisez d'autres documentations que la documentation officielle, vous pourriez trouver d'autres solutions, mais c'est parce que le module **`asyncio`** évolue très rapidement et que les recommandations associées suivent le rythme. D'autre part, il n'est pas exclu que les futures versions de Python introduisent également des nouveautés.

Dans cet algorithme, la future et la tâche sont liées. Le fait d'utiliser **`future.set_result`** va faire passer la future au statut terminé :

```
>>> async def main():
...     loop = asyncio.get_running_loop()
...     future = loop.create_future()
...     task = loop.create_task(compute_value(future))
...     print(f"future: done={future.done()}")
...     print(f"task : done={task.done()}")
```

© Editions ENI - All rights reserved

```
...     print(await future)
...     print(f"Future: done={future.done()}, result={future.result()}")
...     print(f"task : done={task.done()}")
...
>>> asyncio.run(main())
future: done=False
task : done=False
286
Future: done=True, result=286
task : done=True
```

On voit ici comment récupérer le statut terminé de la future et de la tâche, comment ce statut évolue, mais aussi comment récupérer le résultat.

5.4 Annulation

Remarque

Ce chapitre est dédié aux lecteurs avancés. Pour des cas pratiques concrets, vous pouvez aller directement à la prochaine section.

Il est possible d'annuler une future. Dans ce cas, la future passe au statut terminé et les tâches liées sont annulées :

```
>>> async def main():
...     loop = asyncio.get_running_loop()
...     future = loop.create_future()
...     task = loop.create_task(compute_value(future))
...     print(f"future: done={future.done()},
cancelled={future.cancelled()}")
...     print(f"task : done={task.done()}, cancelled={task.cancelled()}")
...     print("Annulation")
...     future.cancel()
...     print(f"Future: done={future.done()},
cancelled={future.cancelled()}")
...     print(f"task : done={task.done()}, cancelled={task.cancelled()}")
...
>>> asyncio.run(main())
future: done=False, cancelled=False
task : done=False, cancelled=False
Annulation
Future: done=True, cancelled=True
task : done=False, cancelled=False
```

Il est aussi possible d'annuler une tâche :

```
>>> async def main():
...     loop = asyncio.get_running_loop()
...     future = loop.create_future()
...     task = loop.create_task(compute_value(future))
...     print(f"future: done={future.done()},
```

```
cancelled={future.cancelled()}")
...     print(f"task : done={task.done()}, cancelled={task.cancelled()}")
...     print("Annulation")
...     task.cancel()
...     await asyncio.sleep(1)
...     print(f"Future: done={future.done()},
cancelled={future.cancelled()}")
...     print(f"task : done={task.done()}, cancelled={task.cancelled()}")
...
>>> asyncio.run(main())
future: done=False, cancelled=False
task : done=False, cancelled=False
Annulation
Future: done=False, cancelled=False
task : done=True, cancelled=True
```

On notera qu'il faut laisser un peu de temps à la tâche pour qu'elle puisse être annulée, l'effet n'est pas immédiat.
On notera aussi que la méthode **cancelled** permet d'obtenir l'état d'annulation de la future ou de la tâche.

5.5 Gestion des exceptions

Remarque

Ce chapitre est dédié aux lecteurs avancés. Pour des cas pratiques concrets, vous pouvez vous rendre directement à la prochaine section.

Il est possible qu'une exception se produise dans une coroutine, que cette exception soit volontaire (un **raise**) ou qu'elle soit due à quelque chose de non prévu ou à une erreur de programmation.
Voici un exemple de coroutine qui lève une exception :

```
>>> async def mistake(future):
...     await asyncio.sleep(1)
...     future.set_result(42)
...     raise Exception("This is a mistake!")
...
```

Et voici un exemple d'erreur de programmation non gérée :

```
>>> async def unhandle_mistake(future):
...     await asyncio.sleep(1)
...     l = [42]
...     future.set_result(l[0])
...     print(l[1])  # This should raise an exception !
...
```

Pour ce qui suit, chacune de ces deux fonctions d'erreur générerait le même résultat.
Commençons par faire comme tout à l'heure, sans gérer l'exception :

```
>>> async def main():
```

© Editions ENI - All rights reserved

```
...     loop = asyncio.get_running_loop()
...     future = loop.create_future()
...     task = loop.create_task(mistake(future))
...     print(f"future: done={future.done()}")
...     print(f"task : done={task.done()}")
...     print(await future)
...     print(f"Future: done={future.done()}, result={future.result()},
exception={future.exception()}")
...     print(f"task : done={task.done()}")
...     print("---")
...
>>> asyncio.run(main())
future: done=False
task : done=False
42
Future: done=True, result=42, exception=None
task : done=True
```

Jusqu'à présent, rien de nouveau. On voit bien que tout se déroule correctement, et qu'à la fin, la future et la tâche sont bien terminées. Par contre, juste en dessous, on voit ce message :

```
Task exception was never retrieved
future: <Task finished coro=<mistake() done, defined at <bpython-input-
1254>:1> exception=Exception('This is a mistake!')>
Traceback (most recent call last):
  File "<bpython-input-1254>", line 4, in mistake
    raise Exception("This is a mistake!")
Exception: This is a mistake!
```

Python nous informe qu'il s'est produit une exception qui n'a pas été gérée.
Voyons comment nous pouvons la gérer :

```
>>> async def main():
...     loop = asyncio.get_running_loop()
...     future = loop.create_future()
...     task = loop.create_task(mistake(future))
...     print(f"future: done={future.done()}")
...     print(f"task : done={task.done()}")
...     print(await future)
...     print(f"Future: done={future.done()}, result={future.result()},
exception={future.exception()}")
...     print(f"task : done={task.done()}")
...     print(f'The coroutine raised an exception: {task.exception()!r}')
...     print(task.result())  # Va produire l'exception
...
>>> asyncio.run(main())
future: done=False
task : done=False
42
```

```
Future: done=True, result=42, exception=None
task : done=True
The coroutine raised an exception: Exception('This is a mistake!')
```

On voit ici le statut de la future, celui de la tâche et on peut récupérer l'exception et l'afficher. Par contre, s'il s'est produit une exception, alors il est impossible d'accéder au résultat. Sinon, l'exception est lancée :

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    asyncio.run(main())
  File "/usr/lib/python3.7/asyncio/runners.py", line 43, in run
    return loop.run_until_complete(main)
  File "/usr/lib/python3.7/asyncio/base_events.py", line 584, in
run_until_complete
    return future.result()
  File "<input>", line 11, in main
    print(task.result())
  File "<input>", line 4, in mistake
    raise Exception("This is a mistake!")
Exception: This is a mistake!
```

Sachant tout cela, voici une proposition pour exécuter du code et relancer l'exception si elle s'est produite :

```
>>> async def main():
...     loop = asyncio.get_running_loop()
...     future = loop.create_future()
...     task = loop.create_task(unhandle_mistake(future))
...     result = await future
...     task.result()
...     print(result)
...
```

Soit l'appel **`task.result()`** ne va rien faire, soit il va relancer l'exception :

```
>>> asyncio.run(main())
Traceback (most recent call last):
IndexError: list index out of range
```

Le traceback a ici été coupé, mais on voit le résultat.
Voici maintenant ce qu'il se passe lorsqu'il n'y a pas d'exception, comme c'est le cas avec ce code :

```
>>> async def main():
...     loop = asyncio.get_running_loop()
...     future = loop.create_future()
...     task = loop.create_task(compute_value(future))
...     result = await future
...     task.result()
...     print(result)
...
```

© Editions ENI - All rights reserved

L'appel de la coroutine va simplement renvoyer le résultat :

```
>>> asyncio.run(main())
42
```

Voyons maintenant comment il est possible de gérer l'erreur et de simplement afficher un message lorsqu'elle se produit, sans utiliser une boucle d'exception :

```
>>> async def main():
...     loop = asyncio.get_running_loop()
...     future = loop.create_future()
...     task = loop.create_task(unhandle_mistake(future))
...     await future
...     exception = task.exception()
...     if exception:
...         print(f"The coroutine raised an exception: {exception!r}")
...     else:
...         print(f"The coroutine returned : {future.result()}")
...
>>> asyncio.run(main())
The coroutine raised an exception: IndexError('list index out of range')
```

Voici maintenant ce qu'il se passe lorsque l'exception n'est pas levée :

```
>>> async def main():
...     loop = asyncio.get_running_loop()
...     future = loop.create_future()
...     task = loop.create_task(compute_value(future))
...     await future
...     exception = task.exception()
...     if exception:
...         print(f"The coroutine raised an exception: {exception!r}")
...     else:
...         print(f"The coroutine returned : {future.result()}")
...
>>> asyncio.run(main())
The coroutine returned : 42
```

Voici maintenant une méthode utilisant une boucle d'exception :

```
>>> async def main():
...     loop = asyncio.get_running_loop()
...     future = loop.create_future()
...     task = loop.create_task(unhandle_mistake(future))
...     await future
...     try:
...         task.result()
...     except Exception as e:
...         print(f"The coroutine raised an exception:
{task.exception()!r}")
...     else:
...         print(f"The coroutine returned : {task.result()}")
```

```
...
>>> asyncio.run(main())
The coroutine raised an exception: IndexError('list index out of range')
```

Et voici ce qu'il se passe lorsqu'il n'y a pas d'exceptions :

```
>>> async def main():
...     loop = asyncio.get_running_loop()
...     future = loop.create_future()
...     task = loop.create_task(compute_value(future))
...     await future
...     try:
...         task.result()
...     except Exception as e:
...         print(f"The coroutine raised an exception:
{task.exception()!r}")
...     else:
...         print(f"The coroutine returned : {task.result()}")
...
>>> asyncio.run(main())
The coroutine returned : 42
```

On a vu ici les bases de l'utilisation des futures et des tâches associées, ceci termine l'introduction à ces objets.

6. Utiliser la boucle événementielle

6.1 Utilisation des fonctions de retour (callbacks)

Le rôle du retour de fonction (*callback* en anglais) consiste à permettre de réaliser une opération que l'on souhaite faire avant même que la future n'ait un résultat. On programme donc cette opération avant et elle s'exécute après que la future soit calculée.
Voici un exemple simple où l'on se contente de programmer un affichage. Voici comment on peut écrire un tel callback :

```
>>> def print_callback(future):
...     print('Callback: future done: {}'.format(future.result()))
...
```

Voici maintenant comment l'utiliser :

```
>>> async def main():
...     loop = asyncio.get_running_loop()
...     future = loop.create_future()
...     future.add_done_callback(print_callback)
...     task = loop.create_task(compute_value(future))
...     await future
...     exception = task.exception()
...     if exception:
...         print(f"The coroutine raised an exception: {exception!r}")
...     else:
```

© Editions ENI - All rights reserved

```
...         print(f"The coroutine returned : {future.result()}")
...
>>> asyncio.run(main())
Callback: future done: 42
The coroutine returned : 42
```

La ligne mise en évidence permet de rajouter ce callback. On voit qu'elle prend place avant l'appel concret de la future, ce qui se fait lors de l'utilisation de **`await`**.
Chaque callback peut être ajouté plusieurs fois :

```
>>> async def main():
...     loop = asyncio.get_running_loop()
...     future = loop.create_future()
...     future.add_done_callback(print_callback)
...     future.add_done_callback(print_callback)
...     future.add_done_callback(print_callback)
...     task = loop.create_task(compute_value(future))
...     await future
...     exception = task.exception()
...     if exception:
...         print(f"The coroutine raised an exception: {exception!r}")
...     else:
...         print(f"The coroutine returned : {future.result()}")
...
>>> asyncio.run(main())
Callback: future done: 42
Callback: future done: 42
Callback: future done: 42
The coroutine returned : 42
```

Et il est aussi possible de le supprimer en passant à nouveau le callback. Ceci peut être surprenant si vous êtes habitué au fonctionnement des callbacks avec Gtk, par exemple. Ici, la méthode **`add_done_callback`** ne renvoie rien et pour supprimer un callback, il faut passer en paramètre de **`remove_done_callback`** le même objet que celui que l'on avait passé à **`add_done_callback`** :

```
>>> async def main():
...     loop = asyncio.get_running_loop()
...     future = loop.create_future()
...     future.add_done_callback(print_callback)
...     future.add_done_callback(print_callback)
...     future.add_done_callback(print_callback)
...     future.remove_done_callback(print_callback)
...     task = loop.create_task(compute_value(future))
...     await future
...     exception = task.exception()
...     if exception:
...         print(f"The coroutine raised an exception: {exception!r}")
...     else:
...         print(f"The coroutine returned : {future.result()}")
```

```
...
>>> asyncio.run(main())
The coroutine returned : 42
```

On remarque que si on a ajouté plusieurs fois le même callback, alors lorsqu'on le supprime, on supprime toutes ses occurrences.
Il est à noter qu'un callback ne prend pas d'argument lorsqu'il est passé à la future. Pour faire néanmoins cela, le plus simple est de rajouter les arguments à la fonction de rappel et utiliser ensuite **partial** pour passer ces arguments, en quelque sorte. En fait, **partial** va créer à la volée une fonction qui est celle passée en premier argument avec les arguments préremplis par les autres arguments :

```
>>> async def main():
...     loop = asyncio.get_running_loop()
...     future = loop.create_future()
...     future.add_done_callback(partial(print_callback, prefix="A"))
...     future.add_done_callback(partial(print_callback, prefix="B"))
...     future.add_done_callback(partial(print_callback, prefix="C"))
...     future.remove_done_callback(partial(print_callback, prefix="B"))
...     task = loop.create_task(compute_value(future))
...     await future
...     exception = task.exception()
...     if exception:
...         print(f"The coroutine raised an exception: {exception!r}")
...     else:
...         print(f"The coroutine returned : {future.result()}")
...
>>> asyncio.run(main())
A: future done: 42
B: future done: 42
C: future done: 42
The coroutine returned : 42
```

À noter qu'ici on a voulu supprimer l'appel du callback préfixé par **B**, et cela n'a pas marché. Deux enseignements sont à tirer de cet exemple : le premier est que la méthode **remove_done_callback** ne lève pas d'exception lorsqu'on lui passe un callback qu'elle ne connaît pas, et l'autre est le fait que le même code ne fait pas la même fonction.
En fait, deux fonctions égales ne sont pas identiques. Chaque fonction a un identifiant unique, y compris si elles sont identiques. Ce sont des objets mutables. Par conséquent, si on souhaite éventuellement supprimer des callbacks, il faut garder des pointeurs vers eux, autrement dit les mettre dans des variables au niveau local pour pouvoir les supprimer ensuite. Voici un exemple :

```
>>> async def main():
...     loop = asyncio.get_running_loop()
...     future = loop.create_future()
...     future.add_done_callback(partial(print_callback, prefix="A"))
...     print_b_callback = partial(print_callback, prefix="B")
...     future.add_done_callback(print_b_callback)
...     future.add_done_callback(partial(print_callback, prefix="C"))
```

© Editions ENI - All rights reserved

```
...     future.remove_done_callback(print_b_callback)
...     task = loop.create_task(compute_value(future))
...     await future
...     exception = task.exception()
...     if exception:
...         print(f"The coroutine raised an exception: {exception!r}")
...     else:
...         print(f"The coroutine returned : {future.result()}")
...
>>> asyncio.run(main())
A: future done: 42
C: future done: 42
The coroutine returned : 42
```

6.2 Planifier des appels à des fonctions classiques

Il est possible de planifier des appels à des fonctions qui ne sont pas asynchrones. Ces fonctions seront exécutées de manière asynchrone, c'est-à-dire sans que l'on puisse prédire précisément quand ni même dans quel ordre. L'ordonnanceur les exécutera lorsqu'il détectera un temps libre.
Voici un exemple simple :

```
>>> def function(arg, *, kwarg="default"):
...     print(f"function is called with arg={arg} and kwarg={kwarg}")
...
>>> async def main():
...     loop = asyncio.get_running_loop()
...     loop.call_soon(function, 42)
...     loop.call_soon(partial(function, kwarg="other"), 34)
...     loop.call_soon(partial(function, 16, kwarg="another"))
...     await asyncio.sleep(1)
...
>>> asyncio.run(main())
function is called with arg=42 and kwarg=default
function is called with arg=34 and kwarg=other
function is called with arg=16 and kwarg=another
```

Comme on peut le voir, il est possible de passer des arguments positionnels directement à la fonction **`call_soon`**, mais pas des arguments nommés. Pour cela, il faut passer par l'utilisation de **`functools.partial`**, comme nous l'avions déjà vu précédemment. À noter que l'on peut également utiliser cette dernière pour passer tout type d'arguments.
On peut aussi rajouter un délai d'exécution si on le souhaite :

```
>>> async def main():
...     loop = asyncio.get_running_loop()
...     loop.call_later(0.2, function, 42)
...     loop.call_later(0.5, partial(function, kwarg="other"), 34)
...     loop.call_soon(partial(function, 16, kwarg="another"))
...     await asyncio.sleep(0.8)
...
```

```
>>> event_loop = asyncio.new_event_loop()
>>> try:
...     event_loop.run_until_complete(main(event_loop))
... finally:
...     event_loop.close()
...
function is called with arg=16 and kwarg=another
function is called with arg=42 and kwarg=default
function is called with arg=34 and kwarg=other
```

Il suffit pour cela d'utiliser **`call_later`** et d'y passer en premier argument le délai à utiliser. On peut aussi faire exactement la même chose en utilisant **`call_at`** et en calculant l'instant d'exécution en ajoutant le délai au repère temporel de la boucle que l'on peut avoir grâce à **`loop.time`** :

```
>>> async def main():
...     loop = asyncio.get_running_loop()
...     now = loop.time()
...     print(f"loop.time={now} != time.time={time.time()}")
...     loop.call_at(now + 0.2, function, 42)
...     loop.call_later(0.5, partial(function, kwarg="other"), 34)
...     loop.call_soon(partial(function, 16, kwarg="another"))
...     await asyncio.sleep(0.8)
...
>>> asyncio.run(main())
loop.time=71242.794065111 != time.time=1551522224.046171
function is called with arg=16 and kwarg=another
function is called with arg=42 and kwarg=default
function is called with arg=34 and kwarg=other
```

On notera que **`loop.time()`** n'est pas la même chose que **`time.time()`**.

6.3 Utiliser des signaux

Le système d'exploitation fonctionne à l'aide de signaux. Certains d'entre eux peuvent être récupérés par le programme pour faire une action, ou subis par eux. Par exemple, l'utilisation de **[Ctrl] C**. Avec Linux, cette combinaison de touches envoie le signal SIGINT. Ces signaux sont gérés par le module signal.

Remarque

*Attention ! Tous les raccourcis de ce genre ne sont pas des signaux. Par exemple, **[Ctrl] D** correspond à un **EOF** (end of file) et est utilisé pour dire que l'on souhaite terminer la saisie. C'est pour cela qu'il est utilisé pour terminer la console Python ou Bpython, ou encore pour prendre un exemple hors contexte Python, un terminal.*

Il est possible de gérer ses signaux de manière asynchrone. De cette façon, lorsque l'on quittera un programme asynchrone, on pourra le faire proprement.

© Editions ENI - All rights reserved

Voici une mise en situation :

```
import asyncio
import signal

async def wait_for_interrupt():
    loop = asyncio.get_event_loop()
    future = loop.create_future()
    loop.add_signal_handler(signal.SIGINT, future.set_result, None)
    try:
        await future
    finally:
        loop.remove_signal_handler(signal.SIGINT)

if __name__ == "__main__":
    asyncio.run(wait_for_interrupt())
```

Ici, on crée une future juste pour pouvoir l'attendre. Puis, on rajoute un gestionnaire de signaux qui va écouter **`signal.SIGINT`** et mettre un résultat dans la future pour qu'elle se termine.
Lorsqu'elle sera terminée, on supprime le gestionnaire de signaux.
Ce code est plutôt simple, élégant et il est assez aisé de le mettre en œuvre dans tous vos programmes, voire de l'adapter pour qu'il fasse plus de tâches à la détection du signal, ce que vous ferez en rajoutant des instructions dans la boucle **`finally`**.

```
Ce code se trouve dans asyncio/signal/test_1.py
```

Voici un exemple qui d'un côté gère les signaux et de l'autre les envoie via une coroutine.
Le programme commence par les imports nécessaires :

```
import asyncio
import functools
import os
import signal
```

Voici la liste des signaux que nous allons gérer :

```
SIGNAL_NAMES = ("SIGINT", "SIGHUP", "SIGUSR1")
```

Si vous êtes sous Windows, vous pourriez essayer :

```
SIGNAL_NAMES = ("CRTL_C_EVENT", "CTRL_BREAK_EVENT")
```

Voici la fonction de rappel qui va gérer ces signaux :

```
def signal_handler(name):
    print(f"signal {name!r} handled")
```

Et la coroutine qui va les générer (à l'aide de **`os.kill`**) :

```
async def send_signals():
    pid = os.getpid()
    print(f"starting send_signals for {pid}")

    for name in SIGNAL_NAMES:
```

```
        print(f"Sending signal {name} ...", end="")
        os.kill(pid, getattr(signal, name))
        print("Done")
        await asyncio.sleep(0.01)
    return
```

Voici la coroutine principale qui va créer les gestionnaires de signaux et lancer la tâche qui va envoyer les signaux :

```
async def main():
    event_loop = asyncio.get_event_loop()

    for signal_name in SIGNAL_NAMES:
        event_loop.add_signal_handler(
            getattr(signal, signal_name),
            functools.partial(signal_handler, name=signal_name),
        )

    await send_signals()
```

Il ne reste plus qu'à lancer cette coroutine :

```
if __name__ == "__main__":
    asyncio.run(main())
```

Voici le résultat :

```
$ python test_2.py
starting send_signals for 17403
Sending signal SIGINT ...Done
signal 'SIGINT' handled
Sending signal SIGHUP ...Done
signal 'SIGHUP' handled
Sending signal SIGUSR1 ...Done
signal 'SIGUSR1' handled
```

Comme on peut le voir, le fait de gérer ces signaux permet d'éviter qu'ils arrêtent le programme, car ceci se ferait de manière incontrôlée.
Par contre, pour respecter l'envoi de ces signaux, il faut tout de même faire l'action de fermer proprement le programme. Et si l'on anticipe que cela puisse prendre du temps, alors il faut éventuellement signaler à l'utilisateur que vous avez reçu le signal et que vous êtes en train de le prendre en compte.

Remarque

Ce code se trouve dans asyncio/signal/test_2.py

© Editions ENI - All rights reserved

6.4 Synchronisation

Bien que les programmes asynchrones tournent généralement sur un seul fil d'exécution et un seul processus, ils peuvent néanmoins se retrouver en concurrence. Pour cela, nul besoin d'aller chercher trop loin. Il suffit de savoir utiliser quelques primitives de bas niveau qui vous suffiront à résoudre toutes les problématiques que vous rencontrerez.

La première de ces solutions est l'utilisation d'un jeton (*lock* en anglais, souvent traduit par verrou). Le principe consiste à dire que seul celui qui possède le jeton peut accéder à une ressource partagée. Il commence donc par attendre d'obtenir ce jeton, puis en prend possession, fait son travail et le rend lorsqu'il a terminé pour qu'un autre puisse s'en emparer à son tour.

Voici une telle coroutine :

```
>>> async def say(lock, what):
...     print(f"before {what} {time.strftime('%X')}")
...     await lock.acquire()
...     try:
...         await asyncio.sleep(1)
...         print(f"saying {what} {time.strftime('%X')}")
...     finally:
...         lock.release()
...     return what
...
```

L'instruction **`await lock.acquire()`** permet de prendre possession du jeton tandis que **`lock.release()`** permet de le rendre. Le rendre est immédiat par contre pour l'acquérir, il faut potentiellement attendre qu'une autre coroutine le relâche, d'où l'utilisation de **`await`**. On notera aussi l'utilisation d'un bloc **`try finally`** car il est important de toujours relâcher le jeton, que l'on ait une exception ou non.

Ceci peut avantageusement être remplacé par l'utilisation d'un gestionnaire de contexte asynchrone :

```
>>> async def say(lock, what):
...     print(f"before {what} {time.strftime('%X')}")
...     async with lock:
...         await asyncio.sleep(0.3)
...         print(f"saying {what} {time.strftime('%X')}")
...     return what
...
```

Une fois notre coroutine en place, nous allons volontairement créer le jeton dans un état bloqué puis le débloquer après un certain temps d'attente. Pour cela, nous allons appeler la fonction suivante :

```
>>> def countdown(lock):
...     for i in range(3, -1, -1):
...         print(f"wait for {i} secs ...")
...         time.sleep(1)
...     print(f"\rThis is hapenning now: gogogo!")
...     lock.release()
...
```

Cette fonction va faire un compte à rebours puis relâcher le jeton.
Voici maintenant la coroutine principale :

```
>>> async def main():
...     lock = asyncio.Lock()
...     await lock.acquire()
...
...     loop = asyncio.get_running_loop()
...     loop.call_later(0.2, partial(countdown, lock))
...
...     print(f"started at {time.strftime('%X')}")
...     await asyncio.gather(
...         say(lock, 'hello'),
...         say(lock, 'world'),
...         say(lock, '!!!!!!'),
...     )
...     print(f"finished at {time.strftime('%X')}")
...
```

Les deux premières lignes sont la création du jeton et son verrouillage, les deux secondes servent à appeler avec un délai la fonction de compte à rebours et les autres servent à planifier les tâches à réaliser.
Voici le résultat :

```
started at 17:34:35
before hello 17:34:35
before world 17:34:36
before !!!!! 17:34:36
```

À présent, tout est prêt à démarrer, les tâches sont débutées et attendent toutes le jeton. Le compte à rebours va maintenant commencer :

```
wait for 3 secs ...
wait for 2 secs ...
wait for 1 secs ...
wait for 0 secs ...
This is hapenning now: gogogo!
saying hello 17:34:40
saying world 17:34:41
saying !!!!! 17:34:42
finished at 17:34:41>
```

Une fois que le jeton est lancé, les diverses tâches peuvent s'exécuter. Elles s'exécutent séquentiellement, car elles ne peuvent commencer leur tâche (qui consiste à attendre une seconde puis faire un affichage) que lorsqu'elles ont le jeton. Sans ce jeton, elles attendraient toutes la même seconde, comme on a pu le voir plus haut.
Il existe cependant une solution permettant à plusieurs tâches d'attendre un événement. Lorsque cet événement se produit, alors toutes les tâches peuvent travailler de manière asynchrone.

© Editions ENI - All rights reserved

Pour réaliser cela, on modifie notre coroutine de base qui prend maintenant un événement au lieu d'un jeton et qui attend après cet événement :

```
>>> async def say(event, what):
...     print(f"before {what} {time.strftime('%X')}")
...     await event.wait()
...     await asyncio.sleep(1)
...     print(f"saying {what} {time.strftime('%X')}")
...     return what
...
```

Comme tout à l'heure, on va créer une fonction qui aura pour rôle de déclencher l'événement. Pour cela, on lui passe l'événement en paramètre et on fait un compte à rebours pour ensuite déclencher l'événement, à la dernière ligne :

```
>>> def countdown(event):
...     for i in range(3, -1, -1):
...         print(f"wait for {i} secs …")
...         time.sleep(1)
...     print(f"\rThis is hapenning now: gogogo!")
...     event.set()
...
```

On peut maintenant passer à notre coroutine principale. Celle-ci va créer l'objet événement, puis déclencher la fonction compte à rebours avec un délai et attendre l'exécution des tâches :

```
>>> async def main():
...     event = asyncio.Event()
...
...     loop = asyncio.get_running_loop()
...     loop.call_later(0.2, partial(countdown, event))
...
...     print(f"started at {time.strftime('%X')}")
...     await asyncio.gather(
...         say(event, 'hello'),
...         say(event, 'world'),
...         say(event, '!!!!!!'),
...     )
...     print(f"finished at {time.strftime('%X')}")
...
```

Le résultat obtenu est exactement le même que tout à l'heure dans le sens où les tâches attendent que l'événement se produise avant de se jouer :

```
>>> asyncio.run(main())
started at 17:58:14
before hello 17:58:14
before world 17:58:14
before !!!!! 17:58:14
wait for 3 secs ...
wait for 2 secs ...
```

```
wait for 1 secs ...
wait for 0 secs ...
This is hapenning now: gogogo!
saying hello 17:58:19
saying world 17:58:19
saying !!!!! 17:58:19
finished at 17:58:19
```

Par contre, une fois que l'événement s'est déclenché, toutes les tâches sont à nouveau asynchrones et par conséquent se découlent dans la même seconde.

Enfin, sur ce sujet, on peut également présenter un objet nommé **condition**. Il s'agit en réalité d'une gestion de plusieurs jetons. L'idée est de gérer l'exécution des coroutines en acceptant qu'un certain nombre d'entre elles puisse se lancer, au fil de l'eau. Pour cela, **`condition.notify(X)`** permet d'en laisser **`X`** se dérouler : on donne **`X`** jetons et les premiers à s'en saisir peuvent s'exécuter, mais séquentiellement. Il existe aussi **`condition.notidy_all()`** qui permet de toutes les débloquer.

```
>>> async def say(condition, what):
...     async with condition:
...         print(f"before {what} {time.strftime('%X')}")
...         await condition.wait()
...         await asyncio.sleep(1)
...         print(f"saying {what} {time.strftime('%X')}")
...     return what
...
```

Comme on peut le voir, on utilise le gestionnaire de contexte sur la condition et en plus, on l'attend. Toutes les tâches vont immédiatement jusqu'à la phase d'attente.

Voici maintenant notre fonction de contrôle. On en a fait une coroutine, car on avait besoin de gérer le contexte asynchrone de la condition :

```
>>> async def countdown(conditions):
...     for i in range(3, -1, -1):
...         print(f"wait for {i} secs …")
...         time.sleep(1)
...     print(f"\rThis is hapenning now: gogogo!")
...     async with conditions:
...         conditions.notify(2)
...     await asyncio.sleep(3)
...     for i in range(3, -1, -1):
...         print(f"wait for {i} secs ...")
...         time.sleep(1)
...     print(f"\rThis is hapenning now: gogogo!")
...     async with conditions:
...         conditions.notify_all()
...
```

Dans cette coroutine, on attend 5 secondes, puis on relâche 2 jetons, puis on attend encore 5 secondes avant de tout relâcher.

© Editions ENI - All rights reserved

Voici maintenant la fonction principale :

```
>>> async def main():
...     condition = asyncio.Condition()
...
...     print(f"started at {time.strftime('%X')}")
...     await asyncio.gather(
...         say(condition, 'hello'),
...         say(condition, 'world'),
...         say(condition, '!!!!!!'),
...         countdown(condition),
...     )
...     print(f"finished at {time.strftime('%X')}")
...
```

On crée la condition et on la passe à toutes les coroutines. La coroutine de contrôle **countdown** est simplement rajoutée à **asyncio.gather** pour être lancée.

```
>>> asyncio.run(main())
started at 18:28:51
before hello 18:28:51
before world 18:28:52
before !!!!! 18:28:52
wait for 3 secs ...
wait for 2 secs ...
wait for 1 secs ...
wait for 0 secs ...
This is hapenning now: gogogo!
saying hello 18:28:57
saying world 18:28:58
wait for 3 secs ...
wait for 2 secs ...
wait for 1 secs ...
wait for 0 secs ...
This is hapenning now: gogogo!
Saying !!!!! 18:29:04
finished at 18:29:04
```

On voit dans le résultat que seules deux coroutines **say** s'exécutent après la première attente, puis que la dernière se joue ensuite. On voit aussi qu'elles sont séquentielles, car une seconde se déroule entre les deux premières coroutines **say**.

6.5 Communication entre coroutines

Il existe un objet **asyncio.Queue** qui permet d'échanger des données entre coroutines. Cette queue ou file d'attente peut recevoir des données d'une coroutine et en fournir à une autre. Un des exemples classiques est la problématique producteur/consommateur. Il s'agit d'un objet FIFO (first in, first out, soit premier entré, premier sorti), ce qui signifie que les données sont traitées dans l'ordre où elles sont fournies.

On va simuler ici un producteur qui génère des données toutes les trois dixièmes de secondes et deux consommateurs qui peuvent en lire chacun une toutes les sept dixièmes. À deux, ils sont donc légèrement plus lents que le producteur pour traiter les données.

Remarque

Dans notre exemple, on considère que lorsque la queue est vide, c'est que le producteur a terminé de fournir des données. C'est un parti pris discutable, mais cela nous permettra de séparer les différentes notions dans des coroutines différentes.

Commençons par le consommateur :

```
>>> async def consumer(queue, name):
...     print(f"consumer {name} is starting")
...     while True:
...         print(f"consumer {name} is waiting")
...         item = await queue.get()
...         print(f"consumer {name} is working on {item}")
...         await asyncio.sleep(0.7)
...         queue.task_done()
...     print(f"consumer {name} is ending")
...
```

Ce dernier se met en attente de recevoir de la donnée depuis la queue par un **`await queue.get()`** et lorsqu'il a terminé de traiter la donnée, il faut un **`queue.task_done()`**. Ces deux étapes sont essentielles.
Voici maintenant le producteur :

```
>>> async def producer(queue, nb):
...     print(f"producer is starting")
...     for i in range(1, nb+1):
...         print(f"producer is producing item {i}")
...         await queue.put(i)
...         await asyncio.sleep(0.3)
...
```

Ce dernier va utiliser **`await queue.put()`** pour rajouter de la donnée dans la queue. On utilise ceci car comme on le verra juste en dessous, la queue a une taille maximale et rien ne garantit que, au moment de cet appel, celle-ci ne soit pas déjà remplie. Dans ce cas, le mot-clé **`await`** permet d'attendre que l'un des consommateurs ait consommé une donnée et ainsi laissé une place. Il faut aussi noter que la donnée est consommée et donc retirée de la file d'attente uniquement après l'appel à **`task_done`** et non pas lors du **`get`**.
Voici maintenant le parti pris dont je parlais qui ne fait pas partie de ce que l'on voit usuellement pour traiter les problématiques producteur consommateur mais qui a une utilité pédagogique ici (je ne suis pas convaincu que ce soit un anti-pattern, mais son utilisation dans la vie réelle est pratique aussi) :

```
>>> async def consumer_manager(queue, consumers):
...     print(f"consumer manager is waiting")
...     await queue.join()
...     for consumer in consumers:
```

© Editions ENI - All rights reserved

```
...         print(f"consumer {consumer.name} is cancelled")
...         consumer.cancel()
...
```

L'instruction **`await queue.join()`** permet d'attendre que la file d'attente soit vide. Lorsque c'est le cas, on va annuler les tâches des consommateurs pour terminer le programme (le producteur se termine seul lorsqu'il a fini de produire ses données, tout comme ce gestionnaire des consommateurs).

Remarque

Il existe une autre solution consistant à produire de la fausse donnée (None) à destination des consommateurs pour que ceux-ci se terminent seuls lorsqu'ils la reçoivent. Mais ce n'est peut-être pas la meilleure solution. En effet, cela nous empêche, d'une part, d'utiliser None comme donnée utile, et d'autre part, cela oblige le producteur à connaître le nombre de ses consommateurs, introduisant une contrainte plus forte. Il est préférable d'avoir un gestionnaire de consommateurs qui gère ses consommateurs de manière autonome.

Voici maintenant la fonction principale qui crée une file d'attente, puis crée les consommateurs, le producteur et enfin le gestionnaire des consommateurs. Elle fournit ensuite l'ensemble à **`asyncio.gather`**, en précisant **`return_exceptions=True`**, car on ne veut pas d'exceptions à l'annulation des consommateurs.

```
>>> async def main():
...     loop = asyncio.get_running_loop()
...     queue = asyncio.Queue(maxsize=12)
...
...     consumers = []
...     for name in "AB":
...         task = loop.create_task(consumer(queue, name))
...         task.name = name
...         consumers.append(task)
...
...     print(f"started at {time.strftime('%X')}")
...     await asyncio.gather(
...         producer(queue, 10),
...         consumer_manager(queue, consumers),
...         *consumers,
...          return_exceptions=True,
...     )
...     print(f"finished at {time.strftime('%X')}")
...
```

À noter qu'ici on se complique un peu la vie pour la création des consommateurs, de manière volontaire, pour donner un nom à chaque tâche consommateur, juste pour réaliser l'affichage au moment de l'annulation de ces tâches. Sans cet affichage, on pourrait s'en passer.

Voici maintenant le déroulé, pour un producteur qui crée dix éléments et deux consommateurs :

```
>>> asyncio.run(main())
started at 22:21:52
consumer A is starting
consumer A is waiting
consumer B is starting
consumer B is waiting
producer is starting
producer is producing item 1
consumer manager is starting
consumer A is working on 1
consumer manager is waiting
producer is producing item 2
consumer B is working on 2
producer is producing item 3
consumer A is waiting
consumer A is working on 3
producer is producing item 4
consumer B is waiting
consumer B is working on 4
producer is producing item 5
consumer A is waiting
consumer A is working on 5
producer is producing item 6
consumer B is waiting
consumer B is working on 6
producer is producing item 7
consumer A is waiting
consumer A is working on 7
producer is producing item 8
consumer B is waiting
consumer B is working on 8
producer is producing item 9
producer is producing item 10
consumer A is waiting
consumer A is working on 9
consumer B is waiting
consumer B is working on 10
consumer A is waiting
consumer B is waiting
consumer A is cancelled
consumer B is cancelled
finished at 22:21:56
```

On voit que le gestionnaire des consommateurs est lancé juste après le producteur et les consommateurs. Il y a déjà à ce moment-là une tâche dans la file d'attente, qui est traitée par le premier consommateur, mais encore en cours de traitement, donc pas retirée de la file d'attente.

© Editions ENI - All rights reserved

6.6 Exemple : lecture de l'entrée standard

Pour les exemples qui vont suivre, nous allons créer un serveur de type Echo, c'est-à-dire qui renvoie le message qu'il reçoit, ainsi qu'un client qui envoie un message qui sera tapé au clavier par l'utilisateur.
Pour cela, nous devons lire l'entrée standard et le faire en asynchrone.
Voici une proposition pour réaliser cette tâche :

```
async def ainput(loop, prompt):
    sys.stdout.write(prompt)
    sys.stdout.flush()

    queue = asyncio.Queue()

    def response():
        loop.create_task(queue.put(sys.stdin.readline()))

    loop.add_reader(sys.stdin.fileno(), response)

    try:
        task = loop.create_task(queue.get())
        result = await task
        return result.strip().encode()

    except asyncio.futures.TimeoutError:
        sys.stdout.write("\n")
        sys.stdout.flush()
        return ""

    finally:
        loop.remove_reader(sys.stdin.fileno())
```

Elle est nommée **ainput** pour *input asynchrone*, la fonction qu'elle est censée reproduire en version asynchrone.
Il y a plusieurs choses à dire sur cette fonction. Tout d'abord, on gère l'affichage d'un message d'invite pour indiquer à l'utilisateur ce que l'on attend de lui en passant par **sys.stdout.write**. On va ensuite utiliser un objet de type **asyncio.Queue** pour stocker les messages saisis.
La fonction **response** permet de programmer la lecture de l'entrée saisie par l'utilisateur et la mettre dans la file d'attente.
Puis un reader appelle cette fonction dès lors que la saisie est effectivement réalisée sur l'entrée standard.
Ensuite, nous créons une tâche pour attendre une entrée depuis la file d'attente et attendons cette tâche-là.
Lorsque le résultat arrive, nous retirons le caractère de fin de ligne (ou tout autre espace blanc en fin ou début de chaîne), puis nous le convertissons en octets.
Une fois ceci réalisé, nous supprimons le reader.

C'est ici un excellent exemple permettant d'illustrer la manière dont un reader fonctionne, un autre objet très utilisé avec les transports et les protocoles.

6.7 Programmation parallèle et asynchrone

Lorsqu'une boucle événementielle tourne, elle utilise un fil d'exécution, par défaut le fil d'exécution du programme. Si vous utilisez l'asynchrone et la programmation parallèle en même temps, il faut prendre quelques précautions.

Lorsque vous appelez une tâche depuis un autre fil d'exécution, vous devez utiliser **`loop.call_soon_threadsafe`**. Cette fonction garantit le fait qu'il n'y aura pas de problématiques de partage de ressources. En effet, quasiment toutes les fonctionnalités que l'on a vues jusqu'à présent ne sont pas thread-safe. Par conséquent, si vous appelez depuis un fil d'exécution des tâches qui sont gérées dans un autre, cela va très certainement créer des problèmes. D'où l'utilisation de cette fonction.

6.8 Exécuter du code bloquant

Vous aurez souvent à traiter avec du code bloquant. Or, lorsque vous exécutez du code bloquant, vous décalez d'autant l'appel d'autres tâches ou futures qui auraient pu commencer plus tôt. La solution consiste à utiliser **`loop.run_in_executor`**.

Voici un exemple de code bloquant, tiré de la documentation officielle :

```
>>> def blocking_io():
...     with open('/dev/urandom', 'rb') as f:
...         return f.read(100)
...
```

Voici comment améliorer significativement les performances :

```
>>> import asyncio
>>> async def main():
...     loop = asyncio.get_running_loop()
...     result = await loop.run_in_executor(
...         None, blocking_io)
...     print('default thread pool', result)
>>> asyncio.run(main())
```

La syntaxe est relativement simple, conforme à ce que l'on a vu jusqu'à présent.

On peut aller plus loin en décidant de mélanger l'asynchrone et la programmation parallèle pour utiliser facilement plusieurs fils d'exécution :

```
>>> import asyncio
>>> import concurrent.futures
>>> async def main():
...     loop = asyncio.get_running_loop()
...     with concurrent.futures.ThreadPoolExecutor() as pool:
...         result = await loop.run_in_executor(
...             pool, blocking_io)
...         print('custom thread pool', result)
...
>>> asyncio.run(main())
```

© Editions ENI - All rights reserved

Lorsque le code bloquant n'est pas dû à une attente I/O, mais à de lourds calculs, l'utilisation de processus donne de meilleurs résultats que celle des fils d'exécution :

```
>>> import asyncio
>>> import concurrent.futures
>>> def cpu_bound():
...     return sum(i * i for i in range(10 ** 4))
...
>>> async def main():
...     loop = asyncio.get_running_loop()
...     with concurrent.futures.ProcessPoolExecutor() as pool:
...         result = await loop.run_in_executor(
...             pool, cpu_bound)
...         print('custom process pool', result)
...
>>> asyncio.run(main())
```

L'exemple concret à la fin de ce chapitre illustrera la puissance de cet outil.

7. L'asynchrone suivant les versions de Python

7.1 Python 3.7

Python 3.7 introduit la fonction **`asyncio.run`** que nous utilisons très largement dans ce chapitre (et dans le suivant). Il est à noter que cette fonctionnalité a été introduite à titre probatoire. Il est donc possible qu'elle disparaisse ou soit modifiée dans les versions futures. À l'instant où ce paragraphe est écrit, il semblerait qu'elle ne soit pas modifiée pour Python 3.8. Il est à prévoir que cette fonction soit confirmée ou remplacée pour Python 3.9.

Reprenons un exemple du livre, celui-ci :

```
>>> async def main():
...     loop = asyncio.get_running_loop()
...     loop.call_soon(function, 42)
...     loop.call_soon(partial(function, kwarg="other"), 34)
...     loop.call_soon(partial(function, 16, kwarg="another"))
...     await asyncio.sleep(1)
...
>>> asyncio.run(main())
```

Vous pouvez le remplacer par :

```
>>> async def main(loop):
...     loop.call_soon(function, 42)
...     loop.call_soon(partial(function, kwarg="other"), 34)
...     loop.call_soon(partial(function, 16, kwarg="another"))
...     await asyncio.sleep(1)
...
>>> event_loop = asyncio.get_event_loop()
>>> event_loop.run_until_complete(main(event_loop))
```

Ou encore :

```
>>> async def main(loop):
...     loop.call_soon(function, 42)
...     loop.call_soon(partial(function, kwarg="other"), 34)
...     loop.call_soon(partial(function, 16, kwarg="another"))
...     await asyncio.sleep(1)
...
>>> event_loop = asyncio.new_event_loop()
>>> try:
...     event_loop.run_until_complete(main(event_loop))
... finally:
...     event_loop.close()
...
```

La fonction **asyncio.run** se charge de créer et détruire la boucle événementielle, ce dernier exemple est donc plus proche de ce que fait cette fonction.

De plus, Python 3.7 introduit la fonction **get_running_loop()**, le module **contextvars**, le support de ce module par **loop.call_later()**, **loop.call_soon()**, **loop.call_at**, **Future.add_done_callback()** et **loop.call_soon_threadsafe()**.

On peut aussi citer la fonction **asyncio.create_task()** comme raccourci de **asyncio.get_event_loop().create_task()**, ainsi que l'ajout des deux fonctions **asyncio.all_tasks()** et **asyncio.current_task()**, ainsi que les deux politiques **WindowsSelectorEventLoopPolicy** et **WindowsProactorEventLoopPolicy**, sans mentionner de nombreuses autres améliorations.

7.2 Python 3.6

Cette version a confirmé le module **asyncio** qui était jusque-là considéré comme non validé ou provisoire et l'a largement stabilisé et amélioré (tout comme a été largement amélioré le module **tracemalloc** qui est très utile pour la programmation asynchrone).

Il n'y a cependant pas de nouveautés, pas de nouvelles fonctionnalités. La version 3.6 a été pour l'asynchrone une version qui lui a permis de gagner en maturité et en performance.

On peut cependant signaler l'ajout des générateurs asynchrones et des compréhensions asynchrones.

7.3 Python 3.5

C'est la version de Python qui a introduit la nouvelle syntaxe avec les pas-encore-mots-clés **async** et **await**, ce qui nous permet d'écrire des coroutines dans la forme que nous connaissons aujourd'hui.

Même si ces mots ne sont pas des mots-clés, un développement réalisé sous Python 3.5 ou 3.6 est identique à un développement 3.7, aux nouveautés apportées par Python 3.7 près.

Dans une migration de Python 3.5 vers Python 3.7, il ne sera recommandé d'utiliser **asyncio.run** que si cette dernière est elle aussi validée.

© Editions ENI - All rights reserved

7.4 Python 3.4

Cette version de Python est celle qui introduit le module **asyncio**, qui sera définitivement validé avec Python 3.6. Les mots-clés **async** et **await** n'existant pas encore, il faut s'en remettre à un décorateur et à une syntaxe empruntée aux générateurs.
Ainsi, l'exemple précédent serait réécrit comme suit :

```
>>> @asyncio.coroutine
... def main():
...     loop = asyncio.get_running_loop()
...     loop.call_soon(function, 42)
...     loop.call_soon(partial(function, kwarg="other"), 34)
...     loop.call_soon(partial(function, 16, kwarg="another"))
...     yield from asyncio.sleep(1)
...
>>> event_loop = asyncio.get_event_loop()
>>> event_loop.run_until_complete(main(event_loop))
```

Le fait de réutiliser **yield from** n'est pas idéal, car cela introduit une confusion entre générateurs et fonctions asynchrones.
Pour de nombreuses raisons, si vous avez du code basé sur Python 3.4 ou plus ancien, il vous est conseillé de migrer. La principale raison n'est pas liée à l'asynchrone, mais au fait que le support des anciennes versions d'un langage a toujours une date limite et que vous exposez vos applications à des vulnérabilités si vous ne prévoyez pas de migration au moins tous les trois ans. Une autre raison, liée essentiellement à l'asynchrone, est que l'API fournie par Python est relativement stable entre les versions 3.5 et 3.7.

7.5 Python 3.3 et inférieur

La bibliothèque asyncio est disponible à partir de Python 3.4 et il est précisé dans ses spécifications qu'elle ne sera pas rétroportée, ce qui a du sens, car elle introduit de très importants changements.
En d'autres termes, le langage Python ne vous fournit pas de fonctionnalités pour la programmation asynchrone, si vous êtes en Python 3.3 ou inférieur.
Cependant, si le langage lui-même ne prévoit rien sur ce sujet, il existe de très nombreuses bibliothèques autour.
En effet, on trouve de nombreuses alternatives, qui existaient très largement avant que **asyncio** ne pointe le bout de son nez et certaines autres qui ont pour but de fournir un service similaire à cet excellent module.
Ces alternatives sont décrites dans le chapitre Programmation asynchrone : alternatives. Ces dernières ne sont cependant pas que pour Python 3.3 ou inférieur, elles peuvent tout aussi bien être utilisées avec Python 3.7. À vous de vous faire votre opinion et de vous décider sur ce que vous désirez utiliser.

7.6 Python 2.7

Comme pour Python 3.3, le module **asyncio** ainsi que la nouvelle syntaxe avec **async** et **await** n'existe pas pour Python 2.7 et n'existera pas dans le futur.
Si vous avez du code basé sur Python 2.7, vous pouvez également lire les alternatives détaillées dans le chapitre Programmation asynchrone : alternatives.

8. Cas concret

8.1 Exemple de travail

Pour cet exemple, nous allons présenter un extrait de code non asynchrone, puis le transformer en asynchrone. Ce code fait deux actions : télécharger des images et les enregistrer dans un fichier. Ces deux actions utilisent beaucoup d'I/O, mais la première se fait à travers le réseau et subit donc pas mal de temps d'attente tandis que la seconde utilise directement le système de fichiers et a peu de temps d'attente même si c'est de l'I/O. De plus, aujourd'hui, le langage Python ne propose pas de moyen d'écrire ou de lire des fichiers de manière asynchrone.
Le premier module auquel on pourrait penser pour accéder à une ressource web serait l'excellent module **requests**.
Or, ce dernier n'est pas asynchrone et il ne propose pas de travailler de manière non bloquante, comme la documentation le précise (http://docs.python-requests.org/en/master/user/advanced/#blocking-or-non-blocking). Le programme sera donc bloqué du début du téléchargement de la ressource jusqu'à la fin. Pour travailler en mode non bloquant, la documentation propose trois alternatives : **requests-thread** qui, comme son nom l'indique, propose une solution en utilisant la programmation parallèle, **grequests** qui, comme son nom l'indique plus subtilement, propose d'utiliser gevent, une forme d'utilisation de fils d'exécution légers et vue comme une alternative à l'asynchrone (présentée dans le chapitre Programmation asynchrone : alternatives) et enfin **requests-future** qui utilise la notion de future, mais du module **concurrent.future** et non la notion de **asyncio.Future**, une autre solution impliquant la programmation parallèle.
Nous verrons donc ce cas d'usage et ses alternatives dans le chapitre sur la programmation parallèle (Qualité), ainsi que dans celui sur les alternatives à l'asynchrone (Génération de contenu).
Cette parenthèse étant faite, nous allons proposer un code de très bas niveau pour visualiser tout ce qu'il se passe.
Dans cet exemple, nous allons créer des fonctions de bas niveau nous permettant de télécharger le contenu d'une page HTML et de récupérer toutes ses images. Cela se fait en plusieurs temps parce qu'il faut récupérer d'abord une page internet, puis la parser, chercher les balises d'images, calculer les URL complètes de ses images et enfin télécharger et enregistrer sur le disque ces dernières.
Voici la fonction au cœur de notre cas concret :

```
def wget(uri):
    """
    Renvoie le contenu désigné par une URI
```

© Editions ENI - All rights reserved

```
    Paramètre :
    > uri (str, par exemple 'http://inspyration.org')

    Retour :
    > contenu d'un fichier (bytes, fichier textuel ou binaire)
    """
    # Analyse de l'URI
    parsed = urlparse(uri)
    # Ouverture de la connexion
    with closing(HTTPConnection(parsed.netloc)) as conn:
        # Chemin sur le serveur
        path = parsed.path
        if parsed.query:
            path += '?' + parsed.query
        # Envoi de la requête vers le serveur
        conn.request('GET', path)
        # Récupération de la réponse
        response = conn.getresponse()
        # Analyse de la réponse
        if response.status != 200:
            # 200 = Ok, 3xx = redirection, 4xx = erreur client,
5xx = erreur serveur
            print(response.reason, file=stderr)
            return
        # Renvoi de la réponse si tout est OK.
        print('Réponse OK')
        return response.read()
```

Ce que l'on récupère est le code source HTML de la page. Il faut maintenant le parser à la recherche des images :

```
def get_images_src_from_html(html_doc):
    """Récupère tous les contenus des attributs src des balises img"""
    soup = BeautifulSoup(html_doc, "html.parser")
    return [img.get('src') for img in soup.find_all('img')]
```

Le souci avec une source d'image, c'est qu'il peut s'agir d'une URI absolue ou d'une relative et si elle est relative, elle peut l'être par rapport à la racine du site ou par rapport au répertoire courant. Il faut donc traiter tous ces cas pour retrouver l'URI absolue de chaque image :

```
def get_uri_from_images_src(base_uri, images_src):
    """Renvoie une à une chaque URI d'image à télécharger"""
    parsed_base = urlparse(base_uri)
    result = []
    for src in images_src:
        parsed = urlparse(src)
        if parsed.netloc == '':
            path = parsed.path
            if parsed.query:
```

```
                path += '?' + parsed.query
            if path[0] != '/':
                if parsed_base.path == '/':
                    path = '/' + path
                else:
                    path = '/' + '/'.join(parsed_base.path.split('/')
[:-1]) + '/' + path
            result.append(parsed_base.scheme + '://' +
parsed_base.netloc + path)
        else:
            result.append(parsed.geturl())
    return result
```

Enfin, il faut télécharger l'image :

```
def download(uri):
    """
    Enregistre sur le disque dur un fichier désigné par une URI
    """
    content = wget(uri)
    if content is None:
        return None
    with open(uri.split(sep)[-1], 'wb') as f:
        f.write(content)
        return uri
```

On réutilise ici la fonction **wget**, puisque télécharger du HTML ou une image est un processus identique.
Voici pour terminer la fonction qui fait travailler toutes les autres :

```
def get_images(page_uri):
    #
    # Récupération des URI de toutes les images d'une page
    #
    html = wget(page_uri)
    if not html:
        print("Erreur: aucune image n'a été trouvée", sys.stderr)
        return None
    images_src_gen = get_images_src_from_html(html)
    images_uri_gen = get_uri_from_images_src(page_uri, images_src_gen)
    #
    # Récupération des images
    #
    for image_uri in images_uri_gen:
        print('Téléchargement de %s' % image_uri)
        download(image_uri)
```

Et la manière dont on peut tester la performance de ce script :

```
if __name__ == '__main__':
    print('--- Starting standard download ---')
```

© Editions ENI - All rights reserved

```
web_page_uri = 'http://www.formation-python.com/'
print(timeit('get_images(web_page_uri)',
             number=10,
             setup="from __main__ import get_images, web_page_uri"))
```

Remarque

Cet exemple est présent dans le fichier get_images_standard.py.

Sur la machine de l'auteur, cela donne un résultat de 1,75 seconde pour télécharger 10 images.

9. Exemple retravaillé en utilisant des générateurs

Quand utiliser des générateurs et quand utiliser l'asynchrone ? L'asynchrone est à réserver lorsque l'on fait des actions impliquant du travail externe (utilisation de nombreuses I/O qui mettent le programme en pause). Tant qu'il s'agit d'un algorithme manipulant des données, si on ne souhaite pas attendre de collecter toutes les données avant de les traiter, la meilleure chose à faire est d'essayer d'utiliser des générateurs.
Et les générateurs ne sont pas exclusifs de la programmation asynchrone : nous pouvons utiliser les deux ensemble. Nous allons détailler comment transformer le code de base pour l'optimiser un peu :
Le processus de récupération des images depuis le code source HTML peut facilement être transformé en générateur :

```
def get_images_src_from_html(html_doc):
    """Récupère tous les contenus des attributs src des balises img"""
    soup = BeautifulSoup(html_doc, "html.parser")
    return (img.get('src') for img in soup.find_all('img'))
```

Il a suffi de remplacer les crochets par des parenthèses.
L'étape suivante, consistant à retrouver les URI absolues, est plus complexe. Il faut donc utiliser cette technique :

```
def get_uri_from_images_src(base_uri, images_src):
    """Renvoie une à une chaque URI d'image à télécharger"""
    parsed_base = urlparse(base_uri)
    for src in images_src:
        parsed = urlparse(src)
        if parsed.netloc == '':
            path = parsed.path
            if parsed.query:
                path += '?' + parsed.query
            if path[0] != '/':
                if parsed_base.path == '/':
                    path = '/' + path
                else:
                    path = '/' + '/'.join(parsed_base.path.split('/')
[:-1]) + '/' + path
            yield parsed_base.scheme + '://' + parsed_base.netloc + path
```

```
        else:
            yield parsed.geturl()
```

Les changements sont en gras. Là encore, ils sont assez légers. On ne construit pas de variable résultat pour la renvoyer à la fin, on renvoie chaque résultat au moment où il arrive.
Sur la machine de l'auteur, cela donne un résultat de 1,65 seconde pour télécharger 10 images, mais étant donné que cette opération n'est pas ce qui prend le plus de temps, ce n'est pas l'optimisation essentielle. On utilise également moins de mémoire.

Remarque

Cet exemple est présent dans le fichier get_images_generateurs.py.

9.1 Télécharger en asynchrone

Si on regarde le code, il y a plusieurs endroits où l'on utilise des ressources externes. Tout d'abord, on télécharge le code de la page. Mais ce dernier est passé à BeautifulSoup, qui est un parseur qui a besoin de l'ensemble de la page pour travailler. Ce n'est donc pas là que l'on va pouvoir faire des optimisations.
Par contre, on peut optimiser la fonction qui télécharge les images et celle qui écrit le fichier en les rendant asynchrones.
On va commencer par la fonction qui télécharge les images, car comme cela utilise le réseau, le gain potentiel de performances est le plus important.
On va utiliser le module **aiohttp** (*Asynchronous Input/Output HTTP*), qui est un module conçu pour travailler de manière asynchrone, contrairement aux primitives standards du langage Python :
Pour télécharger un fichier avec ce module, rien de plus simple :

```
>>> async def main(uri):
...     async with aiohttp.ClientSession() as session:
...         async with session.get(uri) as response:
...             if response.status != 200:
...                 return None
...             if response.content_type.startswith("text/"):
...                 return await response.text()
...             else:
...                 return await response.read()
...
>>> asyncio.run(main("http://www.formation-python.com/"))
```

On utilise ici deux fois le gestionnaire de contexte asynchrone. La première pour gérer le client, qui va créer la connexion réseau et la quitter à la fin du bloc, puis la seconde pour gérer la session qui elle aussi a une durée de vie et doit être quittée proprement.
Cet algorithme fait aussi la différence entre un contenu textuel (d'après le content type) et va soit renvoyer la donnée sous forme de texte (idéal pour du code HTLM), soit sous forme d'octets (idéal pour les images).
On va donc pouvoir l'utiliser à la fois pour télécharger la page HTML et pour télécharger chaque image.

© Editions ENI - All rights reserved

Dans notre exemple, la session durera tout le temps d'exécution de l'algorithme. Elle sera donc passée en paramètres aux fonctions appelées.
Au final, la fonction **wget** ressemblera à :

```
async def wget(session, uri):
    async with session.get(uri) as response:
        if response.status != 200:
            return None
        if response.content_type.startswith("text/"):
            return await response.text()
        else:
            return await response.read()
```

La fonction **download** sera modifiée ainsi :

```
async def download(session, uri):
    content = await wget(session, uri)
    if content is None:
        return None
    with open(uri.split(sep)[-1], "wb") as f:
        f.write(content)
        return uri
```

Toute la dernière partie (les trois dernières lignes) est l'opération d'écriture du fichier sur le disque dur et elle n'est pas asynchrone. Elle constitue la dernière partie bloquante de ce code.

9.2 Parser en asynchrone

La technique utilisée ici est l'utilisation de générateurs asynchrones.
Pour parser le fichier, nous allons utiliser un générateur asynchrone : dès qu'il aura trouvé une image, il la passera à la fonction suivante et la main nous sera rendue lors d'une prochaine attente ailleurs dans le programme :

```
async def get_images_src_from_html(html_doc):
    soup = BeautifulSoup(html_doc, "html.parser")
    for img in soup.find_all('img'):
        yield img.get('src')
        await asyncio.sleep(0.001)
```

Le rajout de l'instruction d'attente permet de s'assurer qu'on laisse bien la main.
La fonction suivante est :

```
async def get_uri_from_images_src(base_uri, images_src):
    """Renvoi un à un chaque URI d'image à télécharger"""
    parsed_base = urlparse(base_uri)
    async for src in images_src:
        parsed = urlparse(src)
        if parsed.netloc == '':
            path = parsed.path
            if parsed.query:
                path += '?' + parsed.query
```

```
            if path[0] != '/':
                if parsed_base.path == '/':
                    path = '/' + path
                else:
                    path = '/' + '/'.join(parsed_base.path.split('/')[:-
1]) + '/' + path
                yield parsed_base.scheme + '://' + parsed_base.netloc + path
            else:
                yield parsed.geturl()
            await asyncio.sleep(0.001)
```

Il s'agit aussi d'un générateur, mais qui travaille à partir d'un autre générateur qui lui est passé en paramètres : **`images_src`**.
Cette technique est utile pour enchaîner le traitement de données.

9.3 Faire plusieurs traitements en asynchrone

On arrive maintenant dans la fonction principale. La première partie de celle-ci consiste à télécharger la page et à récupérer le code HTML. On va ensuite créer les deux générateurs enchaînés et les utiliser dans une itération asynchrone :

```
async def get_images(session, page_uri):
    html = await wget(session, page_uri)
    if not html:
        print("Erreur: aucune image n'a été trouvée", sys.stderr)
        return None
    images_src_gen = get_images_src_from_html(html)
    images_uri_gen = get_uri_from_images_src(page_uri, images_src_gen)
    async for image_uri in images_uri_gen:
        print('Téléchargement de %s' % image_uri)
        await download(session, image_uri)
```

Cette itération permet de traiter chaque élément au fur et à mesure et de minimiser les temps d'attente.

Voici maintenant la fonction principale, conformément à ce que l'on a fait ici pour Python 3.7 :

```
async def main():
    web_page_uri = 'http://www.formation-python.com/'
    async with aiohttp.ClientSession() as session:
        await get_images(session, web_page_uri)
```

Puis :

```
asyncio.run(main())
```

Remarque

Cet exemple se trouve dans asyncio/exemples/get_images_asyncio.py

© Editions ENI - All rights reserved

Pour Python 3.5 ou 3.6, il faudrait écrire, à la place de cette dernière ligne, ceci :

```
event_loop = asyncio.get_event_loop()
event_loop.run_until_complete(main())
```

Remarque

Cet exemple se trouve dans asyncio/exemples/get_images_asyncio_35.py

Pour Python 3.4, il faudrait en plus remplacer tous les **`await`** par des **`yield from`**, supprimer tous les mots-clés **`async`** et rajouter le décorateur **`@asyncio.coroutine`** à toutes les coroutines.

Remarque

Cet exemple se trouve dans asyncio/exemples/get_images_asyncio_34.py

Le temps mis par ce programme est de 0,75 seconde (avec Python 3.7).

9.4 Utiliser un exécuteur

Nous avons vu qu'il y a une partie du code qui est bloquante. Pour diminuer l'attente, nous avons l'option de gérer cette partie du code dans un exécuteur à côté du code principal. Voici les transformations à faire. En premier lieu, créer la fonction qui gère uniquement la partie bloquante :

```
def write_in_file(filename, content):
    with open(filename, "wb") as f:
        f.write(content)
```

Puis en second lieu, utiliser un exécuteur :

```
async def download(session, uri):
    content = await wget(session, uri)
    if content is None:
        return None
    loop = asyncio.get_running_loop()
    await loop.run_in_executor(None, partial(write_in_file,
uri.split(sep)[-1], content))
    return uri
```

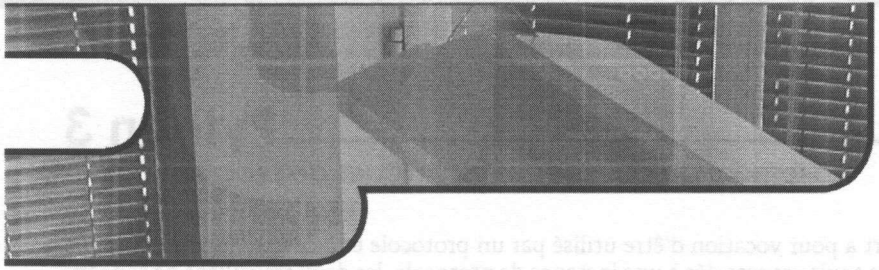
Cette simple modification permet de diviser encore le temps par 3, puisque l'on arrive à 0,3 seconde au lieu de 0.75 sans cette dernière modification et 1.75 avec le code non asynchrone de départ.

9.5 Pour aller plus loin

Le module **`aiohttp`** (http://aiohttp.readthedocs.org/en/stable/) est une référence de qualité pour faire du webscrapping, mais également pour énormément de besoins plus simples. Il est également possible de l'utiliser pour servir du contenu et pour de nombreuses autres raisons (https://github.com/KeepSafe/aiohttp/tree/master/examples).

Il se marie parfaitement bien avec le module **websocket** (https://websockets.readthedocs.org/en/stable/) et, avec les deux travaillant ensemble, il est possible de créer un serveur web Python implémentant le websocket et dialoguant avec du code JavaScript côté client.

© Editions ENI - All rights reserved



Chapitre 3.3
Programmation asynchrone : avancée

1. Transports et protocoles

1.1 Introduction

Ces notions de transport et de protocole (et on pourrait rajouter la notion de flux) sont des notions de bas niveau. Ces notions sont faites pour permettre d'automatiser des implémentations haute-performance de cas pratiques asynchrones usuels, comme on le verra plus tard pour du TCP, de l'UDP ou de l'appel de processus externe.

Les objets **`Transport`** ou **`Protocol`** ne sont pas faits pour être créés et gérés directement dans des scripts ou des applications, mais plutôt conçus comme des outils à destination de bibliothèques qui fourniront des outils de haut niveau, comme par exemple **`autobahn`** présenté dans le chapitre sur la programmation réseau.

Nous allons essayer de présenter ici les concepts généraux, sachant que la partie sur laquelle on insistera le plus sera celle présentant les transports et protocoles déjà implémentés par le langage.

Les transports sont faits pour décrire la manière dont les données sont transmises. Du moment que l'on parle de transmission de donnée à bas niveau, on parle toujours d'octets. Les transports décriront donc comment transporter des octets.

Les protocoles sont faits pour décrire quelles sont les données à transmettre, dans quel ordre les transmettre, quand les transmettre. En général, ils implémentent des RFC (spécifications génériques) et peuvent donc être potentiellement très complexes. Chaque instance d'un transport sera associée à une et une seule instance de protocole.

1.2 Transports

Les transports sont des classes fournies par **`asyncio`** permettant de créer une abstraction d'un canal de communication. Les transports ne sont généralement pas instanciés par vous, mais par la boucle d'événement. Cette dernière va initier le transport puis le canal de communication sous-jacent.

Un transport a pour vocation d'être utilisé par un protocole et à ce titre, une instance de transport est toujours associée à une instance de protocole, les deux travaillent de concert.
Pour commencer, nous allons importer la classe de base :

```
>>> from asyncio import BaseTransport
```

Pour créer un transport, il faut hériter de cette classe et implémenter quatre méthodes qui sont **is_closing**, **close**, **set_protocol** et **get_protocol**. Pour résoudre ces quatre méthodes, nous allons simplement utiliser deux attributs privés **_is_closing** et **_protocol**. Ces derniers seront initialisés dans le constructeur.
Pour gérer la fermeture, nous irons également programmer de manière asynchrone :
– la fermeture du protocole associé au transport courant
– la notification d'une future pour arrêter l'attente dans la boucle principale
En effet, la boucle principale ira faire un **await** sur une future, celle qui sera passée au constructeur du transport et qui sera notifiée au moment de la fermeture du transport pour terminer l'attente. Le constructeur prendra en paramètre cette future ainsi que la boucle, pour nous permettre de programmer les tâches dont nous avons besoin et le protocole avec lequel le transport courant travaillera main dans la main.
Voici une proposition pour cette classe :

```
>>> class ExampleTransport(BaseTransport):
...
...     def __init__(self, loop, protocol, future, extra=None):
...         super().__init__(extra)
...         self._loop = loop
...         self.set_protocol(protocol)
...         self._future = future
...         self._is_closing = False
...         self._loop.call_soon(self._protocol.connection_made, self)
...         print("Transport is created.")
...
...     def is_closing(self):
...         if self._is_closing:
...             print(f"The transport is closing")
...         else:
...             print(f"The transport is open")
...         return self._is_closing
...
...     def close(self):
...         if self._is_closing:
...             print(f"The transport is already closing")
...             return
...         self._is_closing = True
...         print(f"The transport is now closing")
...         self._loop.call_soon(self._call_connection_lost, None)
...
...     def set_protocol(self, protocol):
...         self._protocol = protocol
...
```

© Editions ENI - All rights reserved

```
...     def get_protocol(self):
...         return self._protocol
...
...     def _call_connection_lost(self, exc):
...         print("Connection is lost: triggering protocol.connection_lost
and notify future.")
...         try:
...             self._protocol.connection_lost(exc)
...         finally:
...             self._loop.call_soon(self._notify_future, self._future)
...             self._protocol = None
...             self._loop = None
...
...     def _notify_future(self, future):
...         print("notify future : this is the end.")
...         if not future.cancelled():
...             future.set_result(None)
...
```

Maintenant que le transport est écrit, nous allons détailler le protocole associé. Avant cela, sachez qu'il existe des transports abstraits spécialisés et que chacun a une liste de méthodes supplémentaires à implémenter :

- **ReadTransport** :
  - hérite de **BaseTransport**
  - méthodes à surcharger :
    - pause_reading
    - resume_reading
- **WriteTransport** :
  - hérite de **BaseTransport**
  - méthodes à surcharger :
    - **get_write_buffer_size**
    - **write**
    - **writelines**
    - **write_eof**
    - **can_write_eof**
    - **abort**
- **Transport**
  - adapté, entre autres, au protocole TCP
  - hérite de **ReadTransport** et **WriteTransport**
  - méthodes à surcharger : toutes les méthodes précédentes
- **DatagramTransport** :
  - hérite de **BaseTransport**
  - adapté au protocole UDP
  - hérite de **BaseTransport**

  - méthodes à surcharger :
    - **sendto**
    - **abort**
- **SubprocessTransport** :
  - hérite de **BaseTransport**
  - adapté à l'appel de processus externe (subprocess)
  - hérite de BaseTransport
  - méthodes à surcharger :
    - **get_pid**
    - **get_returncode**
    - **get_pipe_transport**
    - **send_signal**
    - **terminate**
    - **kill**

Il existe également un Mixin, nommé FlowControlMixin qui permet de gérer la logique d'écriture d'un flux de données.

Les classes concrètes de ces protocoles sont à trouver dans **asyncio.selector_events** et sont des classes privées, telles que **_SelectorSocketTransport**, par exemple, pour le protocole TCP.

Le fait que ces classes soient privées est cohérent avec le fait qu'elles sont destinées à être utilisées par les outils que nous verrons tout à l'heure et non directement par nous, développeurs d'application.

Pour terminer, il y a un détail très important à aborder dans le transport, c'est qu'il fournit une méthode nommée **get_extra_info** qui, par défaut, va renvoyer des éléments stockés dans l'attribut de type dictionnaire **_extra**. Dans la classe **BaseTransport**, ce dictionnaire ne contient rien, mais dans votre propre classe, vous pourrez y rajouter toutes les informations que vous jugerez utiles. Voici par exemple comment on peut lire un socket et en extraire des informations :

```
self._extra['socket'] = sock
self._extra['sockname'] = sock.getsockname()
if 'peername' not in self._extra:
    try:
        self._extra['peername'] = sock.getpeername()
    except socket.error:
        self._extra['peername'] = None
```

Cet exemple est extrait du code d'asyncio. Dans cet exemple, il laisse la possibilité de fixer peername en le passant dans le dictionnaire en argument du constructeur. Si cela n'est pas fait, alors il va interroger le socket pour le récupérer lui-même.

Lorsque le transport est construit au-dessus d'un socket, il fournit les informations suivantes :

- peername, l'adresse distante à laquelle le socket est connecté
- socket, l'instance du socket
- sockname, l'adresse du socket

Cela correspond au code que l'on a vu ci-dessus.

© Editions ENI - All rights reserved

Lorsque le transport est construit au-dessus d'un socket SSL, on trouve :
- compression : le nom de l'algorithme de compression utilisé ou **None**, le résultat d'un appel à **ssl.SSLSocket.compression()**.
- cipher : un 3-uplet contenant le nom du cipher utilisé, la version du protocole SSL ainsi que le nombre de bits secrets utilisés (résultat de **ssl.SSLSocket.cipher()**).
- peercert : le certificat peer, résultat de **ssl.SSLSocket.getpeercert()**.
- ssl context : l'instance de type **SSLContext** utilisée.

Pour les transports construits au-dessus de pipes :
- pipe : l'objet pipe au-dessus duquel le transport est construit.

Et enfin, au-dessus de subprocess :
- subprocess : l'instance **subprocess.Popen** utilisée par le transport.

Remarque

Pour information, la notion de pipe est une notion de programmation système qui est développée dans le chapitre dédié, de la même manière que l'utilisation du module **subprocess**. *Les autres notions sont développées dans le chapitre sur la programmation réseau.*

1.3 Protocoles

Les protocoles sont des abstractions de protocoles réseau, tels que TCP, UDP ou même HTTP, SSL ou autre. Le protocole va analyser les données entrantes et prendre les décisions appropriées au regard de la norme définie par le protocole. Ces décisions couvrent le fait de renvoyer de la donnée.

Pour créer un protocole, il faut hériter de **BaseProtocol**, que l'on peut trouver dans le fichier source **asyncio.protocols** :

```
>>> from asyncio import Baserotocol
```

Toutes les méthodes de cette classe sont vides, si ce n'est de la documentation, mais il ne s'agit pas d'une classe abstraite, puisqu'elle pourrait parfaitement être instanciée. La surcharge de ses méthodes est conseillée, mais pas obligatoire. Ces méthodes sont des fonctions de rappel appelées par le transport. Il ne faut donc jamais les appeler, sauf si on implémente un transport.

```
>>> class ExampleProtocol(BaseProtocol):
...
...     def __init__(self, loop, callback):
...         print("Protocol is created")
...         self._loop = loop
...         self._callback = callback
...         self._paused = False
...         self._exc = None
...         self._connection_lost = False
...         self._transport = None
...
...     def connection_made(self, transport):
...         print(f"Connection made")
```

```
...         self._transport = transport
...         self._loop.create_task(self._callback(self))
...
...     def connection_lost(self, exc):
...         print(f"connection lost, the protocol may want to do something
about it")
...         self._exc = exc
...         self._connection_lost = True
...         self._transport = None
...
```

La classe **BaseProtocol** a deux autres méthodes de rappel : **pause_writing** et **resume_writing**. La première est utile lorsque le buffer va au-delà d'une limite haute et la seconde l'est une fois qu'il repasse sous une limite basse.
On définit ici les deux méthodes que nous avons utilisées dans notre transport et nous nous limiterons là. Ces deux méthodes sont là pour préciser ce que l'on doit faire si on obtient une nouvelle connexion ou si on en perd une.
Sachez qu'il existe également :

- **Protocol** :
  - adapté au protocole TCP
  - a les méthodes de rappel suivantes :
    - **data_received**
    - **eof_received**
- **DatagramProtocol** :
  - adapté au protocole UDP
  - a les méthodes de rappel suivantes :
    - **datagram_received**
    - **error_received**
- **SubprocessProtocol** :
  - adapté aux appels externes (subprocess)
  - a les méthodes de rappel suivantes :
    - **pipe_data_received**
    - **pipe_connection_lost**
    - **process_exites**

Dans notre exemple, nous ne faisons pas grand-chose si ce n'est montrer les relations logiques entre ces composants et pour arriver à ce but, il faut que l'on utilise nos deux classes.

1.4 Mise en œuvre

Pour mettre en œuvre ces deux classes, il a été créé une fonction de rappel qui se contente d'attendre une seconde avant de fermer le transport. La voici :

```
>>> async def callback(protocol):
...     print("callback")
...     await asyncio.sleep(0.1)
...     protocol._transport.close()
...
```

© Editions ENI - All rights reserved

Voici maintenant comment mettre en route l'ensemble :

```
>>> async def main():
...     loop = asyncio.get_running_loop()
...     future = loop.create_future()
...     protocol = ExampleProtocol(loop, callback)
...     transport = ExampleTransport(loop, protocol, future)
...
...     print("## await future")
...     await future
...     print("## future awaited")
...
```

On va récupérer la boucle événementielle courante pour créer une future puis le protocole, le transport et enfin attendre la future.
Ce code est ce que vous ne verrez généralement jamais. En règle générale, on crée aussi des factory pour générer le protocole et le transport.
L'exécution de ce code donne le résultat suivant :

```
>>> asyncio.run(main())
Protocol is created
Transport is created.
## await future
Connection made
callback
The transport is now closing
Connection is lost: triggering protocol.connection_lost and notify future.
connection lost, the protocol may want to do something about it
notify future : this is the end.
## future awaited
```

On voit ici le déroulé du code et comment les éléments s'imbriquent entre eux, ce qui était le seul propos de ces exemples.
Ce qu'il faut retenir est que dans ces classes, vous pouvez créer toutes les méthodes que vous voulez, mais ces classes sont hautement imbriquées. Les méthodes de l'une sont faites pour être utilisées par les méthodes de l'autre et elles sont également faites pour répondre à un besoin spécifique.
Si vous souhaitez visualiser du code concret, vous pouvez regarder ces liens :
https://github.com/crossbario/autobahn-python/blob/master/autobahn/asyncio/websocket.py
https://github.com/crossbario/autobahn-python/blob/master/autobahn/websocket/protocol.py

1.5 Exemple pour le protocole TCP

Remarque

Vous trouverez toutes les explications nécessaires sur le protocole TCP ainsi que des exemples de serveurs TCP de bas niveau et de haut niveau dans le chapitre Programmation réseau.

Voici un exemple de serveur TCP. Pour ce fichier, on importe les modules nécessaires :

```
import asyncio
import logging
import coloredlogs
import sys
```

Puis on précise l'adresse réseau que le serveur va écouter :

```
SERVER_ADDRESS = ("localhost", 8040)
```

On crée un outil de logging pour avoir des affichages agréables et précis :

```
main_logger = logging.getLogger("main")
logger_format = "%(asctime)s,%(msecs)03d %(message)s"
coloredlogs.install(fmt=logger_format, level="DEBUG", logger=main_logger)
```

On peut maintenant créer notre classe, qui héritera de **`Protocol`** :

```
class EchoServerProtocol(asyncio.Protocol):
```

Voici la fonction **`connection_made`**, déclenchée après la connexion d'un client : elle agrège le transport, pour permettre à d'autres méthodes de l'utiliser éventuellement plus tard, elle récupère des informations réseau sur le client et initie un logger spécial pour cette instance (pour les différencier entre eux et du logger général).
Le transport dans ce cas-ci est une abstraction asynchrone qui utilise un socket TCP.

```
    def connection_made(self, transport):
        self.transport = transport
        self.address = transport.get_extra_info("peername")
        self.logger = logging.getLogger(f"Client {self.address}")
        coloredlogs.install(level="DEBUG", logger=self.logger)
        self.logger.debug("connection made")
```

La méthode suivante est déclenchée lorsqu'une donnée est reçue par le client : on se contente de la renvoyer, entre deux affichages. Pour la renvoyer, on écrit dans le transport :

```
    def data_received(self, data):
        self.logger.debug(f"received {data!r} from {self.address}")
        self.transport.write(data)
        self.logger.debug(f"sent {data!r}")
```

On rappelle que, lorsque l'on traite de donnée brute, échangée sur un réseau, par exemple, cette donnée est de l'octet. On rappelle aussi que le transport et le protocole ont une relation 1 à 1. En conséquence, la classe **`EchoServerProtocol`** est la classe qui s'occupe de la connexion avec un et un seul client et non pas LE serveur au sens où on l'entend couramment, c'est-à-dire celui qui écoute toutes les connexions nouvelles.

© Editions ENI - All rights reserved

En plus de cette méthode pour écouter les données reçues, il y a une méthode spéciale pour écouter les fins de communications, détectées par l'envoi d'un EOF (end of file, soit littéralement fin de fichier). La méthode qui suit est appelée lorsque le client envoie un tel signal :

```
    def eof_received(self):
        self.logger.debug("received EOF")
        if self.transport.can_write_eof():
            self.transport.write_eof()
```

On donne ici l'information au transport. Le protocole du client la recevra ainsi et pourra décider de la manière de la traiter. On notera que le protocole demande si le transport supporte cette instruction, car tous ne le font pas et que le protocole est agnostique, il ne connaît rien du détail du canal de communication derrière le transport, il sait simplement comment il doit utiliser le transport pour arriver à ses fins.
Voici maintenant comment gérer la perte de connexion. Cette fois-ci, on est dans le cas où l'on a détecté que la connexion cliente est terminée, soit parce qu'il s'est produit une erreur, soit parce qu'elle a fermé.

```
    def connection_lost(self, exc):
        if exc:
            self.logger.error(f"ERROR: {exc}")
        else:
            self.logger.debug("closing")
        super().connection_lost(exc)
```

Dans notre cas, on ne fait rien de particulier (le transport est notifié de la perte de connexion par le client, il ne faut pas le faire des deux côtés).
Enfin, voici comment créer le serveur :
– on récupère la boucle événementielle
– on crée une usine du protocole serveur à l'aide de **`loop.create_server`**
– on attache cette usine à la boucle événementielle

```
if __name__ == "__main__":
    event_loop = asyncio.get_running_loop()
    server_factory = event_loop.create_server(EchoServer,
*SERVER_ADDRESS)
    server = event_loop.run_until_complete(server_factory)
    main_logger.debug("starting up on {SERVER_ADDRESSES[0]} port {SER-
VER_ADDRESSES[1]}")
```

Revenons un peu sur ce qu'il se passe en arrière-plan : à chaque fois qu'une nouvelle connexion est créée, un nouvel objet du protocole **`EchoServerProtocol`** est créé. D'où le fait que l'on passe cette classe en paramètre, elle est créée autant de fois qu'il y aura de nouvelles connexions.
Le rôle du serveur est justement d'écouter sur son adresse, de détecter ces nouvelles connexions, d'utiliser l'usine et c'est tout. Par la suite, chaque client communiquera directement avec le protocole serveur via son propre socket.

Voici maintenant la partie finale :
- on fait tourner le serveur indéfiniment
- lorsqu'il se termine :
  - on ferme le serveur (objet de type **asyncio.base_events.Server**)
  - on attend qu'il soit effectivement fermé
  - on ferme la boucle événementielle
  - on quitte le programme

Voici ce que cela donne :

```
try:
    event_loop.run_forever()
except KeyboardInterrypt:
    pass
finally:
    main_logger.debug("closing server")
    server.close()
    event_loop.run_until_complete(server.wait_closed())
    main_logger.debug("closing event loop")
    event_loop.close()
```

Remarque

Ce code est fourni dans asyncio/tcp/server_1.py.

Nous en avons maintenant terminé avec le serveur. Il reste cependant une question majeure... Quid du transport ? En effet, nous n'avons nulle part spécifié que nous souhaitons faire du TCP. Donc, pourquoi avons-nous ici un serveur TCP ?
Précisons à nouveau que le protocole que nous avons créé ici est agnostique. Il pourrait parfaitement fonctionner avec autre chose que du TCP, tant que cette autre chose suit les mêmes règles.
Ce qui fait que le protocole associé au transport soit TCP est l'appel à **loop.create_server**. Cette méthode va, par défaut, choisir d'utiliser un socket de type TCP. Il est cependant possible de l'utiliser pour créer un autre type de socket ou pour gérer un protocole bâti au-dessus de TCP. Nous la réutiliserons plus loin pour gérer du SSL.
Voici maintenant le client. Nous passons sur le début du fichier :

```
import asyncio
import functools
import logging
import coloredlogs
import sys

SERVER_ADDRESS = ("localhost", 8040)

main_logger = logging.getLogger("main")
logger_format = "%(asctime)s,%(msecs)03d %(message)s"
coloredlogs.install(fmt=logger_format, level="DEBUG", logger=main_logger)
```

© Editions ENI - All rights reserved

L'adresse renseignée ici est celle du serveur, celle à laquelle il faudra se connecter (et donc la même que dans le fichier précédent).
Voici le protocole client, il définit lui aussi son propre protocole. On aura besoin pour ce client d'avoir accès à la boucle événementielle ainsi qu'à une future :

```
class EchoClient(asyncio.Protocol):

    def __init__(self, loop, future):
        super().__init__()
        self.loop = loop
        self.logger = logging.getLogger("EchoClient")
        coloredlogs.install(level="DEBUG", logger=self.logger)
        self.future = future
```

Voici la méthode qui va être appelée lorsque la connexion sera établie :

```
    def connection_made(self, transport):
        self.transport = transport
        self.address = transport.get_extra_info("peername")
        self.logger.debug(f"connecting to {self.address[0]} port
{self.address[1]}")
        self.task = event_loop.create_task(self.read_from_input())
```

Cette dernière va agréger le transport ainsi que l'adresse du serveur (pour des besoins d'affichage), puis elle va lancer une tâche : lire l'entrée standard.
Voici maintenant cette tâche :

```
    async def read_from_input(self):
        try:
            while True:
                msg = await ainput(self.loop, "? ")
                if msg:
                    self.transport.write(msg)
                    self.logger.debug(f"sending {msg!r}")
                else:
                    return
        except KeyboardInterrupt:
            print("Quitting client")
        finally:
            print("Exiting client")
            if self.transport.can_write_eof():
                self.transport.write_eof()
```

Cette tâche réalise une boucle infinie ou elle attend un message et dès qu'elle le reçoit, elle l'envoie au transport. Si le message est vide, alors elle ferme le client et envoie un EOF au serveur pour qu'il sache que sa connexion est perdue.
Comme la lecture de l'entrée standard est asynchrone, pendant que l'utilisateur tape son texte, d'autres événements peuvent se produire, tels que la réception d'une donnée du serveur, gérée par cette méthode :

```
    def data_received(self, data):
```

```
        self.logger.debug(f"received {data!r}")
```

On se contente d'afficher ce que le serveur nous renvoie.
Voici maintenant la fonction qui gère un EOF envoyé par le serveur :

```
    def eof_received(self):
        self.logger.debug("received EOF")
        self.transport.close()
        if not self.future.done():
            self.future.set_result(True)
```

Cette méthode va marquer la future comme terminée, afin de couper son attente (qui se produit plus bas dans le code), technique que l'on a introduit précédemment.
Voici maintenant comment gérer la perte de connexion :

```
    def connection_lost(self, exc):
        self.logger.debug("server closed connection")
        self.transport.close()
        if not self.future.done():
            self.future.set_result(True)
        super().connection_lost(exc)
```

Nous réalisons essentiellement deux actions : terminer le transport et la future.
Voici maintenant comment utiliser ce protocole :

```
if __name__ == "__main__":
    event_loop = asyncio.get_event_loop()

    client_completed = event_loop.create_future()

    client_factory = functools.partial(
        EchoClient,
        loop=event_loop,
        future=client_completed,
    )

    factory_coroutine = event_loop.create_connection(
        client_factory,
        *SERVER_ADDRESS,
    )
```

La première chose que nous faisons, comme toujours, consiste à récupérer la boucle événementielle. Puis, nous créons la future que nous allons passer à notre protocole, celle qui aura pour but de nous dire lorsque le client aura terminé.
Nous créons ensuite une factory via la fonction **`functools.partial`**, comme on a pu le voir précédemment, ce qui permet d'avoir une classe qui s'instancie sans paramètres.
Ensuite, nous créons notre coroutine via l'utilisation de **`loop.create_connection`**, cette fois-ci (ceci est une différence majeure avec le côté serveur).
Une fois que ce travail est fait, nous n'avons plus qu'à faire fonctionner la boucle événementielle en lui demandant de faire tourner la coroutine qu'est le client, puis attendre que la future soit terminée.

© Editions ENI - All rights reserved

```
    main_logger.debug("waiting for client to complete")
    try:
        event_loop.run_until_complete(factory_coroutine)
        event_loop.run_until_complete(client_completed)
    finally:
        main_logger.debug("closing event loop")
        event_loop.close()
```

Remarque

Ce code est fourni dans asyncio/tcp/client_1.py.

Enfin, nous pouvons fermer la boucle événementielle une fois que tout est terminé. Le code de lancement du serveur et du client qui est présenté ici est celui que vous êtes le plus susceptible de trouver. Néanmoins, avec Python 3.7 et ce qu'il introduit, nous pourrions procéder ainsi pour le client :

```
async def main():
    event_loop = asyncio.get_running_loop()

    client_completed = event_loop.create_future()

    client_factory = functools.partial(
        EchoClient,
        loop=event_loop,
        future=client_completed,
    )

    factory_coroutine = event_loop.create_connection(
        client_factory,
        *SERVER_ADDRESS,
    )

    main_logger.debug("waiting for client to complete")

    await factory_coroutine
    await client_completed


if __name__ == "__main__":
    asyncio.run(main())
```

Remarque

Ce code est fourni dans asyncio/tcp/client_2.py.

On utilise ici deux fois l'instruction **`await`** pour lancer la coroutine puis pour attendre la future.

Pour le serveur, c'est un peu plus subtil, de par la présence de **loop.run_forever**. Pour cela, il faut utiliser une solution qui n'est pas parfaite :

```
async def main():
    event_loop = asyncio.get_running_loop()
    server_factory = event_loop.create_server(EchoServerProtocol,
*SERVER_ADDRESS)
    server = await server_factory
    main_logger.debug(f"starting up on
{SERVER_ADDRESS[0]}:{SERVER_ADDRESS[1]}")

    try:
        await asyncio.sleep(1000000000)
    except KeyboardInterrupt:
        pass
    finally:
        main_logger.debug("closing server")
        server.close()
        await server.wait_closed()
        main_logger.debug("closing event loop")

if __name__ == "__main__":
    asyncio.run(main())
```

Remarque

Ce code est fourni dans asyncio/tcp/server_2.py.

En effet, le **sleep** utilisé avec un nombre très grand équivaut à un sleep infini, ce qui est le but recherché, au final.

Maintenant que nous avons présenté ces deux solutions, nous pouvons les améliorer encore un peu. D'une part côté serveur, pour faire en sorte de mieux gérer le fait de quitter le serveur par un [Ctrl] **C** :

```
async def main():
    event_loop = asyncio.get_running_loop()
    server_factory = event_loop.create_server(EchoServerProtocol,
*SERVER_ADDRESS)
    server = await server_factory
    main_logger.debug(f"starting up on {SERVER_ADDRESSES[0]} port
{SERVER_ADDRESSES[1]}")

    future = event_loop.create_future()
    event_loop.add_signal_handler(signal.SIGINT, future.set_result, None)
    try:
        await future
    finally:
        main_logger.debug("closing server")
        event_loop.remove_signal_handler(signal.SIGINT)
        server.close()
```

© Editions ENI - All rights reserved

```
        event_loop.run_until_complete(server.wait_closed())
        main_logger.debug("closing event loop")
        event_loop.close()
```

Nous réutilisons ici les signaux, et nous utilisons la future qui est liée au déclenchement de ce signal pour faire attendre le serveur. Cette méthode a aussi l'avantage de nous débarrasser du **sleep** qui manquait d'élégance.
D'autre part, côté client, nous pouvons réécrire notre client pour lui permettre de ne pas quitter lorsque la personne ne renvoie rien, mais de ne quitter que lors de la détection du [Ctrl] **C**.
Ceci se fait très simplement. D'une part, nous ne quittons plus lorsque l'utilisateur ne saisit rien, nous envoyons le message vide, autrement dit, il y a deux lignes de code qui ont disparu dans le code suivant :

```
    async def read_from_input(self):
        try:
            while True:
                msg = await ainput(self.loop, "?")
            self.transport.write(msg)
            self.logger.debug(f"sending '{msg!r}'")
        except KeyboardInterrupt:
            print("Quitting client")
        finally:
            print("Exiting client")
            if self.transport.can_write_eof():
                self.transport.write_eof()
```

D'autre part, nous allons connecter le signal SIGINT à la future déjà existante, ce qui nous permet d'éviter de faire un code trop complexe :

```
async def main():
    event_loop = asyncio.get_running_loop()
    client_completed = event_loop.create_future()
    client_factory = functools.partial([...])
    factory_coroutine = event_loop.create_connection([...])

    event_loop.add_signal_handler(signal.SIGINT,
client_completed.set_result, True)

    main_logger.debug("waiting for client to complete")

    await factory_coroutine
    try:
        await client_completed
    finally:
        event_loop.remove_signal_handler(signal.SIGINT)
```

On place ensuite l'instruction **await** dans une boucle **try** et dans la boucle **finally**, on va supprimer le gestionnaire de signaux.

Remarque

Ce code est fourni dans asyncio/tcp/server_3.py et asyncio/tcp/client_3.py.

On a donc maintenant deux fichiers qui sont compatibles avec la syntaxe Python 3.7 et qui fonctionnent très bien, en gérant la communication et la fin des programmes.
Un exercice proposé est de faire un serveur de chat. Pour cela, le client demande le pseudo de l'utilisateur avant de créer le protocole, puis ceci est envoyé en tant que premier message. Lorsque ce message est reçu par le serveur, ce dernier l'enregistre. Le serveur doit trouver un moyen de faire en sorte que, lorsqu'un client envoie un message, ce dernier soit reçu par tous les clients.

Remarque

Ce code est fourni dans asyncio/tcp/server_4.py et asyncio/tcp/client_4.py.

1.6 Exemple pour le protocole SSL

Le module **asyncio** supporte également de façon native le protocole SSL. Pour l'activer, il faut faire exactement comme ce qui précède, en passant un argument supplémentaire, le contexte SSL.

Remarque

Pour plus d'informations sur le SSL, vous pouvez lire la section qui y est consacrée dans le chapitre sur la programmation réseau.

Ce contexte SSL est géré par deux éléments : un certificat et une clé. Tous deux sont générés par la commande suivante :

```
$ openssl req -newkey rsa:2048 -nodes -keyout my.key -x509 -days 365
-out my.crt
```

Remarque

Ces certificats sont ce que l'on appelle des certificats auto-générés. Dans un contexte web, ils ne valent pas grand-chose, car ils ne sont pas associés à un niveau de confiance que l'on est en droit d'attendre d'un organisme de certification officiel. Il faudra donc passer par ces organismes pour les acheter, voire par votre hébergeur. Par contre, pour réaliser des tests, ils sont largement suffisants et de ce point de vue-là, ils sont en tout point identiques à de vrais certificats.

Nous allons commencer par le commencement : le contexte SSL. Voici les modifications à apporter au serveur par rapport au fichier précédent (le fichier server_3.py).
D'abord, créer le contexte :

```
    ssl_context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
    ssl_context.check_hostname = False
    ssl_context.load_cert_chain("my.crt", "my.key")
```

© Editions ENI - All rights reserved

Puis l'utiliser :

```
    server_factory = event_loop.create_server(EchoServerProtocol,
*SERVER_ADDRESS, ssl=ssl_context)
```

Pour le client, ce sont à peu près des changements similaires qu'il faudra apporter. D'abord, la création du contexte, semblable à ce qui précède, à quelques petites variations près.

```
ssl_context = ssl.create_default_context(ssl.Purpose.SERVER_AUTH)
ssl_context.check_hostname = False
ssl_context.load_verify_locations('my.crt')
```

Puis l'utilisation de ce contexte :

```
factory_coroutine = event_loop.create_connection(
    client_factory,
    *SERVER_ADDRESS,
    ssl=ssl_context,
)
```

On aurait accompli là tout le travail, si ce n'était le fait que le protocole SSL est différent du protocole TCP et qu'il ne supporte pas l'EOF.
Par conséquent, il faut gérer par nous-mêmes la fin des connexions.
Pour le client d'abord :
Si le client valide la saisie sans rien avoir entré, alors on quitte le client et par conséquent, il faut fermer la connexion. Ceci est géré ainsi :

```
async def read_from_input(self):
    try:
        while True:
            msg = await ainput(self.loop, "?")
            if not msg:
                self.transport.write(b"\x00")  # send EOF
            else:
                self.transport.write(msg)
            self.logger.debug(f"sending '{msg!r}'")
    except KeyboardInterrupt:
        print("Quitting client")
    finally:
        print("Exiting client")
```

On envoie l'octet **\x00** pour signifier la fin de la connexion au serveur.
Il faut aussi retirer la fermeture du transport côté client :

```
def eof_received(self):
    self.logger.debug("received EOF")
    if not self.future.done():
        self.future.set_result(True)
```

Au niveau du serveur, il faut faire ces modifications :

```
def data_received(self, data):
    self.logger.debug(f"received {data!r} from {self.address}")
    terminate = data.endswith(b"\x00")
```

```
        if terminate:
            data = data.rstrip(b"\x00")
            print(f"Terminate detected, closing transport soon")
        if data:
            self.transport.write(data)
            self.logger.debug(f"sent {data!r}")
        if terminate:
            self.eof_received()
```

On vérifie la présence de l'octet **\x00** à la fin de la séquence d'octets pour savoir si l'on doit simplement renvoyer le message ou s'il faut terminer la connexion, auquel cas on appelle simplement la méthode **eof_received**.
Il faut aussi modifier cette méthode pour rajouter la fermeture du transport qui est maintenant gérée côté serveur au lieu de côté client :

```
    def eof_received(self):
        self.logger.debug(f"EOF received for {self.address}")
        self.transport.close()
```

Jusque-là, tout va bien, le client et le serveur fonctionnent et lorsque, avec le client, vous ne saisissez rien, alors le client quitte proprement et c'est bien géré côté serveur. Par contre, le fait de quitter le client ou le serveur via un [Ctrl] **C** mène à une erreur.
Pour la résoudre, il faut rajouter un nouveau gestionnaire de signaux et lui faire fermer proprement les transports. Pour le serveur, il faut avoir un moyen d'accéder à toutes les instances. Pour cela, on met en place ceci :

```
class EchoServerProtocol(asyncio.Protocol):

    instances = set()

    def connection_made(self, transport):
        self.instances.add(self)
        [...]

    def connection_lost(self, exc):
        self.instances.discard(self)
        [...]
```

Maintenant que nous avons accès à toutes les instances disponibles et donc toutes les connexions en cours, nous pouvons créer une méthode de classe pour gérer la fin de toutes les connexions :

```
    @classmethod
    def eof_received_for_all_instances(cls):
        for instance in cls.instances:
            instance.eof_received()
```

Il ne reste plus qu'à utiliser cette méthode de classe :

```
    event_loop.add_signal_handler(
        signal.SIGINT,
        EchoServerProtocol.eof_received_for_all_instances)
```

© Editions ENI - All rights reserved

Nous pouvons maintenant fermer le serveur proprement et voir tous les clients s'arrêter. En ce qui concerne le client, il est possible de faire exactement la même chose. Bien entendu, le client n'est lancé qu'une seule fois et, pour chaque programme Python différent lançant ce client, il n'y aura qu'une et une seule instance. Une autre solution aurait consisté à transformer ce client en singleton et utiliser cette propriété.

1.7 Exemple pour le protocole UDP

Les protocoles UDP et TCP sont les protocoles réseau les plus utilisés.

Remarque

Pour plus d'informations sur le protocole UDP et sa mise en œuvre dans un cadre non asynchrone, vous pouvez vous reporter au chapitre Programmation réseau.

Toutefois, le protocole UDP est très différent du TCP. Cela se voit donc naturellement dans les classes. Voici la classe serveur :

```
class EchoServerProtocol(asyncio.Protocol):

    def connection_made(self, transport):
        self.transport = transport
        self.logger = logging.getLogger(" EchoServerProtocol")
        coloredlogs.install(level="DEBUG", logger=self.logger)
        self.logger.debug("connection made")
```

Jusque-là, pas de différences avec le protocole TCP, hormis le fait que l'on ne va plus lire **peername**, car il est à **None** dans ce contexte.

Dans le protocole TCP, la connexion est établie par le client et elle dure jusqu'à ce qu'elle soit fermée. Il y a donc un couple (transport, protocole) pour chaque client qui a établi une connexion.

Dans le protocole UDP, la connexion est établie à chaque dialogue. En ce qui nous concerne, un protocole et un transport sont créés dès le lancement du serveur et ce sont ces deux objets qui vont gérer les connexions en provenance de tous les clients.

L'autre méthode essentielle à implémenter est celle-ci :

```
    def datagram_received(self, data, addr):
        self.logger.debug(f"received {data.decode()!r} from {addr}")
        self.transport.sendto(data, addr)
        self.logger.debug(f"sent {data.decode()!r}")
```

Au passage, notons que le nom de la méthode est différent, tout comme le fait qu'il y ait un paramètre supplémentaire, de la même manière que le protocole UDP utilise **socket.recv_from** là ou le protocole TCP utilise **socket.recv**. Par ailleurs, nous n'utilisons pas **transport.write**, mais **transport.sendto**, de la même manière que le socket UDP utilise **socket.sentto** au lieu de **socket.send** pour TCP.

On n'a pas besoin de **connexion_lost**, car le protocole étant déconnecté, il n'y a pas moyen de savoir lorsque le client a terminé de nous envoyer des requêtes. La connexion en tant que telle est établie à chaque échange et se termine immédiatement.
Vous pouvez cependant la conserver pour constater qu'elle ne sera jamais appelée.
La coroutine **main** est elle aussi modifiée en profondeur. On utilise maintenant **event_loop.create_datagram_endpoint** en y passant l'adresse **locale**.
Lorsque l'on attend l'objet ainsi créé, on obtient le couple transport et protocole et comme nous ne manipulerons qu'une seule instance de ces deux objets, ils peuvent être manipulés depuis la coroutine **main**.
On garde notre système d'arrêt au signal d'interruption et lorsque l'on quitte, on ferme à ce moment-là le transport, ce qui simplifie les choses par rapport au TCP.

```
async def main():
    event_loop = asyncio.get_running_loop()

    server_factory = event_loop.create_datagram_endpoint(
        EchoServerProtocol, local_addr=SERVER_ADDRESS)
    transport, protocol = await server_factory
    main_logger.debug(f"starting up on
{SERVER_ADDRESS[0]}:{SERVER_ADDRESS[1]}")

    future = event_loop.create_future()
    event_loop.add_signal_handler(signal.SIGINT, future.set_result, None)


    try:
        await future
    finally:
        main_logger.debug("closing server")
        event_loop.remove_signal_handler(signal.SIGINT)
        transport.close()
```

Voici maintenant le client :

```
class EchoClientProtocol(asyncio.Protocol):

    def __init__(self, loop, future):
        super().__init_?)
        self.loop = loop
        self.logger = logging.getLogger("EchoClient")
        coloredlogs.install(fmt=logger_format, level="DEBUG",
logger=self.logger)
        self.future = future
```

On ne constate aucun changement par rapport à TCP dans la méthode d'initialisation, tout comme pour la fonction **connection_made** qui va elle aussi récupérer le **peername**.

```
    def connection_made(self, transport):
        self.transport = transport
```

© Editions ENI - All rights reserved

```
        self.address = transport.get_extra_info("peername")
        self.logger.debug(f"connecting to
{self.address[0]}:{self.address[1]}")
        self.task = self.loop.create_task(self.read_from_input())
```

En effet, le **peername** désigne là l'adresse sur laquelle communiquer. L'agréger n'est pas particulièrement respectueux de la norme, mais comme nous n'aurons qu'un seul transport et qu'un seul protocole, nous décidons de nous en affranchir pour une simple raison, on en a besoin pour la méthode suivante :

```
    async def read_from_input(self):
        try:
            while True:
                msg = await ainput(self.loop, "?")
                if not msg:
                    self.transport.close()
                else:
                    self.transport.sendto(msg, self.address)
                self.logger.debug(f"sending '{msg!r}'")
        except KeyboardInterrupt:
            print("Quitting client")
        finally:
            print("Exiting client")
```

Ici, nous avons besoin de l'adresse pour utiliser **transport.sendto**. Et cette adresse ne peut pas être facilement passée à la tâche indépendante. La solution de l'agrégation de l'adresse trouve donc son utilité ici.
Voici maintenant ce que l'on fait lorsque l'on reçoit une donnée (on l'affiche) :

```
    def datagram_received(self, data, addr):
        self.logger.debug(f"received {data.decode()!r}")
```

Puis ce que l'on fait en cas d'erreur (comme par exemple lorsque l'on envoie un message à un serveur éteint) :

```
    def error_received(self, exc):
        print(f" Error received {exc!r}")
```

Enfin, voici la coroutine **main** qui, cette fois-ci, utilise **event_loop.create_datagram_endpoint**, tout comme le serveur, mais en passant **remote_addr** (au lieu de **local_addr**). Pour le reste, le code est relativement similaire à celui du serveur.

```
async def main():
    event_loop = asyncio.get_running_loop()

    client_completed = event_loop.create_future()

    server_factory = event_loop.create_datagram_endpoint(
        lambda: EchoClientProtocol(event_loop, client_completed),
        remote_addr=SERVER_ADDRESS)

    transport, protocol = await server_factory
```

```
    event_loop.add_signal_handler(signal.SIGINT, client_completed.set_re-
sult, True)

    main_logger.debug("waiting for client to complete")

    try:
        await client_completed
    finally:
        event_loop.remove_signal_handler(signal.SIGINT)
        transport.close()
```

Notons que comme nous avons dans notre protocole un constructeur qui a besoin de paramètres, nous utilisons ici une lambda expression pour créer une factory à la volée.
Avec le protocole UDP, il est possible de faire tourner le client alors que le serveur est éteint : lors de l'envoi d'un message, une erreur sera levée, c'est tout. On peut donc ouvrir et fermer le serveur et quelques clients à volonté et tout devrait fonctionner à merveille.

Remarque

Le code source pour ces exemples est dans asyncio.udp.

1.8 Traiter d'autres types de protocoles réseau

Le TCP/IP et l'UDP sont les deux protocoles massivement utilisés. Les sockets utilisés sont respectivement :

```
sock_tcp = socket.sock(socket.AF_INET, socket.SOCK_STREAM)
sock_udp = socket.sock(socket.AF_INET, socket.SOCK_DRAM)
```

Si vous voulez utiliser un socket personnalisé, vous pouvez utiliser les méthodes **`create_server`**, **`create_connection`** ou encore **`create_datagram_endpoint`** en y passant un socket que vous auriez créé par vous-même.
Il vous faudra ensuite travailler sur le protocole pour gérer les spécificités de la norme que vous souhaitez implémenter.
Voici quelques exemples de sockets que l'on pourrait éventuellement chercher à adapter, à commencer par le socket Bluetooth :

```
s = socket.socket(socket.AF_BLUETOOTH, socket.SOCK_STREAM,
                  socket.BTPROTO_RFCOMM)
```

Ou encore un socket pour le CAN Bus :

```
s = socket.socket(socket.AF_CAN, socket.SOCK_DGRAM, socket.CAN_ISOTP)
```

Encore une fois, ce sera le rôle d'une bibliothèque d'implémenter un transport et un protocole pour une telle utilisation, sachant que les normes associées sont relativement complexes. Votre rôle en tant que développeur d'application sera simplement d'utiliser cette bibliothèque.

© Editions ENI - All rights reserved

1.9 Exemple pour Subprocess

La bibliothèque subprocess est la bibliothèque Python permettant de faire des appels externes. Elle permet d'appeler un programme de la même manière que nous le ferions depuis un terminal.

Remarque

Pour plus d'informations sur les appels externes, rendez-vous dans la section qui y est consacrée, dans le chapitre sur la programmation système.

Nous allons faire un appel à la commande **ls** et pour cela, nous allons mettre en place un protocole dédié au traitement de cette commande.

Remarque

Vous pourrez trouver cet exemple dans le fichier asyncio/subprocess/subprocess_protocol.py

Voici les premières lignes de ce fichier qui montrent les imports ainsi que la création de l'outil de logging :

```
import asyncio
import functools
import logging
import coloredlogs

main_logger = logging.getLogger("main")
logger_format = "%(asctime)s,%(msecs)03d %(message)s"
coloredlogs.install(fmt=logger_format, level="DEBUG", logger=main_logger)
```

Voici maintenant la classe protocole :

```
class LSProtocol(asyncio.SubprocessProtocol):
    FD_NAMES = ["stdin", "stdout", "stderr"]
```

Cet attribut de classe n'est pas utile en soi, si ce n'est pour aider à l'affichage, nous le verrons plus bas.
Pour l'initialisation, on agrège la future, une technique que l'on a vue déjà plusieurs fois et on crée un buffer qui aura pour rôle de récupérer les résultats de la commande au fur et à mesure qu'ils arrivent.

```
    def __init__(self, done_future):
        self.done = done_future
        self.buffer = bytearray()
        super().__init__()
```

Comme pour la communication via un socket, nous avons cette méthode **connection_made** qui permet de mettre en place un canal de communication entre le sous-processus et notre classe. Le transport est de type **BaseSubprocessTransport**. À travers ce transport, nous pourrions à la fois lire ce que le sous-processus nous renvoie, mais aussi écrire dedans, pour répondre, par exemple, à des questions posées par ce sous-processus.

Ici, nous nous contentons d'agréger le transport :

```
def connection_made(self, transport):
    main_logger.debug(f"process started {transport.get_pid()}")
    self.transport = transport
```

Cette méthode est une fonction de rappel utilisée à chaque fois que des octets arrivent en provenance du sous-processus. Elle prend deux paramètres, à savoir **fd** pour file descriptor, c'est-à-dire la provenance, et **data**, la donnée transmise.

```
def pipe_data_received(self, fd, data):
    main_logger.debug(
        f"read {len(data)} bytes from {fd}({self.FD_NAMES[fd]})")
    if fd == 1:
        self.buffer.extend(data)
```

La valeur de **fd** est :

- 0 pour l'entrée standard
- 1 pour la sortie standard
- 2 pour la sortie d'erreur

Ce sont les indices de la liste **FD_NAMES** écrite plus haut. Le rôle de cette liste est de faciliter l'affichage. Dans la fonction, nous testons que **fd** vaut 1, car nous ne sommes intéressés que par le résultat de la sortie standard.

Voici maintenant la fonction de rappel appelée lorsque le sous-processus quitte :

```
def process_exited(self):
    main_logger.debug("process exited")
    return_code = self.transport.get_returncode()
    main_logger.debug(f"return code {return_code}")
    if not return_code:
        cmd_output = bytes(self.buffer).decode()
        results = self._parse_results(cmd_output)
    else:
        results = []
    self.done.set_result((return_code, results))
```

À ce stade, nous avons le buffer qui contient l'ensemble des données. Il est donc exploitable. Nous avons aussi accès au code de retour, à travers le transport. Nous allons donc le tester et en fonction de ceci, soit parser le résultat, soit renvoyer un résultat vide. Ce résultat est mis dans la future. La fonction suivante est dédiée au traitement des résultats. Dans notre cas, ce traitement est très simple :

```
def _parse_results(self, output):
    main_logger.debug("parsing results")
    return output.splitlines()
```

Voici maintenant la fonction principale. Celle-ci va :

- récupérer la boucle événementielle courante
- créer la future qui nous permettra à la fois d'attendre le résultat et de récupérer sa valeur
- créer la procédure à l'aide de **event_loop.subprocess_exec**

© Editions ENI - All rights reserved

– attendre cette procédure pour récupérer le transport et le protocole
– attendre la future
– afficher le résultat
– fermer le transport

```
async def main():
    event_loop = asyncio.get_running_loop()
    cmd_done = asyncio.Future(loop=event_loop)
    factory = functools.partial(LSProtocol, cmd_done)
    proc = event_loop.subprocess_exec(
        factory,
        "ls",  # "existe_pas",
        stdin=None,
        stderr=None,
    )
    try:
        main_logger.debug("launching process")
        transport, protocol = await proc
        main_logger.debug("waiting for process to complete")
        await cmd_done
    except Exception as e:
        main_logger.error(f"an exception occurs {e!r}")
    else:
        return_code, results = cmd_done.result()
        if return_code:
            main_logger.error(f"error exit {return_code}")
        else:
            for i, r in enumerate(results):
                print(f"{i:2}: {r}")

    finally:
        transport.close()
```

Le programme se termine ainsi, pour qu'il soit appelable :

```
if __name__ == "__main__":
    asyncio.run(main())
```

Voici maintenant un exemple d'appel du programme :

```
$ python subprocess_protocol.py
2019-03-06 12:37:11,879 launching process
2019-03-06 12:37:11,880 process started 8013
2019-03-06 12:37:11,880 waiting for process to complete
2019-03-06 12:37:11,881 read 36 bytes from 1(stdout)
2019-03-06 12:37:11,881 process exited
2019-03-06 12:37:11,881 return code 0
2019-03-06 12:37:11,881 parsing results
 0: fonctions.py
 1: subprocess_protocol.py
```

Comme précédemment, nous ne voyons pas directement l'instanciation du transport, mais ce dernier est bien un transport adapté à **subprocess** car nous avons utilisé **event_loop.subprocess_exec**, la fonction qui va l'instancier.
Maintenant que la technique générale pour réaliser ce genre de tâche est vue, il vous est possible de créer des classes protocoles pour gérer des appels plus complexes à **ls** ou encore à tout autre type de commande. Il est à noter que si la commande prend plusieurs arguments, ceux-ci sont passés de manière positionnelle, comme lors d'un appel à **subprocess.run**, sans le **shell=True**.
Vous pouvez d'ailleurs voir que dans le code, il y a un commentaire **# "existe_pas"**. Vous pouvez donc voir ou rajouter des éléments supplémentaires. Au passage, je vous encourage à supprimer ce commentaire et exécuter à nouveau le fichier pour voir son comportement dans des cas d'erreur.
Outre cette technique, il est aussi possible d'utiliser une autre façon de faire, assez haut niveau, permettant de se rapprocher de l'utilisation classique des fonctions du module **subprocess**. Elle est de haut niveau, car on n'a pas besoin de gérer le protocole ou le transport, tous ces détails sont gérés de manière transparente.
Voici l'exemple tiré de la documentation officielle :

```
>>> async def run(cmd):
...     proc = await asyncio.create_subprocess_shell(
...         cmd,
...         stdout=asyncio.subprocess.PIPE,
...         stderr=asyncio.subprocess.PIPE)
...
...     stdout, stderr = await proc.communicate()
...
...     print(f'[{cmd!r} exited with {proc.returncode}]')
...     if stdout:
...         print(f'[stdout]\n{stdout.decode()}')
...     if stderr:
...         print(f'[stderr]\n{stderr.decode()}')
...
```

Lorsque ce programme est appelé avec une commande qui va fonctionner, on obtient :

```
>>> asyncio.run(run('ls'))
['ls' exited with 0]
[stdout]
client_1.py
server_1.py
```

Lorsqu'il est appelé avec une commande qui ne va pas fonctionner, on obtient :

```
>>> asyncio.run(run('ls existe_pas'))
['ls existe_pas' exited with 2]
[stderr]
ls: impossible d'accéder à 'existe_pas': Aucun fichier ou dossier de ce type
```

© Editions ENI - All rights reserved

Cet exemple utilise **asyncio.create_subprocess_shell** (donc équivalent à utiliser **shell=True** dans les fonctions **subprocess**). Cette méthode est de haut niveau et elle s'accorde particulièrement bien à l'asynchrone tel qu'il existe pour Python 3.7.
À partir du code fourni par la documentation officielle, on peut s'amuser à recréer les fonctions essentielles de **subprocess** en utilisant l'asynchrone :

```
>>> async def getstatutoutput(cmd):
...     proc = await asyncio.create_subprocess_shell(
...         cmd,
...         stdout=asyncio.subprocess.PIPE,
...         stderr=asyncio.subprocess.PIPE)
...
...     stdout, stderr = await proc.communicate()
...
...     if not stdout:
...         result = stderr.strip()
...     elif not stderr:
...         result = stdout.strip()
...     else:
...         result = stdout + b"\n" + stderr
...
...     return proc.returncode, result.decode()
...
```

Voici ce que cela donne en cas de succès :

```
>>> asyncio.run(getstatutoutput('ls'))
(0, 'client_1.py\nserver_1.py\n')
```

Et en cas d'erreur :

```
>>> asyncio.run(getstatutoutput('ls existepas'))
(2, "ls: impossible d'accéder à 'existepas': Aucun fichier ou dossier de
ce type\n")
```

Ces appels sont rigoureusement identiques à **subprocess.getstatutoutput**.
Voici maintenant pour **getoutput** :

```
>>> async def getoutput(cmd):
...     proc = await asyncio.create_subprocess_shell(
...         cmd,
...         stdout=asyncio.subprocess.PIPE,
...         stderr=asyncio.subprocess.PIPE)
...
...     stdout, stderr = await proc.communicate()
...
...     if not stdout:
...         result = stderr.strip()
...     elif not stderr:
...         result = stdout.strip()
...     else:
...         result = stdout + b"\n" + stderr
```

```
...
...     return result.decode()
...
```

Comme vous pouvez le constater, la fonction est quasiment identique à ce qui précède. Voici maintenant les deux appels de test :

```
>>> asyncio.run(getoutput('ls'))
'client_1.py\nserver_1.py'
>>> asyncio.run(getoutput('ls existepas'))
"ls: impossible d'accéder à 'existepas': Aucun fichier ou
dossier de ce type"
```

Là encore, dans les deux cas, on obtient le même résultat. On peut maintenant passer à la fonction **check_output** qui est légèrement plus délicate. On va en effet réutiliser l'exception réellement générée par la fonction d'origine :

```
>>> from subprocess import CalledProcessError
```

Il ne nous reste plus qu'à la déclencher au bon moment :

```
>>> async def check_output(cmd):
...     proc = await asyncio.create_subprocess_shell(
...         cmd,
...         stdout=asyncio.subprocess.PIPE,
...         stderr=asyncio.subprocess.PIPE)
...
...     stdout, stderr = await proc.communicate()
...
...     if proc.returncode != 0:
...         raise CalledProcessError(proc.returncode, cmd)
...
...     if not stdout:
...         result = stderr.strip()
...     elif not stderr:
...         result = stdout.strip()
...     else:
...         result = stdout + b"\n" + stderr
...
...     return result.decode()
...
```

Testons cette dernière fonction, en cas de succès :

```
>>> asyncio.run(check_output('ls'))
'client_1.py\nserver_1.py'
```

Puis, ce qui nous intéresse vraiment, en cas d'erreur :

```
>>> asyncio.run(check_output('ls existepas'))
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    asyncio.run(check_output('ls existepas'))
  File "/usr/lib/python3.7/asyncio/runners.py", line 43, in run
```

© Editions ENI - All rights reserved

```
    return loop.run_until_complete(main)
  File "/usr/lib/python3.7/asyncio/base_events.py", line 584, in
run_until_complete
    return future.result()
  File "<input>", line 10, in check_output
    raise CalledProcessError(proc.returncode, cmd)
subprocess.CalledProcessError: Command 'ls existepas' returned
non-zero exit status 2.
```

Là encore, le résultat est strictement identique à la méthode d'origine.
Ces méthodes sont directes et très faciles à utiliser une fois que l'on a compris leur rôle.

2. Flux

2.1 Introduction

Les flux sont un outil de plus bas niveau que les protocoles et les transports que nous avons présentés précédemment. Ils permettent de manipuler directement les données échangées. Un flux est unidirectionnel et peut être soit en lecture, soit en écriture. Pour avoir l'équivalent d'un transport entre, par exemple, un client et un serveur, il faut mettre en place un flux qui soit en lecture pour le client et en écriture pour le serveur ainsi qu'un autre flux qui soit en écriture pour le client et en lecture pour le serveur.
Il est conseillé d'utiliser préférentiellement les protocoles, mais dans certains cas, l'utilisation d'un simple flux peut s'avérer utile. Nous allons donc présenter la manière de les utiliser.

2.2 Exemple avec le protocole TCP

Nous allons ici écrire un nouveau serveur TCP qui renvoie ce qu'il reçoit, mais en utilisant les flux. Voici la fonction qui effectue ce travail :

```
async def echo(reader, writer):
    address = writer.get_extra_info("peername")
    logger = logging.getLogger(f"Client {address}")
    coloredlogs.install(fmt="%(name)s: %(message)s", level="DEBUG",
logger=logger)
    logger.debug("connection made {address}")

    while True:
        data = await reader.read(128)
        if data:
            logger.debug(f"received {data!r}")
            writer.write(data)
            await writer.drain()
            logger.debug(f"sent {data!r}")
        else:
            logger.debug(f"{address} closing")
            writer.close()
            return
```

On voit que cette fonction est en deux parties : la première partie correspond au travail qui est fait dans la méthode **connection_made** du protocole, la seconde partie est une boucle infinie qui contient tout ce qui était dans les méthodes **data_received**, **eof_received** et **connection_lost**.
Ce code est donc plus court, peut-être plus simple à comprendre lorsqu'on n'est pas familier avec les fonctions de rappel, mais moins dissocié.
On voit que cette méthode reçoit les deux flux en paramètre, ils sont donc créés en amont de l'appel de cette fonction. Ici, on ne fait que les utiliser. Le reader est une instance **StreamReader** et dispose de la méthode **read** qui est un awaitable, alors que le writer est une instance **StreamWriter** et dispose de la méthode **write**.
Une fois que l'on a terminé d'écrire, il faut attendre **writer.drain()** qui permet de s'assurer que tout a bien été envoyé, puis à **writer.close()** pour fermer le flux et ainsi terminer la connexion.
Ce qui va nous intéresser maintenant est la création de ces deux flux :

```
event_loop = asyncio.get_running_loop()
    server_factory = asyncio.start_server(echo, *SERVER_ADDRESS)
    server = await server_factory
```

Ici, l'appel à **asyncio.start_server** va créer les deux flux et les passer en paramètre à notre fonction. Le reste du code reste inchangé par rapport à l'utilisation d'un protocole.
Voyons maintenant pour le client :

```
reader, writer = await asyncio.open_connection(*address)
```

Les deux flux sont créés à l'aide de **asyncio.open_connection**. On peut, à partir de là, gérer séparément ces deux flux et c'est d'ailleurs leur intérêt principal :

```
async def main():
    event_loop = asyncio.get_running_loop()
    client_completed = event_loop.create_future()
    event_loop.add_signal_handler(signal.SIGINT,
                                  client_completed.set_result, True)
    reader, writer = await asyncio.open_connection(*SERVER_ADDRESS)
    task_input = event_loop.create_task(
        read_from_input(client_completed, writer))
    task_output = event_loop.create_task(
        print_from_server(client_completed, reader))
    event_loop.add_signal_handler(signal.SIGINT, task_input.cancel)
    try:
        await asyncio.gather(
            client_completed,
            task_input,
            task_output,
            return_exceptions=True,
        )
    finally:
        event_loop.remove_signal_handler(signal.SIGINT)
        writer.close()
```

© Editions ENI - All rights reserved

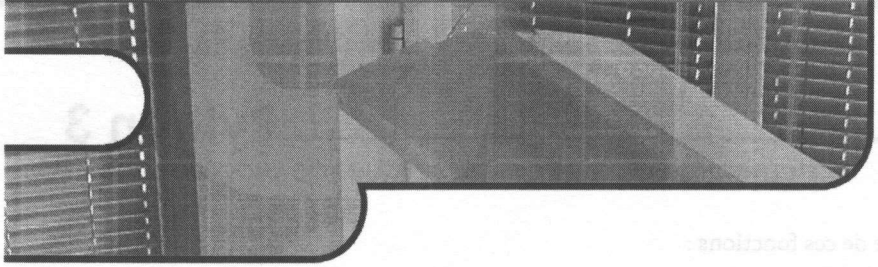
Ici, nous avons utilisé une future pour nous permettre de gérer la fin du programme, puis une tâche pour gérer l'arrivée de données depuis le serveur et une autre tâche pour gérer l'écriture de données à envoyer au serveur.
Il est possible de terminer le programme soit par [Ctrl] **C**, soit en écrivant une chaîne vide.
Voici maintenant la tâche permettant de lire les informations en provenance du clavier de l'utilisateur client :

```
async def read_from_input(client_completed, writer):
    try:
        while True:
            msg = await ainput("?")
            writer.write(msg)
            if not msg:
                return
    finally:
        print("Exiting client")
        if writer.can_write_eof():
            writer.write_eof()
        await writer.drain()
        client_completed.set_result(True)
        return
```

On s'assure que, quoi qu'il se passe, le **`writer`** est bien vidé et on termine la future si l'on n'a plus rien à écrire au serveur.
Voici maintenant la gestion des retours du serveur :

```
async def print_from_server(client_completed, reader):
    while True:
        data = await reader.read(128)
        if data:
            main_logger.debug(f"{data.decode()!r}")
        else:
            main_logger.debug("server closed connection")
            return
```

Ce code est assez direct : on attend la donnée et lorsqu'elle arrive, on l'affiche.



Chapitre 3.4
Programmation asynchrone : alternatives

1. Gevent

1.1 Introduction

La bibliothèque Gevent permet de créer en Python des coroutines pour gérer efficacement des problématiques réseau. Elle est basée sur l'utilisation de greenlet et une boucle événementielle performante (libev ou libuv). L'installation se fait ainsi :

```
$ pip install gevent
```

La bibliothèque contient donc un objet socket qui travaille de manière concurrente et quelques fonctions utiles, comme le montre cet exemple adapté de la documentation officielle :

```
>>> from gevent import socket, spawn, joinall
>>> urls = ["eni.fr", "www.inspyration.fr", "www.python.org"]
>>> jobs = [spawn(socket.gethostbyname, url) for url in urls]
>>> joinall(jobs, timeout=10)
>>> [job.value for job in jobs]
['185.42.28.200', '213.246.53.26', '151.101.40.223']
```

Dans cet exemple, on va demander l'IP associée à un nom d'hôte et on va créer plusieurs tâches pour chaque hôte, pour les mettre en concurrence.

La fonction **joinall** permet d'attendre que toutes les tâches soient terminées. La dernière ligne permet de récupérer le résultat de chaque tâche.

1.2 DNS

Gevent inclut les outils nécessaires pour la résolution de nom de domaine, comme l'exemple précédent en est une illustration. Il s'agit d'une réécriture iso-fonctionnelle partielle des fonctionnalités présentes dans le module python **socket**. Vous pourrez donc consulter la documentation officielle pour savoir comment ces fonctions doivent être utilisées et ce qu'elles retournent.

Voici la liste de ces fonctions :

- get_hostbyname
  - prend en paramètre un nom de domaine
  - renvoie une IPv4
  - fonction utilisée dans l'exemple ci-dessus
- get_hostbyname_ex
  - prend en paramètre un nom de domaine
  - renvoie un 3-uplet (nom de domaine, une liste d'aliases et une liste d'IPv4)
- getaddrinfo
  - prend en paramètre un hôte et un port et 4 filtres permettant de diminuer la liste des résultats suivant ce que l'on veut réellement
  - renvoie tous les paramètres nécessaires pour créer un socket (5-uplet)
- gethostbyaddr
  - prend en paramètre une IP
  - renvoie un 3-uplet (IP, une liste d'aliases et une liste d'IPv4)
- ·getnameinfo
  - prend une adresse socket et des drapeaux
  - renvoie un 2-uplet (hôte, port)
  - en fonction des drapeaux, l'hôte est un nom de domaine ou une IP.

1.3 Appels système

Le module **`gevent.subprocess`** est une réécriture iso-fonctionnelle partielle du module subprocess et contient, entre autres :

```
>>> from gevent.subprocess import getoutput
>>> getoutput("ls -al")
```

Ce code permet de récupérer le résultat. Le code suivant permet de récupérer également le code de sortie du programme (0 si aucun souci rencontré, un nombre positif autrement) :

```
>>> from gevent.subprocess import getstatusoutput
>>> getstatusoutput("ls -al")
```

On peut aussi utiliser une méthode permettant de lancer automatiquement une exception Python en cas de soucis, au lieu d'avoir à vérifier le code de sortie :

```
>>> from gevent.subprocess import check_output
>>> check_output("ls -al", shell=True)
```

Enfin, il est possible aussi de rediriger le résultat vers un fichier, par exemple :

```
>>> with open("out.txt", "w") as f:
...     call("ls -al", shell=True, stdout=f)
...
0
```

La fonction call permet aussi d'accéder à la sortie d'erreur et à l'entrée standard. Cette fonction renvoie le code de sortie du programme (le zéro à la fin de l'exemple).

© Editions ENI - All rights reserved

1.4 Programmation asynchrone

Il est possible de faire de l'asynchrone en utilisant une fonctionnalité particulière de Gevent :

```
>>> from gevent import sleep
>>> sleep(0)
```

Cette fonctionnalité va permettre de rendre la main à la boucle événementielle pour qu'elle fasse passer un autre processus léger. C'est assez similaire à ce que l'on a vu dans le chapitre précédent, sauf que la syntaxe est différente et que gevent est disponible sur Python 2.x et pour les premières versions de Python 3.x.
On peut créer autant de tâches que l'on souhaite et gérer une synchronisation de la fin des tâches comme vu dans les exemples précédents.
Au-delà de ces fonctionnalités, la bibliothèque Gevent permet de modifier à chaud des bibliothèques usuelles de Python pour les rendre asynchrones. La modification à chaud (ou monkey patch) permet de modifier dynamiquement des fonctions, méthodes ou classes sans avoir à toucher au code source d'origine.
Voici un exemple, toujours adapté de la documentation officielle :

```
>>> from gevent import spawn, wait, monkey
>>> monkey.patch_all()
True

>>> import requests
>>> urls = ["http://eni.fr", "http://inspyration.org", "http://
www.python.org"]
>>> def print_head(url):
...     print("Starting", url)
...     data = requests.get(url).text
...     print(f"{url}: {len(data)}: {data[:50]}")
...
>>> jobs = [spawn(print_head, _url) for _url in urls]
>>> wait(jobs)
```

Dans cet exemple, on commence par appliquer le monkey patch avant de faire quoi que ce soit d'autre. C'est pour cette raison que vous ne pourrez pas utiliser la console interactive bpython sur ces exemples (il se passe trop de choses au lancement de la console pour que le monkey patching puisse se faire. Vous obtiendrez en effet cette erreur :

```
MonkeyPatchWarning: Monkey-patching ssl after ssl has already been imported
may lead to errors, including RecursionError on Python 3.6. It may also
silently lead to incorrect behaviour on Python 3.7. Please monkey-patch
earlier.
```

Une fois que cette étape préparatoire est réalisée, nous pouvons importer le célèbre module **`requests`** et l'utiliser pour afficher le tout début du code HTML de chaque page.
Comme tout à l'heure, on crée nos différentes tâches en avance et on utilise **`wait`** pour attendre que toutes soient terminées. Dans ce cas, il n'y a pas de résultat à récupérer, notre fonction ne va rien renvoyer, mais faire des affichages à l'écran.

Grâce à ceci, on peut écrire du code usuel de Python en asynchrone, la complexité étant cachée. Il n'y a que deux lignes à écrire en tout début de script.

2. Twisted

2.1 Introduction

Twisted est un outil permettant de créer un serveur internet et non pas seulement un serveur web. En d'autres mots, il ne gère pas que le protocole HTTP, mais il peut faire du WebDav, du FTP ou implémenter vos propres protocoles.
Pour l'installer :

```
$ pip install twisted
```

L'objet central est un réacteur et ce dernier va faire tourner des protocoles, lesquels sont des descriptions des données attendues et des réponses à fournir. Ces protocoles sont construits au-dessus de transports, tels que le transport TCP ou UDP.

2.2 Client/serveur TCP

Voici un exemple de serveur TCP Basique : il renvoie tel un écho la donnée qu'il reçoit. On va commencer par importer ce dont nous aurons besoin :

```
from twisted.internet import reactor
from twisted.internet.protocol import Protocol, ServerFactory
```

Voici un exemple simple de protocole permettant de renvoyer la donnée reçue :

```
class Echo(Protocol):
    def dataReceived(self, data):
        "As soon as any data is received, echo it back."
        self.transport.write(data)
```

Il suffit alors de connaître les différentes fonctions à surcharger pour créer son propre protocole et lui faire faire ce que l'on souhaite.
Pour faire tourner ce protocole, il faut monter le réacteur :

```
def main():
    """This runs the protocol on port 8000"""
    factory = ServerFactory()
    factory.protocol = Echo
    reactor.listenTCP(8000,factory)
    reactor.run()
```

Ce code devrait vous sembler assez familier si vous avez déjà lu le chapitre sur la programmation réseau ou encore celui sur la programmation web.
Il faut ensuite l'exécuter :

```
if __name__ == '__main__':
    main()
```

Pour la partie cliente, on écrit également un protocole :

```
class EchoClient(protocol.Protocol):
```

© Editions ENI - All rights reserved

```
    def connectionMade(self):
        self.transport.write(b"hello, world!")

    def dataReceived(self, data):
        print("Server said:", data)
        self.transport.loseConnection()

    def connectionLost(self, reason):
        print("connection lost")
```

On surcharge ici un peu plus de méthode. Comme tout à l'heure, la méthode **dataReveived** est surchargée, elle est appelée lorsque le client reçoit un message de la part du serveur (la réponse du serveur). On surcharge également **connectionMade** pour envoyer un message dès la connexion établie, ce qui est fait ici juste pour avoir un exemple fonctionnel, car cela ne fait pas sens dans le cadre d'une utilisation classique. La dernière méthode **connectionLost** permet de gérer le cas où la connexion se termine.
Toujours pour illustrer les possibilités de Twisted, nous allons surcharger la classe **ClientFactory** pour lui demander d'arrêter le serveur lorsque la connexion est terminée. Pour cela :

```
class EchoFactory(ClientFactory):
    protocol = EchoClient

    def clientConnectionFailed(self, connector, reason):
        print("Connection failed")
        reactor.stop()

    def clientConnectionLost(self, connector, reason):
        print("Connection lost")
        reactor.stop()
```

Il ne reste plus qu'à lancer le serveur ;

```
def main():
    f = EchoFactory()
    reactor.connectTCP("localhost", 8000, f)
    reactor.run()
```

Et l'exécuter :

```
if __name__ == '__main__':
    main()
```

Pour tester ces deux bouts de code, vous pourrez vous rendre dans le répertoire twisted_examples et ouvrir dans deux terminaux d'abord le serveur, puis le client :

```
$ python echo_server.py
$ python echo_client.py
```

L'ordre a ici une importance, puisque le client ne pourra pas être lancé si le serveur ne l'est pas encore.

2.3 Serveur/client UDP

Il existe quelques différences significatives entre la manière de gérer le TCP et l'UDP avec Twisted, elles s'expliquent par les différences entre ces deux transports.
Pour le serveur, il faut importer un protocole spécifique :

```
from twisted.internet import reactor
from twisted.internet.protocol import DatagramProtocol
```

Il faut ensuite créer sa propre classe et surcharger les méthodes utiles. La signature de la méthode est différente, car comme en UDP la connexion n'est pas persistante, il faut passer l'adresse à chaque échange :

```
class Echo(DatagramProtocol):

    def datagramReceived(self, datagram, address):
        self.transport.write(datagram, address)
```

Au niveau du réacteur, il y a une méthode dédiée à l'UDP :

```
def main():
    reactor.listenUDP(8000, Echo())
    reactor.run()
```

Voici maintenant comment lancer le tout (aucune surprise ici) :

```
if __name__ == '__main__':
    main()
```

Voici maintenant pour le client :

```
from twisted.internet import reactor
from twisted.internet.protocol import DatagramProtocol
```

Nous allons ici créer un client qui enverra un message d'accueil lors de la connexion, puis qui, à chaque réponse reçue, renverra un nouveau message. Au bout du dixième message, il s'arrêtera :

```
class EchoClient(DatagramProtocol):
    nb = 0

    def startProtocol(self):
        self.transport.connect('127.0.0.1', 8000)
        self.transport.write(b"Hello!")

    def datagramReceived(self, datagram, host):
        print('Datagram received: ', repr(datagram))
        if self.nb > 9:
            reactor.stop()
            return
        self.nb += 1
        self.transport.write(f"This is message #{self.nb}".encode("utf-8"))
```

On a utilisé pour cela deux méthodes, la première pour gérer ce que l'on souhaite qu'il se passe après la création de la première connexion et la seconde pour gérer ce que l'on souhaite faire à réception d'un message.

© Editions ENI - All rights reserved

Voici la fonction permettant de lancer le serveur :

```
def main():

    t = reactor.listenUDP(0, EchoClient())
    reactor.run()
```

Il ne reste plus qu'à lancer ce client :

```
if __name__ == '__main__':
    main()
```

Là encore, il est recommandé de lancer les deux scripts dans deux terminaux différents pour tester le comportement et modifier ces fichiers pour faire ses propres essais :

```
$ python echo_udp_server.py
$ python echo_udp_client.py
```

2.4 AMP

AMP (*Asynchronous Messaging Protocol*) est un protocole permettant de gérer plusieurs flux requêtes/réponses en même temps et sur la même connexion. Ce protocole est volontairement minimaliste pour vous permettre de l'adapter et ne contient que les fonctionnalités basiques. La documentation officielle est disponible ici : https://twistedmatrix.com/documents/current/api/twisted.protocols.amp.html
Pour la compléter, nous allons introduire un exemple légèrement différent.
Pour utiliser ce protocole, il faut définir le format du message, ce qui se fait via la définition d'une commande qui décrit les entrants et sortants :

```
from twisted.protocols.amp import Command, Integer

class Add(Command):
    arguments = [(b'a', Integer())]
    response = [(b'number', Integer())]
```

On va maintenant faire la même chose, mais en précisant un troisième élément, la gestion des exceptions. Dans notre cas, on refuse que notre valeur puisse être négative et l'on va créer notre propre exception pour gérer cela :

```
class InvalidResult(Exception):
    pass


class Sub(Command):
    arguments = [(b'a', Integer())]
    response = [(b'number', Integer())]
    errors = {InvalidResult: b'INVALID_RESULT'}
```

On peut maintenant passer à la description du protocole. Ce dernier va permettre de modifier une donnée, ici un simple nombre stocké en tant qu'attribut d'une classe. Dans des conditions réelles, nous utiliserions un gestionnaire vers une base de données.

```
class DataHandler:
    number = 3
```

Voici ce protocole :

```
from twisted.protocols.amp import AMP

class Number(AMP):

    def add(self, a):
        print('New number {} + {}'.format(DataHandler.number, a))
        DataHandler.number += a
        return {'number': DataHandler.number}
    Add.responder(add)

    def sub(self, a):
        print('New number {} - {}'.format(DataHandler.number, a))
        DataHandler.number -= a
        if DataHandler.number < 0:
            DataHandler.number = 0
            raise InvalidResult()
        return {'number': DataHandler.number}
    Sub.responder(sub)
```

On crée dans cette classe les méthodes permettant de faire les différentes opérations décrites par les commandes plus haut et on les lie à ces commandes. Dans la seconde, il y a la levée d'une exception, comme prévu par la commande.

Voici maintenant comment créer et lancer le serveur servant le protocole :

```
from twisted.internet import reactor
from twisted.internet.protocol import Factory

def main():
    pf = Factory()
    pf.protocol = Number
    reactor.listenTCP(1234, pf)
    print('reactor started')
    reactor.run()

if __name__ == '__main__':
    main()
```

Nous allons maintenant nous tourner vers le code du client :

```
from twisted.internet import reactor, defer, endpoints
from twisted.internet.endpoints import TCP4ClientEndpoint,
connectProtocol
from twisted.protocols.amp import AMP
from server import Add, Sub, InvalidResult

def doNumber():
    destination = TCP4ClientEndpoint(reactor, '127.0.0.1', 1234)
    addDeferred = connectProtocol(destination, AMP())
    def connected(ampProto):
```

© Editions ENI - All rights reserved

```
            return ampProto.callRemote(Add, a=13)
        addDeferred.addCallback(connected)
        def get_number(result):
            return result['number']
        addDeferred.addCallback(get_number)

        subDeferred = connectProtocol(destination, AMP())
        def connected(ampProto):
            return ampProto.callRemote(Sub, a=15)
        subDeferred.addCallback(connected)
        subDeferred.addCallback(get_number)
        def trapInvalidResult(result):
            result.trap(InvalidResult)
            print("Invalid result: returning 0")
            return 0
        subDeferred.addErrback(trapInvalidResult)

        def done(result):
            print('Done with number:', result)
            reactor.stop()
        defer.DeferredList([addDeferred, subDeferred]).addCallback(done)

    if __name__ == '__main__':
        doNumber()
        reactor.run()
```

La première étape est la création de la variable destination qui va permettre de dire comment joindre le serveur. On crée ensuite des objets de type **deferred**, ce que l'on peut assimiler à des promesses. On va pouvoir rajouter à ces objets des fonctions de rappel pour permettre de traiter le résultat lorsqu'il sera parvenu.
On a des fonctions de rappel pour récupérer le résultat ou les exceptions et même pour afficher le résultat final lorsque toute la chaîne asynchrone sera terminée.
Toute la fonction **doNumber** est le paramétrage de tous ces objets. Tant que le réacteur n'est pas lancé, rien n'est concrètement réalisé.
Voici comment lancer le serveur (et les résultats provenant de deux lancements du client que l'on va voir plus tard) :

```
$ python server.py
reactor started
New number 3 + 13
New number 16 - 15
New number 1 + 13
New number 14 - 15
```

Voici le premier lancement du client :

```
$ python client.py
Done with number: [(True, 16), (True, 1)]
```

Comme on part du nombre 3, que l'on ajoute 13 pour retirer 15, on termine avec le nombre 1. Lors du second appel, on terminerait avec le nombre -1, si ce n'était l'exception lancée, le résultat ressemblera donc à cela :

```
$ python client.py
Invalid result: returning 0
Done with number: [(True, 14), (True, 0)]
```

Nous avons donc testé ici ce qu'il faut faire pour maîtriser le formatage des messages entrants et sortants et la gestion des exceptions.
Le protocole AMP est une alternative très intéressante à d'autres solutions que nous avons déjà présentées ici, telles que Pyro, par exemple (qui est synchrone) ou encore ZMQ.

2.5 Autres protocoles

Sur le site de Twisted, vous trouverez d'autres exemples pour gérer des protocoles différents, en particulier dans la section présentant des exemples. Il existe par exemple une solution pour gérer le protocole FTP. Utiliser ces protocoles signifie les maîtriser et la difficulté réside dans la connaissance et la maîtrise de la théorie plus que sur la difficulté technique inhérente à Twisted.
La maîtrise de Twisted peut être un challenge, mais ce produit est une référence et peu d'alternatives permettent d'en faire autant que lui.

3. Trollius

Comme vous le savez, asyncio est disponible seulement pour Python 3.4 ou supérieur. Une alternative existe cependant pour Python 2.7 dont la branche est toujours active et le restera encore quelques années.
Tout comme asyncio, trollius offre une boucle événementielle ainsi que des transports et protocoles parmi lesquels on peut trouver TCP, UDP ou SSL comme nous vous l'avons présenté dans le chapitre précédent. Cependant, attention, ces protocoles sont basés sur ceux de **`twisted`** plus que sur ceux d'**`asyncio`**. C'est une des raisons pour laquelle nous avons introduit **`twisted`** juste au-dessus.
Ce module vous permettra aussi, plus basiquement, de créer des coroutines et des tâches et de les synchroniser.
Voici un exemple de protocole UDP (très ressemblant à ce que vous avez pu voir précédemment) :

```
class ServerUdpEchoProtocol:

    def connection_made(self, transport):
        print('{transport} started'.format(transport=transport))
        self.transport = transport

    def datagram_received(self, data, addr):
        print('Data {data} received from {addr}'.format(data=data,
addr=addr))
        self.transport.sendto(data, addr)
```

© Editions ENI - All rights reserved

```
    def error_received(self, exc):
        print('Error received: {exc!r}'.format(exc=exc))

    def connection_lost(self, exc):
        print('stop {exc!r}'.format(exc=exc))
```

Voici une fonction permettant de démarrer le serveur :

```
import trollius as asyncio
def start_server(loop, addr):
    t = asyncio.Task(loop.create_datagram_endpoint(
        ServerUdpEchoProtocol, local_addr=addr))
    transport, server = loop.run_until_complete(t)
    return transport
```

Notez la manière d'importer le module **trollilus** (on se croirait en train de lire un code Python 3 utilisant réellement **asyncio**) :
Voici maintenant la gestion de la boucle événementielle :

```
import signal

loop = asyncio.get_event_loop()
loop.add_signal_handler(signal.SIGINT, loop.stop)
server = start_server(loop, (args.host, args.port))
try:
    loop.run_forever()
finally:
    server.close()
    loop.close()
```

On peut également présenter le client. D'abord le protocole :

```
class ClientUdpEchoProtocol:

    message = 'Hello world!'

    def connection_made(self, transport):
        self.transport = transport
        print('sending "{message}" to {transport}'.format(
            message=self.message, transport=transport))
        self.transport.sendto(self.message.encode())
        print('waiting to receive back')

    def datagram_received(self, data, addr):
        print('received "{data}" from {transport}'.format(
            data=data.decode(), addr=addr))
        self.transport.close()

    def error_received(self, exc):
        print('Error received: {exc!r}'.format(exc=exc))
```

```
    def connection_lost(self, exc):
        print('closing transport: {exc!r}'.format(exc=exc))
        loop = asyncio.get_event_loop()
        loop.stop()
```

Voici la fonction pour lancer le client :

```
def start_client(loop, addr):
    t = asyncio.Task(loop.create_datagram_endpoint(
        ClientUdpEchoProtocol, remote_addr=addr))
    loop.run_until_complete(t)
```

Voici maintenant comment gérer la boucle événementielle :

```
loop = asyncio.get_event_loop()
loop.add_signal_handler(signal.SIGINT, loop.stop)
start_client(loop, (args.host, args.port))

try:
    loop.run_forever()
finally:
    server.close()
    loop.close()
```

En conclusion, à l'aide de ce module, vous pourrez faire de la programmation asynchrone quasiment comme avec **asyncio**.

4. HTTP asynchrone avec aiohttp

4.1 Côté serveur

Nous vous avons déjà présenté bottlepy et évoqué l'alternative flask. Il s'agit de microframeworks web permettant de répondre à des besoins simples.
Il existe une alternative qui, syntaxiquement, est extrêmement proche et qui présente l'avantage d'utiliser la programmation asynchrone.
Voici l'exemple d'introduction de aiohttp :

```
from aiohttp.web import  RouteTableDef, Response , Application,  run_app

routes = RouteTableDef()

@routes.get('/')
async def hello(request):
    return Response(text="Hello World!")

app = Application()
app.add_routes(routes)
run_app(app)
```

© Editions ENI - All rights reserved

À comparer à un exemple similaire avec bottlepy :

```
from bottle import route, run, template

@route('/')
def index():
    return "Hello World!"
run(host="localhost", port=8080)
```

ou Flask :

```
from flask import Flask
app = Flask("test App")

@app.route("/")
def hello():
    return "Hello World!"
```

Avec aiohttp, les méthodes peuvent s'écrire en utilisant la programmation asynchrone :

```
from aiohttp.web import json_response

async def test(self, request):
    data = await request.json()
    return json_response(data)
```

Il existe également un module pour utiliser jinja2 de manière asynchrone :

```
from aiohttp_jinja2 import template

@template('test.html')
async def test(self, request):
    data = await request.json()
    return data
```

La vue renvoie un simple dictionnaire, lequel est ensuite interprété de manière asynchrone par jinja2. La syntaxe de ces gabarits HTML est la syntaxe classique Jinja2.
Voici, pour terminer, un exemple d'une fonction asynchrone qui permet de gérer une base de données :

```
@aiohttp_jinja2.template('index.html')
async def index(request):
    async with request.app['db'].acquire() as conn:
        cursor = await conn.execute(db.item.select())
        records = await cursor.fetchall()
        return {'items': [dict(q) for q in records]}
```

On voit ici toutes les spécificités de la programmation asynchrone et ses conséquences syntaxiques. L'intérêt est que l'on peut maintenant gérer en même temps plusieurs requêtes sur un même fil d'exécution. Si la différence n'est pas flagrante avec un serveur et un seul client, lorsque le nombre de clients augmente significativement, l'amélioration des performances devient sensible.

4.2 Côté client

Ce module httpio peut également être utilisé côté client, pour permettre de créer une alternative asynchrone au célèbre module requests. Voici un exemple de code :

```
import asyncio
from aiohttp import ClientSession

async def hello(url):
    async with ClientSession() as session:
        async with session.get(url) as response:
            response = await response.read()
            print(response)

URLS = [
    "http://python.org/",
    "http://inspyration.org/",
    "http://eni.fr/",
]

async def main():
    await asyncio.gather(*(get(url) for url in URLS))
if __name__ == "__main__":
    asyncio.run(main())
```

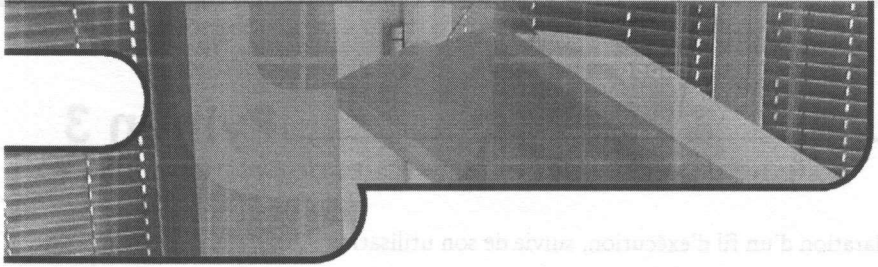
La fonction get est un équivalent (très léger) asynchrone à requests.get. L'intérêt de cette fonction est qu'elle peut être utilisée plusieurs fois en même temps, à l'aide de asyncio.gather. Dans notre exemple, on télécharge trois pages, dont les URL sont définies dans une constante.

On utilise ici une syntaxe Python 3.7.

Il est à noter que pour un équivalent réel de l'excellent module requests, il est possible d'aller chercher le module request-futures (https://github.com/ross/requests-futures) ou encore grequests (https://github.com/kennethreitz/grequests) basé sur Gevent, vu plus haut dans ce chapitre.

Côté client comme côté serveur, au-delà de tout ce que nous avons pu voir jusqu'à présent, il existe également de très nombreuses autres solutions, certaines légères, d'autres très complètes, telles que Tornado, par exemple.

© Editions ENI - All rights reserved



Chapitre 3.5
Programmation parallèle

1. Utilisation d'un fil d'exécution

1.1 Gestion d'un fil d'exécution

1.1.1 Présentation

Python a longtemps proposé un module nommé **thread** qui proposait une API de bas niveau et qui permettait de créer des fils d'exécution selon différentes solutions qui pouvaient correspondre aux capacités de Python en ce domaine.

L'implémentation CPython ayant des pans entiers de son code incompatibles avec une création propre de fils d'exécution, ce module a évolué et a finalement été déprécié avec la finalisation d'une API de haut niveau **threading** qui est à la fois plus simple à utiliser et plus proche des besoins, mais aussi plus fiable.

Pour Python 3.x, l'ancien module de bas niveau déprécié a été simplement retiré (renommé avec un souligné (http://www.python.org/dev/peps/pep-3108/), voir la PEP au chapitre Obsolete) et le nouveau module bénéficie de tous les efforts portés sur l'implémentation de CPython pour permettre d'avoir de vrais fils d'exécution.

Comme on le verra, la problématique n'est pas tellement la création et l'exécution en parallèle de fils d'exécution, mais plutôt la gestion des ressources qu'ils partagent et de leurs communications.

1.1.2 Création

Pour créer un fil d'exécution, il faut utiliser le module de haut niveau :

```
>>> from threading import Thread
```

Pour les besoins de cet exemple, nous chargeons également ceci :

```
>>> from time import time, ctime, sleep
```

Voici la déclaration d'un fil d'exécution, suivie de son utilisation :

```
>>> class Worker(Thread):
...     def __init__(self, name, delay):
...         self.delay = delay
...         Thread.__init__(self, name=name)
...     def run(self):
...         for i in range(5):
...         print("%s: Appel %s, %s" % (self.getName(), i, ctime(time())))
...               sleep(self.delay)
...
>>> try:
...     t1 = Worker("T1", 2)
...     t2 = Worker("T2", 3)
...     t1.start()
...     t2.start()
... except:
...    print("Error: unable to start threads")
...
```

Ce fil d'exécution va effectuer un affichage et se mettre en attente pendant un temps donné. Or, on crée deux fils d'exécution en parallèle, l'un de deux secondes et l'autre de 3 secondes.

Voici le résultat :

```
T1: Appel 0, Fri Oct 21 12:31:17 2011
T2: Appel 0, Fri Oct 21 12:31:17 2011
>>> T1: Appel 1, Fri Oct 21 12:31:19 2011
T2: Appel 1, Fri Oct 21 12:31:20 2011
T1: Appel 2, Fri Oct 21 12:31:21 2011
T1: Appel 3, Fri Oct 21 12:31:23 2011
T2: Appel 2, Fri Oct 21 12:31:23 2011
T1: Appel 4, Fri Oct 21 12:31:25 2011
T2: Appel 3, Fri Oct 21 12:31:26 2011
T2: Appel 4, Fri Oct 21 12:31:29 2011
```

On voit que dès le démarrage des deux premiers fils d'exécution, ils prennent la main pour commencer à écrire. À la troisième ligne, la présence des trois signes en début de ligne montre que le programme principal est terminé et que la main est rendue.

Par contre, les fils d'exécution sont lancés et leur exécution se poursuit. Cette façon d'opérer n'est pas bonne, parce que l'on crée une rupture profonde entre les fils d'exécution et le flux principal d'instructions.

Il faut savoir qu'un fil d'exécution ne peut être interrompu une fois qu'il est lancé. La seule manière pour qu'il se termine est que sa fonction **run** soit terminée.

On peut donc jouer là-dessus pour créer un moyen de les interrompre, plus précisément de leur demander de se terminer. Voici le fil d'exécution précédent avec un mécanisme lui permettant de s'interrompre :

```
>>> class Worker(Thread):
...     def __init__(self, name, delay):
```

© Editions ENI - All rights reserved

```
...             print('Création du worker %s' % name)
...             self.delay = delay
...             self.job_ended = False
...             Thread.__init__(self, name=name)
...         def run(self):
...             for i in range(10):
...                 if self.job_ended:
...                         print('Arrêt forcé de: %s' % self.getName())
...                         return
...                  print("%s: Appel %s, %s" %
(self.getName(), i, ctime(time())))
...                  sleep(self.delay)
...              print('Arrêt naturel de: %s' % self.getName())
...
```

Voici un nouveau fil d'exécution qui a pour but de réaliser une interruption au bout d'un temps donné, il prend en paramètre la liste des autres fils d'exécution sur lesquels il peut agir et un délai :

```
>>> class Stopper(Thread):
...     def __init__(self, threads, delay):
...         print('Création du stoppeur')
...         self.delay = delay
...         self.threads = threads
...         Thread.__init__(self)
...     def run(self):
...         sleep(self.delay)
...         print('Demande d\'arrêt des fils d'exécution')
...         for t in self.threads:
...             t.job_ended = True
...
```

La seule chose qu'elle fait est de modifier un attribut d'un fil d'exécution, celui qui est interrogé, pour savoir s'il doit s'interrompre.
Voici maintenant un code lançant trois fils d'exécution plus un autre pour gérer une interruption au bout de 15 secondes :

```
>>> try:
...     t1 = Worker("T1", 1)
...     t2 = Worker("T2", 2)
...     t3 = Worker("T3", 3)
...     s = Stopper((t1, t2, t3), 15)
...     t1.start()
...     t2.start()
...     t3.start()
...     s.start()
... except:
...    print("Error: unable to start thread")
...
```

Et voici le résultat. On note que le programme rend la main à la huitième ligne, que l'un des fils d'exécution s'interrompt seul :

```
Création du worker T1
Création du worker T2
Création du worker T3
Création du stoppeur
T1: Appel 0, Fri Oct 21 12:54:36 2011
T2: Appel 0, Fri Oct 21 12:54:36 2011
T3: Appel 0, Fri Oct 21 12:54:36 2011
>>> T1: Appel 1, Fri Oct 21 12:54:37 2011
T2: Appel 1, Fri Oct 21 12:54:38 2011
T1: Appel 2, Fri Oct 21 12:54:38 2011
T1: Appel 3, Fri Oct 21 12:54:39 2011
T3: Appel 1, Fri Oct 21 12:54:39 2011
T2: Appel 2, Fri Oct 21 12:54:40 2011
T1: Appel 4, Fri Oct 21 12:54:40 2011
T1: Appel 5, Fri Oct 21 12:54:41 2011
T3: Appel 2, Fri Oct 21 12:54:42 2011
T2: Appel 3, Fri Oct 21 12:54:42 2011
T1: Appel 6, Fri Oct 21 12:54:42 2011
T1: Appel 7, Fri Oct 21 12:54:43 2011
T2: Appel 4, Fri Oct 21 12:54:44 2011
T1: Appel 8, Fri Oct 21 12:54:44 2011
T3: Appel 3, Fri Oct 21 12:54:45 2011
T1: Appel 9, Fri Oct 21 12:54:45 2011
T2: Appel 5, Fri Oct 21 12:54:46 2011
Arrêt naturel de: T1
T3: Appel 4, Fri Oct 21 12:54:48 2011
T2: Appel 6, Fri Oct 21 12:54:48 2011
T2: Appel 7, Fri Oct 21 12:54:50 2011
Demande d'arrêt des fils d'exécution
Arrêt forcé de: T3
Arrêt forcé de: T2
```

Là encore, la solution n'est pas bonne.
On doit créer un fil d'exécution pour s'occuper de trois autres fils d'exécution et le programme principal rend la main sans attendre le résultat.
Il y aurait plusieurs façons de résoudre ces problèmes manuellement, mais Python dispose des outils adéquats pour résoudre cela.

1.2 Gestion de plusieurs fils d'exécution

1.2.1 Lancement et contrôle

Créer plusieurs fils d'exécution permet de paralléliser plusieurs travaux de manière à ce que ce soit plus rapide que de les exécuter séquentiellement.
Par exemple, lorsque l'on traite avec un serveur, on a souvent des temps de latence et l'idée est de profiter de l'attente d'un fil d'exécution pour en exécuter un autre.

© Editions ENI - All rights reserved

Toujours est-il qu'une fois que toutes ces actions ont été menées, il faut être capable de reprendre le fil d'exécution principal. Pour cela, il faut utiliser une file (*queue* en anglais), et y rattacher nos fils d'exécution après les avoir lancés.
Par contre, si la file doit inclure les fils d'exécution, ceux-ci doivent également pointer vers les files pour les informer que leur travail est terminé.
En reprenant l'exemple précédent, voici comment il faut modifier notre travailleur :

```
>>> class Worker(Thread):
...     def __init__(self, queue, name, delay):
...         print('Création du worker %s' % name)
...         self.queue = queue
...         self.delay = delay
...         self.job_ended = False
...         Thread.__init__(self, name=name)
...     def run(self):
...         for i in range(5):
...             if self.job_ended:
...                 print('Arrêt forcé de: %s' % self.getName())
...                 self.queue.task_done()
...                 return
...             print("%s: Appel %s, %s" %
(self.getName(), i, ctime(time())))
...             sleep(self.delay)
...         print('Arrêt naturel de: %s' % self.getName())
...         self.queue.task_done()
...
```

Le stoppeur n'a pas à être modifié, par contre, il faut changer la manière de créer les travailleurs en utilisant leur nouvelle signature, et créer la file auparavant :

```
>>> try:
...     q = Queue(3)
...     t1 = Worker(q, "T1", 1)
...     t2 = Worker(q, "T2", 2)
...     t3 = Worker(q, "T3", 3)
...     t1.start(), t2.start(), t3.start()
```

Ensuite, il faut ajouter chaque fil d'exécution à la file :

```
...     q.put(t1)
...     q.put(t2)
...     q.put(t3)
```

Puis il faut ajouter les autres éléments :

```
...     s = Stopper((t1, t2, t3), 6)
...     s.start()
```

Et il faut utiliser la méthode **join** de la file pour mettre le flux principal en attente :

```
...     print('Attente de la fin des fils d'exécution')
...     q.join()
...     print('fils d'exécution terminés, reprise du flux principal
```

```
d\'instructions')
... except:
...     print("Error: unable to start thread")
...
```

Voici les différences essentielles avec le résultat précédent :

```
T1: Appel 0, Fri Oct 21 14:02:34 2011
T2: Appel 0, Fri Oct 21 14:02:34 2011
Création du stoppeur
T3: Appel 0, Fri Oct 21 14:02:34 2011
Attente de la fin des fils d'exécution
T1: Appel 1, Fri Oct 21 14:02:35 2011
[...]
Arrêt naturel de: T1
[...]
Demande d'arrêt des Tâches
Arrêt forcé de: T2
Arrêt forcé de: T3
fils d'exécution terminés, reprise du flux principal d'instructions
```

En gras, on a les deux instructions qui entourent la méthode d'attente.

1.2.2 Opportunité d'utiliser un fil d'exécution

Si l'on utilise des fils d'exécution pour réaliser des opérations qui ne font rien d'autre que d'utiliser à pleine puissance les ressources de la machine, celles-ci seront dans la pratique exécutées séquentiellement. Cela n'a donc pas de sens, puisque l'on ne gagnera pas de temps, on en perdra même pour la gestion des fils d'exécution eux-mêmes.
Par contre, si le programme passe du temps à attendre après une ressource qui ne dépend pas de la machine, on peut alors mettre ce temps à profit pour faire passer les autres fils d'exécution.
Voici par exemple une liste de documents à télécharger :

```
>>> documents = [
...     'http://docs.python.org/py3k/archives/python-3.2.2-docs-
html.tar.bz2',
...     'http://docs.python.org/archives/python-2.7.2-docs-html.tar.bz2',
...     'http://docs.python.org/dev/archives/python-3.3a0-docs-
html.tar.bz2',
... ]
```

Voici comment les télécharger, les uns après les autres :

```
>>> t = time()
>>> for document in documents:
...     response = os.popen("wget %s" % document, "r")
...     while True:
...         line = response.readline()
...         if not line:
...             break
...
```

© Editions ENI - All rights reserved

```
Téléchargements terminés séquentiellement: 13.89
```

Concrètement, on perd du temps pendant l'instruction **`response.readline()`**.
L'utilisation de fils d'exécution permet alors de compenser la perte de temps due à une ressource extérieure en utilisant les ressources internes de la machine pour les autres fils d'exécution pendant les temps d'attente :

```
>>> class Worker(Thread):
...     def __init__(self, queue, document):
...         self.queue = queue
...         self.document = document
...         Thread.__init__(self)
...     def run(self):
...         response = os.popen("wget %s" % self.document, "r")
...         while True:
...             line = response.readline()
...             if not line:
...                 break
...         self.queue.task_done()
...
>>> try:
...     t = time()
...     q = Queue(3)
...     for document in documents:
...         print(1)
...         task = Worker(q, document)
...         print(2)
...         task.start()
...         print(3)
...         q.put(task)
...         print(4)
...     print('Attente de la fin des téléchargements')
...     q.join()
...     print('Téléchargements terminées: %.2f' % (time() - t))
... except:
...     print("Error: unable to start thread")
...
Attente de la fin des téléchargements
Téléchargements terminés: 7.59
```

Le délai est quasiment diminué par deux, mais cela dépend de beaucoup de paramètres. La même expérience peut être faite en réalisant un ping sur des ordinateurs du réseau ou même sur des noms de domaine.
L'utilisation des fils d'exécution est également indiquée lorsque l'on traite avec un serveur qui est sur une autre machine (sur la même machine, le temps perdu en attendant côté client est utilisé côté serveur, l'intérêt est moindre)...
Encore faut-il trouver une action à réaliser pendant ce temps et la mettre à profit.

1.3 Résolution des problématiques liées

1.3.1 Synchronisation

Dans certains cas, il est possible qu'une ressource ou une série d'instructions soit partagée par plusieurs fils d'exécution, mais qu'il soit nécessaire qu'elle le soit par un et un seul fil d'exécution en même temps.

La résolution de cette problématique passe par la synchronisation des fils d'exécution autour de la section critique à protéger.

Pour mettre en évidence le problème, voici un exemple qui s'exécute deux fois : une fois sans synchronisation et une fois avec.

Pour cela, nous avons besoin de :

```
>>> from threading import Thread, Lock
>>> from queue import Queue
>>> from time import time, sleep
>>> from io import StringIO
```

Voici la fonction critique :

```
>>> def critical_fonction(buffer, letter, lock=None):
...     if lock is not None:
...         lock.acquire()
...     for i in range(10):
...         buffer.write(letter)
...         sleep(0.1)
...     buffer.write('\n')
...     if lock is not None:
...         lock.release()
...
```

Elle est chargée d'écrire dans un buffer qui est partagé par tous les fils d'exécution, sauf qu'elle écrit petit à petit (en simulant du temps nécessaire pour d'éventuels calculs) et son travail se termine par le rajout d'un passage à la ligne.

Le code est prévu pour être synchronisé ou non, suivant le cas d'utilisation. Cette opération de synchronisation s'effectue par l'utilisation d'un verrou qui est sollicité, puis relâché. Tout le code contenu entre ces deux événements ne peut s'exécuter qu'une seule fois en même temps.

Voici le code correspondant pour le fil d'exécution utilisateur de la fonction critique :

```
>>> class Worker(Thread):
...     def __init__(self, queue, buffer, letter, lock=None):
...         self.queue = queue
...         self.buffer = buffer
...         self.letter = letter
...         self.lock = lock
...         Thread.__init__(self)
...     def run(self):
...         for i in range(5):
...             critical_fonction(self.buffer, self.letter, self.lock)
```

© Editions ENI - All rights reserved

```
...                 sleep(0.1)
...             self.queue.task_done()
...
```

Voici le code qui crée les fils d'exécution, la file d'attente et qui gère le verrou :

```
>>> try:
...     for lock in (None, Lock()):
...         buffer = StringIO()
...         q = Queue(3)
...         t1 = Worker(q, buffer, "A", lock)
...         t2 = Worker(q, buffer, "B", lock)
...         t3 = Worker(q, buffer, "C", lock)
...         t1.start()
...         t2.start()
...         t3.start()
...         q.put(t1)
...         q.put(t2)
...         q.put(t3)
...         q.join()
...         print(buffer.getvalue())
... except:
...     print("Error: unable to start thread")
...
```

Ce verrou est alors un élément connexe aux fils d'exécution qui peut être passé en paramètre ou utilisé depuis un espace de nommage extérieur.
Voici le résultat dans le premier cas :

```
ABCABCBACBACABCBACABCBACABCBAC


BACABCABCBACABCBACABCBACBACABC
```

Comme on a un délai d'attente pour chaque fil d'exécution d'un dixième de seconde à chaque opération, c'est-à-dire écriture d'une lettre (ce qui est énorme par rapport à la durée d'exécution d'une instruction), on passe au fil d'exécution suivant.
Si l'on utilise un verrou, voici ce que l'on a :

```
AAAAAAAAAA
BBBBBBBBBB
CCCCCCCCCC
AAAAAAAAAA
BBBBBBBBBB
CCCCCCCCCC
```

Cela correspond exactement à ce que l'on souhaite avoir.
Il est important d'utiliser ce genre de verrou dans un contexte parallèle lorsqu'on accède à une ressource partagée comme un buffer. Même si on est persuadé que notre code est suffisamment rapide pour que la main ne passe pas, on n'est pas maître de cette décision.

Le développeur doit considérer que s'il utilise du développement parallèle, il ne maîtrise pas le passage d'un fil d'exécution à un autre ; les règles étant particulièrement complexes et pas forcément déterministes.
Le verrou est donc une bonne solution, simple, à employer dans ce cas précis.
La synchronisation telle qu'on l'a vue est réalisée assez simplement.
Lorsque l'on utilise la méthode **acquire**, il est vérifié que le verrou n'est pas déjà positionné. Tant que c'est le cas, la méthode boucle en attente. Dès que le verrou n'est plus positionné, alors la méthode le positionne à son tour et rend la main. Les instructions suivantes peuvent alors s'exécuter.
Une fois celles-ci terminées, on relâche le verrou, et on continue le flux d'instructions. Ce n'est que lors du changement de fil d'exécution qui suivra que, dans le cas où un autre fil d'exécution a déjà appelé la méthode, **acquire** verra sa boucle d'attente se terminer. Il y a également une syntaxe plus simple permettant de gérer le verrou :

```
>>> def critical_fonction(buffer, letter, lock):
...     with lock:
...         for i in range(10):
...             buffer.write(letter)
...             sleep(0.1)
...         buffer.write('\n')
...
```

Ceci permet d'alléger la syntaxe et est rigoureusement identique à la méthode utilisant **acquire** et **release**.
Pour terminer, il faut noter que lorsqu'un verrou est créé, de quelque manière que ce soit, il est nécessaire de le porter sur le moins d'instructions possible et de le relâcher avant d'en poser un autre quelle que soit la situation.

1.3.2 Synchronisation conditionnelle

Il est parfois nécessaire de gérer la manière dont les fils d'exécution vont interagir entre eux, en particulier essayer de maîtriser un minimum l'ordre dans lequel ils vont accéder à certaines ressources.
On peut prendre l'exemple d'un fil d'exécution producteur de données qui pourraient être consommées par plusieurs autres fils d'exécution.
Pour cela, on a un objet particulier :

```
>>> from threading import Thread, Condition
```

Et notre exemple a également besoin de :

```
>>> from time import sleep
>>> from io import StringIO
```

Il existe un objet particulier permettant de mettre des fils d'exécution en attente, et de les libérer via une notification pour qu'ils se retrouvent de nouveau exécutables.
L'idée est donc de créer un client et de le mettre tout de suite en état d'attente.
Lorsqu'il est libéré, il récupère la donnée qui lui est destinée – sans la modifier, puisqu'il la partage avec d'autres clients potentiels – mais de manière exclusive. Dès que le travail est terminé, le client se remet en attente.

© Editions ENI - All rights reserved

Le producteur utilise également la donnée partagée de manière exclusive et la modifie, et il notifie les fils d'exécution en attente. Cependant, il doit passer la main.
Voici le client :

```
>>> class Consumer(Thread):
...     def __init__(self, name, buffer, condition):
...         self.buffer = buffer
...         self.condition = condition
...         self.current = ''
...         Thread.__init__(self, name=name)
...     def run(self):
...         while True:
...             self.condition.acquire()
...             if self.buffer.getvalue() in ['', self.current]:
...                 print('Attente du consommateur %s'% self.getName())
...                 self.condition.wait()
...             print('Reprise du consommateur %s' % self.getName())
...             self.current = self.buffer.getvalue()
...             if self.buffer.getvalue() == '#':
...                 print('Fin du consommateur %s' %self.getName())
...                 return
...             print('Reçu %s < "%s"' % (self.getName(), self.current))
...             self.condition.release()
...
```

Voici le producteur :

```
>>> class Producer(Thread):
...     def __init__(self, buffer, condition, phrases):
...         self.buffer = buffer
...         self.condition = condition
...         self.phrases = phrases
...         Thread.__init__(self)
...     def run(self):
...         self.phrases.append('#')
...         for phrase in phrases:
...             self.condition.acquire()
...             self.buffer.seek(0)
...             self.buffer.truncate(0)
...             self.buffer.write(phrase)
...             print('Envoyé > "%s"' % phrase)
...             self.condition.notifyAll()
...             sleep(0.000001)
...             self.condition.release()
...         print('Fin du producteur')
...
```

Si on a bloqué tous les consommateurs avant de lancer le producteur, un simple sleep permet de passer la main et celle-ci va être reprise tout de suite par chaque consommateur.

Voici les données que l'on veut faire passer :

```
>>> phrases = [
...     "Phrase 1",
...     "Phrase 2",
...     "Phrase 3",
... ]
```

Voici le flux principal :

```
>>> buffer = StringIO()
>>> condition = Condition()
>>> c1 = Consumer('1', buffer, condition)
>>> c2 = Consumer('2', buffer, condition)
>>> p = Producer(buffer, condition, phrases)
>>> c1.start()
>>> sleep(0.000001)
Attente du consommateur 1
>>> c2.start()
>>> sleep(0.000001)
Attente du consommateur 2
>>> p.start()
```

On a donc bien les deux consommateurs en attente (on s'en assure en réalisant un **sleep**). On peut alors démarrer le producteur :

```
>>> Envoyé > "Phrase 1"
```

Ainsi, le **sleep** du producteur rend la main à un des consommateurs :

```
Reprise du consommateur 1
Reçu 1 < "Phrase 1"
Attente du consommateur 1
```

Ce dernier se replace immédiatement en attente. Les autres consommateurs réalisent alors la même opération :

```
Reprise du consommateur 2
Reçu 2 < "Phrase 1"
Attente du consommateur 2
```

Et le processus se répète pour chaque phrase, jusqu'à l'envoi du mot-clé qui arrête le processus, ferme le producteur et les consommateurs :

```
Envoyé > "#"
Fin du producteur
Reprise du consommateur 1
Fin du consommateur 1
Reprise du consommateur 2
Fin du consommateur 2
```

1.3.3 Sémaphore

La programmation parallèle introduit des concepts très complexes à résoudre. Lorsqu'il s'agit d'une ressource dont on doit restreindre l'accès, la solution est le sémaphore. En plus des outils déjà vus, le sémaphore est utilisable directement en tant que tel.

© Editions ENI - All rights reserved

Une des problématiques qui peut se poser est que l'on accepte de laisser plusieurs fils d'exécution accéder à une même ressource, de manière parallèle, mais que l'on souhaite dans le même temps limiter le nombre d'accès concurrents.
C'est un des cas pour lequel l'utilisation d'un sémaphore est indiquée et il s'utilise exactement de la même manière qu'un verrou. La syntaxe avec le mot-clé **with** est également valable. Semaphore et BoundedSemaphore sont deux fabriques permettant de récupérer ces objets.
Voici un fil d'exécution utilisant un sémaphore :

```
>>> class Worker(Thread):
...     def __init__(self, semaphore, name, delay):
...         print('Création du worker %s' % name)
...         self.semaphore, self.delay = semaphore, delay
...         Thread.__init__(self, name=name)
...     def run(self):
...         with self.semaphore:
...             print("Thread > %s" % self.getName())
...             sleep(self.delay *0.001)
...             print("Thread < %s" % self.getName())
...
```

Voici comment créer le sémaphore et les fils d'exécution :

```
>>> try:
...     s = BoundedSemaphore(value=3)
...     for i in range(10):
...         t = Worker(s, 'T%s' % i, i)
...         t.start()
... except:
...     print("Error: unable to start thread")
...
```

On a donc dix fils d'exécution, dont trois simultanés :

```
Création du worker T0
Thread > T0
Création du worker T1
Thread < T0
Thread > T1
Création du worker T2
Thread > T2
Création du worker T3
Thread > T3
Création du worker T4
Création du worker T5
Thread < T1
Thread > T4
Création du worker T6
Création du worker T7
Création du worker T8
```

```
Thread < T2
Création du worker T9
Thread > T5
>>> Thread < T3
Thread > T6
Thread < T4
Thread > T7
Thread < T5
Thread > T8
Thread < T6
Thread > T9
Thread < T7
Thread < T8
Thread < T9
```

Un autre enseignement de cet exemple est de voir que le très court temps attribué à sleep implique que la création du dernier fil d'exécution a lieu alors que le premier d'entre eux est déjà terminé.

2. Utilisation de processus

2.1 Gestion d'un processus

2.1.1 Présentation

Sur un certain nombre de plans, vu de l'interface de haut niveau fournie par Python, le multitâche (ou multi-fils) est semblable au multiprocessus. Mais la ressemblance s'arrête à celle des listes de méthodes proposées par les objets.

Les problématiques à résoudre sont plus complexes. Leur avantage est de permettre d'exécuter deux actions au même moment sur deux processeurs différents et donc diminuer leur temps d'exécution.

Un processus est indépendant, embarque son propre environnement d'exécution. Il est également lié au processus qui le crée, qui peut être la console Python, la machine virtuelle Python ou un autre processus Python.

2.1.2 Création

Créer un processus est relativement simple. Voici les modules nécessaires :

```
>>> from multiprocessing import Process
>>> from time import sleep
```

Voici un travail à adjoindre au processus :

```
>>> def work(name):
...     print('Debut du travail: %s' % name)
...     for j in range(10):
...         for i in range(10):
...             sleep(0.01)
...             print('.', sep='', end='')
...         print('.')
```

© Editions ENI - All rights reserved

```
...     print('Fin du travail: %s' % name)
...
```

Il suffit maintenant de créer le processus, et de le démarrer :

```
>>> p = Process(target=work, args=('Test',))
>>> p.start()
>>> p.join()
```

Voici le résultat :

```
Debut du travail: Test
..........
..........
..........
..........
..........
..........
..........
..........
..........
..........
Fin du travail: Test
```

On va mettre en évidence les caractéristiques du processus :

```
>>> print ('Flux principal: %s' % os.getpid())
Flux principal: 8879
```

Le processus a pour PID parent ce numéro :

```
>>> def work(name):
...     print('Debut du travail: %s' % name)
...     print('Pid: %s, parent: %s' % (os.getpid(), os.getppid()))
...     print('Fin du travail: %s' % name)
...
```

On va lancer deux processus successivement :

```
>>> for i in range(2):
...     p = Process(target=work, args=('Infos',))
...     p.start()
...     p.join()
...
```

Voici le résultat :

```
Debut du travail: Infos
Pid: 10165, parent: 8879
Fin du travail: Infos
Debut du travail: Infos
Pid: 10166, parent: 8879
```

Si on relance le processus plus tard, les numéros seront différents, parce qu'ils sont attribués par le système d'exploitation et partagés par tous les processus de la machine, Python ou pas :

```
Fin du travail: Infos
>>> for i in range(2):
...     p = Process(target=work, args=('Infos',))
...     p.start()
...     p.join()
...
Debut du travail: Infos
Pid: 10683, parent: 8879
Fin du travail: Infos
Debut du travail: Infos
Pid: 10684, parent: 8879
Fin du travail: Infos
```

On est donc là dans un contexte beaucoup plus général et le fait de créer un processus Python permet de communiquer avec d'autres processus qui ne sont pas forcément Python.
Voici le premier script testé qui a été remanié pour mettre en évidence le fait que le programme principal peut continuer en parallèle du processus fils :

```
>>> def work(name):
...     print('Debut du travail: %s' % name)
...     for j in range(2):
...         for i in range(10):
...             sleep(0.01)
...             print('.', sep='', end='')
...         print('.')
...     print('Fin du travail: %s' % name)
...
```

Voici maintenant comment créer un processus et le faire démarrer en parallèle au flux principal :

```
>>> p = Process(target=work, args=('Test',))
>>> p.start()
>>> Debut du travail: Test
>>> for j in range(4):
...     for i in range(10):
...         sleep(0.01)
...         print('o', sep='', end='')
...     print('o')
...
```

Voici le résultat :

```
..........
oooooooooo
..........
Fin du travail: Test
```

© Editions ENI - All rights reserved

```
oooooooooo
oooooooooo
oooooooooo
```

Au moment où l'on veut se synchroniser avec le processus, il faut utiliser la fonction **`p.join()`**. À ce moment-là, si le processus fils est déjà fini, la méthode rend la main immédiatement, sinon, elle se met en attente comme précédemment.
Lorsque le flux principal crée des processus, il les démarre et se met en attente d'eux. Il ne fait jamais autre chose en parallèle lui-même. Ce qui est présenté juste au-dessus est uniquement présent pour l'exemple, mais ne doit pas être réalisé.

2.2 Gestion de plusieurs processus

2.2.1 Synchronisation

Comme nous l'avons vu précédemment, le processus dispose lui-même de la méthode **`join`** qui permet la synchronisation.
Pour créer plusieurs processus accédant à une ressource protégée, il faut également créer un verrou, mais celui-ci provient du module **`multiprocessing`**.
Voici un travail minimaliste :

```
>>> def work(name, lock):
...     with lock:
...         print('Work with %s' % name)
...         sys.stdout.flush()
...
```

Et la méthode pour lancer tous les processus :

```
>>> lock = Lock()
>>> for i in range(5):
...     Process(target=work, args=(i, lock)).start()
...
Work with 0
Work with 3
Work with 1
Work with 2
Work with 4
```

2.2.2 Paralléliser un travail

Lorsque l'on doit appliquer un calcul sur plusieurs valeurs, il est possible de paralléliser ce calcul. Voici une fonction d'exemple qui ne calcule rien, mais qui renvoie la valeur qu'on lui passe en paramètre ainsi que le numéro du processus qui sert à gérer ce calcul :

```
>>> def f(x):
...     return x, os.getpid()
...
```

Pour appliquer le calcul à une seule valeur, voici comment procéder :

```
>>> pool.apply(f, [16])
(16, 11098)
```

Pour appliquer un calcul à un ensemble de valeurs, il faut créer un réservoir de processus (en l'occurrence, on en autorise quatre simultanés), et d'y mapper la fonction :

```
>>> pool = Pool(processes=4)
>>> print(pool.map(f, range(10)))
[(0, 11097), (1, 11097), (2, 11097), (3, 11097), (4, 11097), (5, 11097),
(6, 11097), (7, 11097), (8, 11097), (9, 11097)]
```

Il existe un mode asynchrone permettant de continuer le flux principal et d'aller rechercher le résultat plus tard ou de le demander immédiatement avec un éventuel timeout :

```
>>> result_id = pool.map_async(f, range(10))
>>> result_id.get(timeout=1)
[(0, 11100), (1, 11100), (2, 11100), (3, 11100), (4, 11100), (5, 11100),
(6, 11100), (7, 11099), (8, 11100), (9, 11098)]
```

Si le timeout est trop court, une exception est logiquement levée :

```
>>> result_id = pool.map_async(f, range(10))
>>> result_id.get(timeout=0.0000001)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.2/multiprocessing/pool.py", line 541, in get
    raise TimeoutError
multiprocessing.TimeoutError
```

Ce mode asynchrone est également valable pour la méthode **apply** avec la méthode **apply_async**. Le fonctionnement ressemble à celui d'une connexion asynchrone LDAP, avec un identifiant à fournir pour récupérer la valeur au final.
Il est également permis de récupérer les valeurs à la manière d'un itérateur :

```
>>> it = pool.imap(f, range(3))
>>> print(next(it))
(0, 11100)
>>> print(next(it))
(1, 11100)
>>> print(next(it))
(2, 11100)
>>> print(next(it))
Traceback (most recent call last): [...]
StopIteration
```

Voici une mise en illustration :

```
>>> for x, pid in pool.imap(g, range(10)):
...     print('recv %s (%s)' % (x, pid))
...
```

Dans un contexte classique, un itérateur calcule une valeur, la renvoie et seulement par la suite, calcule la valeur suivante. Dans un contexte avec plusieurs processus, chaque processus réalise un calcul et le renvoie, mais les envois et les réceptions sont bien intercalés.

© Editions ENI - All rights reserved

Voici le résultat :

```
Send 0
send 1
recv 0 (11758)
recv 1 (11758)
send 2
send 3
send 4
send 5
send 6
recv 2 (11760)
recv 3 (11759)
send 7
recv 4 (11759)
recv 5 (11759)
send 8
send 9
recv 6 (11760)
recv 7 (11760)
recv 8 (11760)
recv 9 (11758)
```

À comparer avec :

```
Send 0
send 1
send 2
send 3
send 4
send 5
send 6
send 7
send 8
send 9
recv 0 (11761)
recv 1 (11761)
recv 2 (11761)
recv 3 (11760)
recv 4 (11758)
recv 5 (11760)
recv 6 (11758)
recv 7 (11760)
recv 8 (11758)
recv 9 (11760)
```

Dans le dernier cas, le code est strictement le même, mais en utilisant map au lieu de imap. Comme on le voit, tant que l'on donne du travail au réservoir de processus, ce sont toujours les mêmes processus (quatre dans notre exemple) qui travaillent et se mettent en attente dès qu'ils ont terminé.

Enfin, cet objet pool dispose de la méthode **`join`** permettant de se mettre en attente de la fin de l'exécution de tous les processus, utile pour savoir quand reprendre le flux principal. Dernière méthode extrêmement importante, la méthode **`terminate`** qui permet de stopper tous les processus en attente.

```
>>> pool.terminate()
```

Avoir un terminal ouvert en parallèle et exécuter la commande **`ps -ax | grep python`** permet de voir (entre autres) la console Python et les processus créés ou détruits. De plus, une fois les processus créés, il embarque son environnement d'exécution. Toute variable (fonction ou classe) déclarée postérieurement n'existe pas chez les processus fils.

2.3 Résolution des problématiques liées

2.3.1 Communication interprocessus

Le module multiprocessing possède un objet dédié à l'échange de données entre processus. Cet objet est créé avant les processus. Il renvoie deux objets multiprocessing.Connection :

```
>>> i, o = Pipe()
```

Lorsque l'un envoie un message, l'autre le reçoit :

```
>>> i.send(42)
>>> o.recv()
42
>>> i.send('Hello')
>>> o.recv()
'Hello'
```

Si deux messages sont envoyés d'affilée, ils sont reçus séquentiellement :

```
>>> i.send(42)
>>> i.send('Hello')
>>> o.recv()
42
>>> o.recv()
'Hello'
```

Lorsque l'on lance la réception sans avoir de données, on se met en attente :

```
>>> o.recv()
^CTraceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyboardInterrupt
```

C'est exactement ce que nous allons faire en lançant un processus qui écrit et un autre qui lit. Ce dernier est démarré en premier et se met en attente du second :

```
>>> def reader(o):
...     while True:
...         data = o.recv()
...         print(data)
...         if data == '#':
...             return
...
>>> def writer(i):
...     for data in ['Message', 42, '#']:
...         i.send(data)
...
>>> i, o = Pipe()
>>> pr = Process(target=reader, args=(o,))
>>> pr.start()
>>> pw = Process(target=writer, args=(i,))
>>> pw.start()
>>> Message
42
#
>>> pw.join()
>>> pr.join()
```

© Editions ENI - All rights reserved

2.3.2 Partage de données entre processus

Au-delà de la communication entre processus, il est parfois nécessaire de partager des données d'un processus à un autre. Cette opération est très particulière et pour des raisons techniques, il faut utiliser des types C.
On utilise une lettre pour spécifier le type et on donne la valeur :

```
>>> value = Value('i', 0)
```

Voici deux fonctions qui interviennent sur la valeur de façon successive et durent un temps volontairement différent :

```
>>> def add(value):
...     for i in range(3):
...         value.acquire()
...         value.value += 2
...         value.release()
...         print(value)
...         sleep(1)
...
>>> def sub(value):
...     while value.value > 0:
...         value.acquire()
...         value.value -= 1
...         value.release()
...         print(value)
...         sleep(2)
...
```

On crée le premier processus et on attend un peu avant de créer le second :

```
>>> pa = Process(target=add, args=(value,))
>>> pa.start()
>>> sleep(0.5)
>>> ps = Process(target=sub, arg<Synchronized wrapper for c_int(2)>
s=(value,))
>>> ps.start()
```

Voici comment évoluent les valeurs :

```
<Synchronized wrapper for c_int(2)>
<Synchronized wrapper for c_int(1)>
<Synchronized wrapper for c_int(3)>
<Synchronized wrapper for c_int(5)>
<Synchronized wrapper for c_int(4)>
<Synchronized wrapper for c_int(3)>
<Synchronized wrapper for c_int(2)>
<Synchronized wrapper for c_int(1)>
<Synchronized wrapper for c_int(0)>
```

Il est aussi possible de créer des tableaux, mais ils ne doivent contenir que des éléments homogènes entre eux :

```
>>> from multiprocessing import Array
>>> arr = Array('d', [1.0, 1.4, 1.7, 1.9, 2.0])
```

2.4 Opportunité d'utiliser les processus

La création d'un processus a un coût en termes de performances. En créer un pour qu'il exécute un petit nombre d'instructions simples n'est en conséquence pas judicieux.
Dès lors que l'on a un travail lourd à effectuer (création de beaucoup de classes, beaucoup de code sans délai de latence), la création d'un processus peut être judicieuse.
Cela permet la délégation de travail et des performances améliorées.
Cela peut également permettre la séparation d'une application sur un plan technique afin de ne pas la figer. Par exemple, une application avec interface graphique doit pouvoir afficher une animation sur une interface graphique en même temps qu'un calcul lourd est effectué.

2.5 Démon

Un démon est la traduction littérale du terme « daemon » et est un rétro-accronyme qui a été défini de manière à justifier son nom.
Les démons sont particulièrement connus dans les systèmes d'exploitation pour être des processus qui démarrent avec le système, se terminent avec lui et qui assurent des actions en arrière-plan, principalement des tâches techniques comme répondre à des requêtes réseau, surveiller le matériel ou autres.
Les plus connus sont sendmail, postfix et toute la panoplie de ceux se terminant par la lettre « d » comme httpd, lighthttpd, slapd, mysqld, crond, lpd...
En réalité, sur un plan purement technique, un démon est simplement un processus orphelin et tout processus orphelin se voit rattaché au processus numéro 1 qui est « Init ». Ceci est vrai pour Linux/Unix et pour les systèmes modernes.
La technique pour créer un processus orphelin est donc de créer un fils et de se terminer ou si l'on souhaite continuer à exister de créer un fils qui crée un fils avant de se terminer selon la technique du « fork off and die ».
Sous Windows, le terme policé de « service » est utilisé pour désigner de tels processus et leur fonctionnement est particulier. Windows étant une boîte noire, la documentation technique sur le fonctionnement de bas niveau est infime et complexe. La création de processus semble se faire par des appels systèmes particuliers, mais leur détail n'est pas aussi diffusé que ce que l'on connaît sur GNU/Linux.
Pour Python, la documentation officielle montre qu'un processus (module **multiprocessing**) possède un attribut **daemon**. Celui-ci ne porte pas la sémantique d'UNIX, mais signifie que Python essaiera de terminer ce processus si son père est terminé au lieu de se contenter d'attendre une jointure.
Ce terme porte également la même sémantique pour le module **threading** mais appliquée à un fil d'exécution au lieu d'un processus. Or, un fil d'exécution peut toujours être terminé. Rendre le thread « démonique » fait que celui-ci peut être terminé à tout moment et qu'il n'est pas nécessaire de l'attendre pour quitter.

© Editions ENI - All rights reserved

Un moyen de créer des processus démon au sens UNIX du terme est d'utiliser les fonctionnalités de bas niveau de Python.
Voici un tel script :

```
#!/usr/bin/python3

import os
import time
import sys

pid=os.fork()
if pid==0:
    for i in range(60):
        print('Je suis le fils PID: %s et je tourne' % os.getpid())
        time.sleep(1)
else:
    print('Je suis le père PID: %s, mon fils est le PID %s' %
(os.getpid(), pid))
    sys.exit(0)
```

On crée un processus par un **fork** et on termine, soit « fork off and die ». Attention, ceci n'est valable que pour UNIX.
Voici la commande à lancer :

```
$ ./daemon.py
```

Et voici les deux phrases que l'on peut lire :

```
Je suis le père PID: 12770, mon fils est le PID 12771
Je suis le fils PID: 12771 et je tourne
```

On illustre donc le fonctionnement du **fork** en même temps que la création du démon. Voici ce que l'on peut observer pendant les soixante secondes ou le programme principal est terminé et le processus est encore en vie.

```
$ ps ax | grep python3
12771 pts/0    S      0:00 /usr/bin/python3 ./daemon.py
```

On peut également voir le détail du rattachement du processus fils au processus 1 :

```
pid=os.fork()
if pid==0:
    for i in range(10):
        print('Je suis le fils PID: %s et je tourne. Mon père est
PID %s' % (os.getpid(), os.getppid()))
        time.sleep(1)
else:
    print('Je suis le père PID: %s, mon fils est le PID %s' %
(os.getpid(), pid))
    time.sleep(5)
    sys.exit(0)
```

Voici ce qu'il se passe avant que la main soit rendue au terminal et après :

```
Je suis le fils PID: 12927 et je tourne. Mon père est PID 12926
$ Je suis le fils PID: 12927 et je tourne. Mon père est PID 1
```

Le **$** qui est l'invite de commande du terminal est le signe que la main est rendue et ce script est disponible pour que vous puissiez tester.

3. Exécution asynchrone

3.1 Introduction

Jusqu'à présent, on a vu comment créer des fils d'exécution et les gérer avec des outils de haut niveau permettant de donner du travail à chaque fil d'exécution, comment les utiliser au mieux et gérer leur exécution finement.

On a également vu comment créer des processus et les gérer toujours avec des outils de haut niveau.

Ces opérations, par rapport à ce qu'elles impliquent comme technicité pour les langages de bas niveau, sont relativement simples à utiliser et on a vu différents cas d'application et différentes mises en œuvre possibles.

Il reste également possible de réaliser des opérations de plus bas niveau encore, comme l'utilisation du **fork** qui reste disponible, même si ce présent chapitre n'insiste pas spécialement sur cette fonctionnalité, préférant mettre le focus sur le haut niveau.

Ainsi, ceux qui connaissent le **fork** en C seront capables d'effectuer la même opération en Python, car il s'agit de la même chose. De plus, les règles de programmation sont communes à tous les langages.

Cependant, l'utilisation du bas niveau, voire des deux modules de haut niveau que sont **threading** et **multiprocessing** induit des risques et nécessite une certaine maîtrise, même si cela est beaucoup plus simple qu'avec des langages de plus bas niveau, comme le C.

Le grand risque est de créer un processus ou un fil d'exécution dont on perd la maîtrise. Si une telle chose arrive, le fil d'exécution ou le processus peut très bien occuper à 100 % le processeur et ne pas rendre la main pendant un temps long, potentiellement problématique pour les autres processus tournant sur la machine (au-delà des autres fils d'exécution ou processus Python) et cela peut entraîner le plantage du système d'exploitation.

Heureusement, les outils de haut niveau que l'on a vus jusqu'à présent, s'ils sont bien utilisés, permettent d'éviter cela. Le recours à **sleep** au sein des fils d'exécution et processus est certes coûteux, mais il est nécessaire de manière à passer la main aux autres s'ils en ont besoin. Il vaut mieux perdre du temps à passer en revue tous les autres processus, quitte à ce qu'ils prennent la main pour la rendre de suite parce qu'ils n'en ont pas besoin, que de perdre le contrôle de sa machine.

Or, ce genre de problématique, même simplifiée, doit être maîtrisée et Python est un langage qui vise à rendre accessible ce genre de fonctionnalités qui est par ailleurs particulièrement utilisé dans le calcul scientifique par exemple ou ailleurs et par des personnes dont le cœur de métier n'est pas l'informatique.

© Editions ENI - All rights reserved

Leur besoin est de lancer des traitements en utilisant les meilleures technologies, en bénéficiant de la plus grande efficacité possible. Ces traitements ne sont pas dépendants les uns des autres et peuvent donc être exécutés de manière asynchrone sans suivre un ordre déterminé.

3.2 Présentation

Pour ces problématiques orientées sur la fonctionnalité plutôt que sur la technique et l'utilisation manuelle des fils d'exécution ou processus, Python offre une interface de plus haut niveau. Celle-ci propose une classe qui définit comment exécuter des actions et qui se décline par l'utilisation des fils d'exécution ou des processus en gardant dans les deux cas exactement la même interface.
Voici un exemple d'action que l'on veut effectuer :

```
>>> from time import sleep
>>> def action(numero, delai):
...     print('debut de %s (attente de %s)' % (numero, delai))
...     sleep(delai)
...     print('> fin de %s (attente de %s)' % (numero, delai))
...
```

Voici les valeurs que l'on souhaite traiter :

```
>>> delais = [2, 5, 3, 6, 1]
```

Voici comment s'y prendre. D'abord les imports nécessaires :

```
>>> from concurrent.futures import ThreadPoolExecutor
>>> from concurrent.futures import as_completed
```

Puis le code :

```
>>> with ThreadPoolExecutor(max_workers=3) as executor:
...     futures={executor.submit(action, n, d): d for n, d in
enumerate(delais)}
...     for future in as_completed(futures):
...         print('#Complété: %s' % future)
...
```

Le code s'exécute en deux temps. D'abord la création des fils d'exécution, puis l'itération sur chaque résultat qui arrive :

```
debut de 0 (attente de 2)
debut de 1 (attente de 5)
debut de 2 (attente de 3)
> fin de 0 (attente de 2)
debut de 3 (attente de 6)
#Complété: <Future at 0x2ad53d0 state=finished returned NoneType>
> fin de 2 (attente de 3)
debut de 4 (attente de 1)
#Complété: <Future at 0x2ad5790 state=finished returned NoneType>
> fin de 4 (attente de 1)
debut de 5 (attente de 3)
```

```
#Complété: <Future at 0x2ad59d0 state=finished returned NoneType>
> fin de 1 (attente de 5)
#Complété: <Future at 0x2ad55d0 state=finished returned NoneType>
> fin de 5 (attente de 3)
#Complété: <Future at 0x2ad6050 state=finished returned NoneType>
> fin de 3 (attente de 6)
#Complété: <Future at 0x2ad5950 state=finished returned NoneType>
```

On voit que dès qu'un résultat est arrivé, on lance immédiatement un autre fil d'exécution puis on renvoie le résultat. Le code est efficace, rapide et l'exécution est bien gérée.
On voit aussi que l'on récupère le retour ou une exception en cas d'erreur :

```
#Complété: <Future at 0x2acf890 state=finished raised NameError>
```

Pour afficher une exception, s'il y en a une, ou pour afficher le résultat, il faut procéder ainsi :

```
if future.exception() is not None:
    print('%s lève une exception: %s' % (n, future.exception()))
else:
    print('%s a pour résultat %s' % (n, future.result()))
```

En procédant ainsi, l'effort supplémentaire à fournir pour utiliser la programmation parallèle à la place de la programmation linéaire est minimaliste, ce qui met cette technique à la portée de tous.
De plus, les risques sont assumés par Python et non plus laissés à l'appréciation du développeur puisqu'aucune fonctionnalité de bas niveau n'est nécessaire.
En ce qui concerne l'utilisation des processus, le mode d'opération est exactement le même. Par contre, un import différent est utilisé :

```
>>> from concurrent.futures import ThreadPoolExecutor
```

De manière à produire un exemple parlant, original et fonctionnellement intéressant, voici comment résoudre le problème des n-reines appliqué à plusieurs valeurs de n.
La méthode de résolution est exactement la même que celle présentée précédemment :

```
>>> def nqueens(n):
...     columns=range(n)
...     for board in permutations(columns):
...         if n == len(set(board[i]+i for i in columns)) \
...            == len(set(board[i]-i for i in columns)):
...                 yield board
...
```

Voici comment afficher le nombre de résultats trouvés et le temps mis :

```
>>> ph = 'Résolution du problème des %2d-Dames : %5d solutions
( %9.3f secondes)'
>>> from time import time
>>> def action(n):
...     t0=time()
...     lr=len(list(nqueens(n)))
...     t1=time()
```

© Editions ENI - All rights reserved

```
...     return ph % (n, lr, t1-t0)
...
```

Et voici comment résoudre les problèmes de 0-reines à 9-reines :

```
>>> with ProcessPoolExecutor(max_workers=4) as executor:
...     futures={executor.submit(action, n): n for n in range(10)}
...     for future in as_completed(futures):
...         if future.exception() is not None: pass
...         else:
...             print(future.result())
...
```

Voici le résultat :

```
Résolution du problème des  1-Dames :      1 solutions (     0.000 secondes)
Résolution du problème des  0-Dames :      1 solutions (     0.000 secondes)
Résolution du problème des  3-Dames :      0 solutions (     0.000 secondes)
Résolution du problème des  4-Dames :      2 solutions (     0.000 secondes)
Résolution du problème des  2-Dames :      0 solutions (     0.000 secondes)
Résolution du problème des  5-Dames :     10 solutions (     0.001 secondes)
Résolution du problème des  6-Dames :      4 solutions (     0.005 secondes)
Résolution du problème des  7-Dames :     40 solutions (     0.029 secondes)
Résolution du problème des  8-Dames :     92 solutions (     0.177 secondes)
Résolution du problème des  9-Dames :    352 solutions (     1.387 secondes)
```

Avec une machine disposant de plusieurs cœurs et/ou processeurs, les résultats sont meilleurs, mais l'intérêt de cet algorithme est très limité.

Ce module permet également d'annuler un appel, de vérifier qu'il a bien été annulé et de rajouter une fonction de rappel (callback) à la fin du traitement.

La documentation officielle présente d'autres exemples (http://docs.python.org/dev/library/concurrent.futures.html) et la PEP qui est consacrée à ce sujet fournit des informations complémentaires (http://www.python.org/dev/peps/pep-3148/).

Voici pour terminer un autre exemple illustrant comment utiliser ce module. En effet, il serait utile, pour gagner du temps, que les traitements nécessaires à la résolution d'un seul problème puissent être séparés :

```
>>> def action(board, n):
...     if n == len(set(board[i]+i for i in columns)) \
...          == len(set(board[i]-i for i in columns)):
...             return True
...     return False
...
>>> def get_nqueens_solutions(n):
...     columns, result = range(n), []
...     with ProcessPoolExecutor(max_workers=4) as executor:
...         futures={executor.submit(action, b, n): b for b in
permutations(columns)}
...         for future in as_completed(futures):
...             if future.exception() is not None:
...                 print(future.exception())
...             else:
```

```
...                     if future.result():
...                         result.append(futures[future])
...     return result
...
```

Ce qu'il se passe ici est que l'on formalise dans une action l'analyse permettant de déterminer la validité d'une seule solution et que l'on connecte chaque worker à cette action.
En clair, comme la vérification elle-même est extrêmement courte, on passe son temps à construire les workers, à les lancer et à les arrêter. La parallélisation n'apporte rien, mais en plus elle coûte beaucoup.
Pour donner un ordre d'idée, elle multiplie les temps de traitement par 100. Mais ce n'est pas la parallélisation elle-même qui est en cause, c'est la manière de l'utiliser. En effet, il est préférable de prendre une autre option :

```
>>> def partial_nqueens(n, max, current):
...     columns=range(n)
...     for board in (p for i, p in
enumerate(permutations(columns)) if i % max == current):
...         if n == len(set(board[i]+i for i in columns)) \
...             == len(set(board[i]-i for i in columns)):
...                 yield board
...
```

Ici, on a une méthode de résolution complète, mais qui porte sur une partie des possibilités (elles sont divisées équitablement en **max** parties).
Voici ce que cela donne pour 4 :

```
>>> list(partial_nqueens(4, 4, 0))
[]
>>> list(partial_nqueens(4, 4, 1))
[(2, 0, 3, 1)]
>>> list(partial_nqueens(4, 4, 2))
[(1, 3, 0, 2)]
>>> list(partial_nqueens(4, 4, 3))
[]
```

Voici maintenant comment calculer toutes les solutions et créer les workers :

```
>>> def get_nqueens_solutions(n, workers=4):
...     result = []
...     with ProcessPoolExecutor(max_workers=workers) as executor:
...         futures={executor.submit(partial_nqueens, n, workers, i):
n for i in range(workers)}
...         for future in as_completed(futures):
...             if future.exception() is not None:
...                 print(future.exception())
...             else:
...                 if future.result():
...                     result.extend(
                            future.result())
...     return result
```

© Editions ENI - All rights reserved

```
...
```

Voici maintenant de quoi visualiser la solution :

```
>>> def print_nqueens_solutions(n):
...     board_sep, board_top = '\n---', '+'+'-+'*n
...     for board in get_nqueens_solutions(n):
...         print(board_top)
...         for l in board:
...             print('|'+ |'*l+'Q|'+ |'*(n-l-1))
...         print(board_top+board_sep)
...
```

Testons pour 4 solutions :

```
>>> print_nqueens_solutions(4)
+-+-+-+-+
| | |Q| |
|Q| | | |
| | | |Q|
| |Q| | |
+-+-+-+-+
---
+-+-+-+-+
| |Q| | |
| | | |Q|
|Q| | | |
| | |Q| |
+-+-+-+-+
---
```

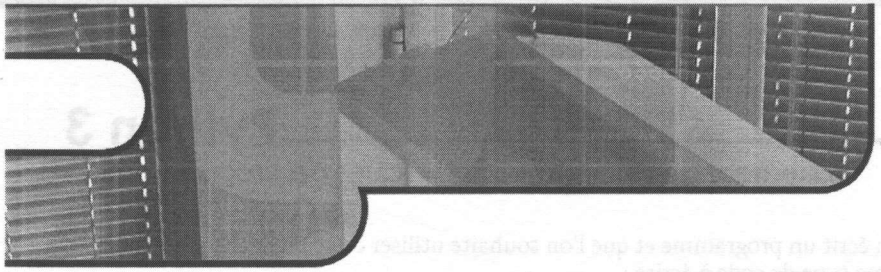
Voici maintenant comment mesurer le temps passé pour la résolution :

```
>>> ph = '%2d-Dames avec %d workers: %5d solutions ( %9.3f
secondes)'
>>> from time import time
>>> def mesure(n, w=4):
...     t0=time()
...     lr=len(list(get_nqueens_solutions(n, workers=w)))
...     t1=time()
...     return ph % (n, w, lr, t1-t0)
...
>>> for n in range(6, 12):
...     for w in range(3, 7):
...         mesure(n, w)
...
' 6-Dames avec 3 workers:     4 solutions (     0.217 secondes)'
' 6-Dames avec 4 workers:     4 solutions (     0.222 secondes)'
' 6-Dames avec 5 workers:     4 solutions (     0.222 secondes)'
' 6-Dames avec 6 workers:     4 solutions (     0.224 secondes)'
' 7-Dames avec 3 workers:    40 solutions (     0.240 secondes)'
' 7-Dames avec 4 workers:    40 solutions (     0.229 secondes)'
```

```
' 7-Dames avec 5 workers:    40 solutions (     0.233 secondes)'
' 7-Dames avec 6 workers:    40 solutions (     0.237 secondes)'
' 8-Dames avec 3 workers:    92 solutions (     0.338 secondes)'
' 8-Dames avec 4 workers:    92 solutions (     0.335 secondes)'
' 8-Dames avec 5 workers:    92 solutions (     0.324 secondes)'
' 8-Dames avec 6 workers:    92 solutions (     0.334 secondes)'
' 9-Dames avec 3 workers:   352 solutions (     1.198 secondes)'
' 9-Dames avec 4 workers:   352 solutions (     1.136 secondes)'
' 9-Dames avec 5 workers:   352 solutions (     1.266 secondes)'
' 9-Dames avec 6 workers:   352 solutions (     1.287 secondes)'
'10-Dames avec 3 workers:   724 solutions (     9.342 secondes)'
'10-Dames avec 4 workers:   724 solutions (    10.152 secondes)'
'10-Dames avec 5 workers:   724 solutions (    10.520 secondes)'
'10-Dames avec 6 workers:   724 solutions (    11.020 secondes)'
'11-Dames avec 3 workers:  2680 solutions (   101.378 secondes)'
'11-Dames avec 4 workers:  2680 solutions (   109.338 secondes)'
'11-Dames avec 5 workers:  2680 solutions (   115.579 secondes)'
'11-Dames avec 6 workers:  2680 solutions (   121.815 secondes)'
```

Le résultat dépend de la machine et de ses capacités, mais est meilleur que le même algorithme linéaire. Il faut bien choisir son nombre de workers. Ici, il s'agit d'un petit portable double-cœur.

© Editions ENI - All rights reserved



Chapitre 3.6
Programmation distribuée

1. Définitions

La programmation distribuée est une solution idéale pour répondre à des problématiques de concurrence où l'on a besoin de faire exécuter des tâches en arrière-plan. La solution consiste à avoir, à côté de l'application principale, une application secondaire capable d'exécuter ces tâches. Il faut ensuite transmettre des messages d'une application à l'autre pour d'une part expliquer la tâche à exécuter et d'autre part renvoyer le résultat (cette dernière partie étant optionnelle).

Pour gérer cette problématique, on se base sur le protocole AMQP (*Advanced Message Queuing Protocol*) qui est un standard pour l'organisation de ces messages. Nous présenterons les principaux patrons de conception qui permettent de résoudre la plupart des besoins en les utilisant à bas niveau. Nous commencerons par présenter ØMQ dont l'interface de programmation est relativement similaire à l'utilisation de simples sockets réseau, puis nous utiliserons une solution basée sur un message broker RabbitMQ.

Dans un second temps, nous présenterons Celery qui est une solution de haut niveau pour gérer la problématique de la programmation distribuée très rapidement et efficacement. Nous terminerons en présentant crossbar et en présentant des solutions mixant Python et JavaScript.

2. ØMQ

2.1 Présentation générale

ØMQ est une solution élégante et performante qui va nous permettre de présenter les patrons de conception essentiels liés à cette problématique. Pour l'installer :

```
$ pip install zmq
```

Lorsque l'on écrit un programme et que l'on souhaite utiliser cette bibliothèque, il y a toujours le même type de code à écrire :

```
>>> from zmq import Context
>>> context = Context()
>>> socket = context.socket(METHODE)
```

Ce code se trouve dans les deux applications (ou plus) qui communiquent entre elles. Il permet de créer le socket réseau dans le cadre duquel les échanges de messages auront lieu. La méthode à utiliser dépend du patron de conception que l'on souhaite utiliser.
Une fois le socket créé, on peut visualiser ses méthodes :

```
>>> dir(socket)
```

Comme on travaille avec un socket, on utilisera la méthode **bind** pour écouter sur une interface réseau et un port particulier. Les protocoles supportés sont TCP, UDP, PGM, EPGM, INPROC et IPC. Il existe également **bind_to_random_port** qui permet de laisser au programme le choix du port. Pour fermer une connexion, il faudra utiliser la méthode **unbind**. L'attribut **closed** permet de connaître l'état du socket.
Le programme qui réalisera l'opération de **bind** sera le serveur, celui qui écoute le port (et il ne peut y en avoir qu'un). L'autre programme devra utiliser **connect** et sera le client. Suivant le paradigme utilisé, la communication est monodirectionnelle ou bidirectionnelle. Autrement dit, dans certains protocoles, le client et le serveur peuvent à la fois envoyer et recevoir de la donnée, alors que dans d'autres, l'un ne fait que recevoir et l'autre qu'envoyer. Pour fermer une telle connexion, il faut utiliser **disconnect**.
Les sockets peuvent ensuite envoyer ou recevoir des données et ils supportent plusieurs formats :

Type d'objets	Méthode réception	Méthode envoi
Octets	**recv**	**send**
Objet convertible en JSON (usuellement un dictionnaire)	**recv_json**	**send_json**
Liste d'octets (essentiellement)	**recv_multipart**	**send_multipart**
Objet Python sérialisable via pickle	**recv_pyobj**	**send_pyobj**
Objet Python (+ méthode de sérialisation passée en paramètre)	**recv_serialized**	**send_serialized**
Chaîne de caractères	**recv_string**	**send_string**
Chaîne de caractères	**recv_unicode**	**send_unicode**

Nous pouvons aussi citer, pour compléter, les méthodes **subscribe** et **unsubscribe** qui sont utiles pour un paradigme particulier, que nous présenterons plus loin.
La méthode **close** permet de fermer le socket. Si cette opération n'est pas faite à la main, la fermeture du socket se fait lorsque la variable **socket** est recyclée (garbage collector).

© Editions ENI - All rights reserved

Remarque

Comme nous allons devoir utiliser beaucoup de terminaux pour tester les différents codes, je vous encourage à installer terminator si vous êtes sous Linux.

2.2 Pair

Ce paradigme permet de mettre en place une connexion exclusive entre un serveur et un seul client. Cette connexion est bidirectionnelle.
Ce paradigme est idéal pour permettre l'échange d'informations entre deux entités, en s'assurant qu'il soit impossible à une troisième entité de venir interroger le serveur.
Voici un exemple de serveur :

```
from zmq import Context, PAIR

context = Context()
socket = context.socket(PAIR)
socket.bind("tcp://*:5556")

while True:
    msg = socket.recv()
    print(f"{msg!r}")
    socket.send(msg)
```

La technique consiste à créer un objet contexte, d'en tirer un socket en précisant le type PAIR puis de recevoir et envoyer des données. Le serveur commence toujours par attendre de recevoir des données, les traite, renvoie une réponse et se replace en attente. Il passera essentiellement son temps en position d'attente. Pour ces raisons, il est possible de coupler ce qui est montré dans ce chapitre avec les techniques de programmation asynchrone, mais on ne traitera pas cette partie ici.
On notera la présence d'un astérisque dans l'URL, lequel permet d'accepter différentes interfaces réseau.
Voici maintenant le client :

```
from zmq import Context, PAIR

context = Context()
socket = context.socket(PAIR)
socket.connect("tcp://localhost:5556")

while True:
    msg = input("> ")
    if not msg:
        break
    socket.send_string(msg)
    msg = socket.recv()
    print("echo:", msg)

socket.disconnect("tcp://localhost:5556")
```

La technique est exactement la même, sauf que l'on commence par envoyer de la donnée avant d'écouter la réponse du serveur. On notera que l'on utilise **connect** (pour le client) au lieu de **bind** (pour le serveur).
Vous pouvez trouver ce code dans le répertoire distribué/zmq et l'exécuter dans deux terminaux, chacun utilisant une de ces lignes :

```
$ python pair_echo_client.py
$ python pair_echo_server.py
```

Le fait de démarrer l'un ou l'autre en premier n'a pas d'effet.
Vous pourrez tester, par exemple, le fait de lancer plusieurs clients pour vérifier que la seconde connexion ne se fait pas.
On pourra partir également de ce code pour tester d'autres propriétés du paradigme PAIR, par exemple le fait que l'on n'est pas obligé d'attendre une réponse du serveur pour lui envoyer une autre donnée.

2.3 Client/serveur

Le paradigme client/serveur ressemble beaucoup au paradigme réseau classique : un serveur écoute un port et répond à un client. La différence essentielle est, ici, que le client peut toucher plusieurs serveurs à la fois.
Voici un exemple de serveur :

```
from zmq import Context, REP
import time
import sys

try:
    port =  int(sys.argv[1])
except IndexError:
    port = 5556
except ValueError:
    port = 5556

context = Context()
socket = context.socket(REP)
socket.bind(f"tcp://*:{port}")

while True:
    msg = socket.recv()
    print(f"{msg!r} from port {port}")
    socket.send(msg)
```

Toute la partie centrale du code permet de lire le port depuis les arguments du programme ou en donner un par défaut.
Pour le reste, c'est essentiellement le même code que précédemment, sauf que le type de socket est REP (pour réponse).

© Editions ENI - All rights reserved

Voici maintenant le client :

```
from zmq import Context, REQ
import sys

context = Context()
socket = context.socket(REQ)
for port in sys.argv[1:]:
    socket.connect(f"tcp://localhost:{port}")

while True:
    msg = input("> ")
    socket.send_string(msg)
    msg = socket.recv()
    print("echo:", msg)
```

Là encore, le client est très similaire au précédent, sauf que l'on utilise REQ (pour requête). On note également le fait que l'on se connecte à autant de serveurs que ce qui est passé en argument.
Voici comment lancer trois serveurs différents :

```
$ python reqrep_echo_server.py
$ python reqrep_echo_server.py 5557
$ python reqrep_echo_server.py 5558
```

Voici comment lancer un client qui écoute sur le premier serveur :

```
$ python reqrep_echo_client.py 5556
```

Seul le premier serveur répondra à ce client. Pour que le client s'adresse à plusieurs serveurs, il faudra préciser les ports d'écoute.

```
$ python reqrep_echo_client.py 5556 5557 5558
$ python reqrep_echo_client.py 5556 5557
$ python reqrep_echo_client.py 5556 5558
$ python reqrep_echo_client.py 5557 5558
```

On peut maintenant tester ce qu'il se passe en ouvrant tout ce petit monde dans des terminaux différents et en envoyant de la donnée.
Je vous recommande aussi de vous créer un client qui envoie des données régulièrement, en espaçant ses envois à l'aide de **`time.sleep(1)`**, par exemple.

2.4 Publish/subscribe

Ce paradigme est l'un des plus importants. L'idée essentielle est que l'on a d'un côté des programmes qui publient des informations, en précisant un sujet (topic) et d'un autre côté d'autres programmes qui écoutent certains topics en particulier. Les programmes clients ne recevront ainsi que ce à quoi ils ont souscrit.
Ce paradigme peut se décliner en deux scénarios : un éditeur (publisher) qui a plusieurs souscripteurs (subscriber) ou un souscripteur qui écoute plusieurs éditeurs.
Cela détermine qui est le serveur et qui est le client.

Dans notre premier scénario, nous n'avons qu'un seul éditeur, ce sera donc le serveur :

```
from zmq import Context, PUB
import time
import sys
from random import choice

TOPICS = (b"topic1", b"topic2", b"topic3")

try:
    port =  int(sys.argv[1])
except IndexError:
    port = 5556
except ValueError:
    port = 5556

context = Context()
socket = context.socket(PUB)
socket.bind(f"tcp://*:{port}")

while True:
    msg, topic = input("> ").encode("utf-8"), choice(TOPICS)
    print(f"{topic!r} {msg!r} from port {port}")
    socket.send(topic + b" " + msg)
```

Par rapport aux serveurs précédents, on utilise un socket PUB au lieu de REP et on préfixe notre message du sujet (topic) avec l'espace comme séparateur.
La grande différence est ici que l'on crée un serveur, mais que ce dernier ne fait qu'envoyer des informations, il n'en reçoit pas (pas de **recv** dans ce code).
Ceci peut paraître étrange, d'autant que l'on a l'habitude d'avoir un serveur dont l'essentiel du temps d'exécution est consacré à l'attente (blocage du code sur un **recv**). Mais ce qui fait qu'un serveur est un serveur est qu'il utilise **bind** et non **connect** et pas forcément la manière dont il gère ses écoutes et envois de données. D'autre part, au niveau réseau pur, il se passe plus de choses que ce que l'on voit là dans ce code de haut niveau.
Notons que nous nous donnons aussi la possibilité d'avoir plusieurs serveurs, chacun écoutant sur un port différent. Ce qui nous permet d'avoir, comme précédemment, des clients à l'écoute de plusieurs serveurs. Voici un client :

```
from zmq import Context, SUB, SUBSCRIBE
import sys

context = Context()
socket = context.socket(SUB)
for port in sys.argv[1:]:
    socket.connect(f"tcp://localhost:{port}")

while True:
    topic = input("topic: ")
    if not topic:
```

© Editions ENI - All rights reserved

```
            break
        socket.subscribe(topic)

    while True:
        msg_parts = socket.recv().split()
        topic, msg = msg_parts[0], " ".join(
            m.decode("utf-8") for m in msg_parts[1:])
        print(f"{topic}: {msg}")
```

Les ports auxquels il faut se connecter sont passés en paramètres. Les sujets sont demandés dès le début du programme.
Nous voyons l'utilisation de la méthode **subscribe** pour se mettre à l'écoute des sujets souhaités. Le filtre qui fait que le client recevra ou non un message se fait côté souscripteur.
Avec ces deux codes, on peut réaliser plusieurs tests en démarrant plusieurs serveurs et plusieurs clients ainsi, dans des terminaux différents :

```
$ python pubsub_1_server_publisher.py
$ python pubsub_1_server_publisher.py 5557
$ python pubsub_1_server_publisher.py 5558
$ python pubsub_1_client_subscriber.py 5556 5557 5558
$ python pubsub_1_client_subscriber.py 5556
$ python pubsub_1_client_subscriber.py 5557 5558
```

On choisira pour chaque client d'entrer une sélection de sujets différents pour vérifier le bon fonctionnement du paradigme.
Pour un fonctionnement qui correspond au paradigme originel, il ne faudrait faire tourner qu'un seul serveur. Cependant, le fait de cumuler le pub/sub avec le paradigme client/serveur vu plus haut est également intéressant.
Le second scénario est moins connu, mais plus intéressant. Il s'agit cette fois-ci d'avoir un serveur qui souscrit à plusieurs clients éditeurs. Là encore, on peut rajouter plusieurs souscripteurs en créant plusieurs serveurs.
Voici le serveur :

```
from zmq import Context, SUB, SUBSCRIBE
from random import sample, randint
import sys

TOPICS = (b"topic1", b"topic2", b"topic3")

try:
    port =  int(sys.argv[1])
except IndexError:
    port = 5556
except ValueError:
    port = 5556

context = Context()
```

```
socket = context.socket(SUB)
socket.bind(f"tcp://*:{port}")

for topic in sample(TOPICS, randint(2, len(TOPICS))):
    print(f"Subscribe to {topic}")

    socket.setsockopt(SUBSCRIBE, topic)

while True:
    msg_parts = socket.recv().split()
    topic, msg = msg_parts[0], " ".join(
        m.decode("utf-8") for m in msg_parts[1:])
    print(f"{topic}: {msg}")
```

Ici, le serveur va recevoir de la donnée sans jamais en renvoyer. Par défaut, nous avons décidé dans cet exemple de faire en sorte qu'il s'abonne automatiquement à au moins deux sujets (et vu qu'il n'y en a que trois, la probabilité qu'il soit abonné au trois est assez élevée, ce qui est utile pour un serveur de contrôle).
On donne aussi la possibilité de passer un port en paramètre pour avoir plusieurs serveurs.
Voici maintenant le client :

```
from zmq import Context, PUB
import time
import sys
from random import choice

TOPICS = (b"topic1", b"topic2", b"topic3")

context = Context()
socket = context.socket(PUB)
for port in sys.argv[1:]:
    socket.connect(f"tcp://localhost:{port}")

while True:
    msg, topic = input("> ").encode("utf-8"), choice(TOPICS)
    print(f"{topic!r} {msg!r} from port {port}")
    socket.send(topic + b" " + msg)
```

Ce client est là encore assez similaire aux précédents : il vous demande de la donnée et la renvoie. Par commodité, le sujet de chaque message est choisi au hasard parmi l'ensemble des possibilités.
Vous pouvez tester le serveur et le client en démarrant plusieurs instances dans des terminaux différents :

```
$ python pubsub_2_server_subscriber.py
$ python pubsub_2_server_subscriber.py 5557
$ python pubsub_2_server_subscriber.py 5558
$ python pubsub_2_client_publisher.py 5556 5557 5558
$ python pubsub_2_client_publisher.py 5556
$ python pubsub_2_client_publisher.py 5557 5558
```

© Editions ENI - All rights reserved

Comme pour le premier scénario, le cas classique d'utilisation de ce paradigme n'implique qu'un seul serveur.
Ces techniques PUB/SUB permettent de gérer facilement la communication inter-applicative sélective. Chaque service ne s'abonne qu'aux sujets qui l'intéresse et réagit à chaque message qu'il reçoit, en sachant qu'il ne reçoit que les messages qu'il a choisis.
Vous aurez aussi noté que la communication est monodirectionnelle : avec PUB, on ne peut invoquer que les méthodes **send*** et avec SUB que les méthodes **recv***.
Pour terminer, précisons un détail important : le souscripteur ne reçoit les messages qu'à partir du moment où il a souscrit. Si juste avant sa souscription des messages avaient été émis, ils sont perdus pour lui.
Une conséquence est que si une instance souscriptrice tombe, elle aura perdu tous les messages jusqu'à ce qu'elle soit remontée.

2.5 PUSH/PULL

Ce paradigme permet de répartir le traitement de données. Dans l'exemple précédent, un message envoyé en PUB était reçu par tous les SUB et trié par rapport au sujet (topic). Dans ce paradigme, tout message envoyé en PUSH ne sera reçu que par un seul des PULL. Ce ne sera cependant pas toujours le même, l'idée étant de faire travailler tout le monde équitablement, de partager le travail.
Sachant cela, voici un serveur PUSH :

```
from zmq import Context, PUSH
import time
import sys
from random import choice

context = Context()
socket = context.socket(PUSH)
socket.bind(f"tcp://*:5556")

while True:
    msg = input("> ").encode("utf-8")
    if not msg:
        break
    print(f"{msg!r} from producer")
    socket.send(msg)
```

Là encore, au niveau du code, il n'y a pas grand-chose de différent. C'est le choix du mode PUSH qui fait que ce code se comportera de façon spécifique.
Voici maintenant le client :

```
from zmq import Context, PULL

context = Context()
socket = context.socket(PULL)
socket.connect(f"tcp://localhost:5556")

while True:
```

```
    msg = socket.recv()
    print(msg)
```

Là encore, dans ce code, il n'y a que le choix de PULL qui est différent.
Cependant, le comportement de l'application est particulier; pour vous en rendre compte, lancez un serveur et plusieurs clients :

```
$ python pushpull_producer.py
$ python pushpull_consumer.py
$ python pushpull_consumer.py
```

Lorsqu'un message part du producteur, seul un consommateur le reçoit. Et comme pour le paradigme PUB/SUB, la communication est monodirectionnelle.
On peut noter que, toujours comme avec PUB/SUB, il existe deux scénarios : il est possible de changer les rôles serveurs et clients, avec un serveur PULL et des clients PUSH. Ceci permettrait de collecter de la donnée provenant de différents processus en un point centralisé.
On peut aussi imaginer utiliser les deux scénarios en même temps, c'est ce que permet le patron pipeline.

2.6 Patron pipeline

Le patron pipeline permet de gérer un flux de données : une application envoie des données se faire consommer par des workers avec du PUSH/PULL et ces derniers envoient un résultat à un collecteur avec du PULL/PUSH.
Voici pour commencer le producteur :

```
from zmq import Context, PUSH
import time
import sys
from random import choice

context = Context()
socket = context.socket(PUSH)
socket.bind("tcp://*:5560")

while True:
    msg = input("> ").encode("utf-8")
    if not msg:
        break
    print(f"{msg!r} from producer")
    socket.send(msg)
```

Ce dernier est similaire au code PUSH précédent. On peut reprendre ce code et l'ajuster pour le collecteur de résultat :

```
from zmq import Context, PULL
import sys
from random import randint
from collections import defaultdict

context = Context()
```

© Editions ENI - All rights reserved

```
socket = context.socket(PULL)
socket.bind("tcp://*:5561")

data = defaultdict(str)

while True:
    msg_parts = socket.recv().decode("utf-8").split()
    consumer_id, msg = msg_parts[0], " ".join(msg_parts[1:])
    # print(f"{consumer_id!r}: {msg!r}")
    if msg == "STOP":
        print(f"Worker {consumer_id!r} is DONE:", data.pop(consumer_id))
    else:
        data[consumer_id] += msg
```

Cette fois-ci, on crée un serveur PULL et ce dernier récupère les différents éléments pour les collecter et les afficher.
L'identifiant du consommateur est en fait celui du worker qui travaille la donnée et le collecteur reçoit la réponse morceau par morceau. On utilise donc un **`defaultdict`** pour gérer la donnée de chaque worker (donc l'identifiant consommateur est la clé).
Dans notre scénario, le programme principal renvoie une chaîne de caractères et le worker retourne au collecteur un caractère par seconde. Le collecteur a pour charge de reconstruire le mot et de l'afficher.
Voici les workers :

```
from time import sleep
from zmq import Context, PULL, PUSH
import sys
from random import randint

try:
    consumer_id = sys.argv[1].encode("utf-8")
except IndexError:
    consumer_id = str(randint(1, 10000)).encode("utf-8")

context = Context()
socket_in = context.socket(PULL)
socket_in.connect("tcp://localhost:5560")
socket_out = context.socket(PUSH)
socket_out.connect("tcp://localhost:5561")

while True:
    msg = socket_in.recv()
    print(f"{consumer_id!r}: {msg!r}")
    for char in msg:
        socket_out.send(consumer_id + b" " + chr(char).encode("utf-8"))
        sleep(1)
    socket_out.send(consumer_id + b" " + b"STOP")
```

Dans cet exemple, on utilise deux sockets. Et comme la communication est unidirectionnelle, on les nomme **socket_in** et **socket_out** pour que leurs rôles soient clairs. Une fois que ces sockets sont montés, il n'y a plus qu'à les utiliser.
On doit aussi gérer un identifiant de consommateur unique.
Voici comment lancer tous ces programmes dans des terminaux différents :

```
$ python pipeline_main.py
$ python pipeline_result_collector.py
$ python pipeline_worker.py worker1
$ python pipeline_worker.py worker2
$ python pipeline_worker.py worker3
$ python pipeline_worker.py worker4
```

Vous pouvez lancer autant de workers que souhaité (au moins un), puis taper des mots dans le programme principal pour visualiser ce qu'il se passe.

3. AMQP avec RabbitMQ

Dans le chapitre précédent, nous avons montré les différents patrons que l'on peut utiliser pour écrire des applications distribuées. Ce chapitre va montrer comment utiliser RabbitMQ à bas niveau et le chapitre suivant montrera une solution de haut niveau.
Nous allons utiliser Pika pour manipuler une file d'attente de RabbitMQ et présenter les différentes manières de les utiliser.

3.1 Installation

Pour installer RabbitMQ, il faut utiliser un dépôt spécifique, car les paquets issus de la distribution, s'ils existent dans le dépôt officiel, sont souvent datés et souffrent de vulnérabilités :

```
# wget -O - "https://packagecloud.io/rabbitmq/rabbitmq-server/gpgkey" |
apt-key add -
# apt update
# apt install rabbitmq-server
```

Pour les autres distributions, il faudra se référer à la documentation officielle (https://www.rabbitmq.com/download.html).

3.2 Configuration

Lorsque l'on utilise PostgreSQL, par exemple, il faut créer un utilisateur avec son mot de passe ainsi qu'une base de données. On peut alors l'utiliser dans notre programme Python à partir d'une URL telle que :

```
database_url = 'postgres://user:secret@localhost:5432/db_name'
```

Avec RabbitMQ, il s'agit de la même chose :

```
rabbit_broker_url = 'amqp://user:secret@localhost:5672/vhost'
```

© Editions ENI - All rights reserved

Il existe ensuite quelques commandes à connaître pour créer un utilisateur, un vhost et pour paramétrer les permissions :

```
# rabbitmqctl add_user test test
# sudo rabbitmqctl add_vhost vhost
# sudo rabbitmqctl set_permissions -p vhost test ".*" ".*" ".*"
```

Nous pouvons maintenant utiliser RabbitMQ à partir de l'URL définie plus haut.
Pour terminer, il faut également installer pika :

```
pip install pika
```

3.3 Introduction

Le premier exemple de code est très basique : un producteur et un consommateur utilisant une file d'attente (queue) pour échanger des messages. Il s'agit essentiellement de voir comment créer une connexion, envoyer ou recevoir des messages.
Voici le producteur :

```
from pika import BlockingConnection
from pika.connection import URLParameters

connection = BlockingConnection(
    URLParameters('amqp://test:test@localhost:5672/vhost'))
channel = connection.channel()

channel.queue_declare(queue='test_1')

while True:
    data = input("> ")
    if not data:
        data = "STOP"
    channel.basic_publish(exchange='', routing_key='test_1', body=data)
    print(f"Sent '{data}'")
    if data == "STOP":
        break

connection.close()
```

Si on décompose ce code, on voit les étapes suivantes pour mettre en place la communication :
- import des deux classes nécessaires pour créer la connexion
- création de la connexion
- création du canal de communication (**channel**)
- déclaration d'une file d'attente

Il est impératif de s'assurer que la file d'attente existe bien avant de l'utiliser. Si elle existe déjà, la commande **channel.queue_declare** ne fait rien de particulier.

Dans le reste du code, comme pour tester ZMQ, on demande de la donnée à l'utilisateur et on l'envoie via le canal à l'aide de la méthode **channel.basic_publish**.
Cette dernière contient le message dans l'attribut **body** et le nom de la file d'attente à utiliser dans l'attribut **routing_key**.
Enfin, à la fin du programme, on ferme la connexion proprement.
Voici maintenant le consommateur qui correspond :

```
from pika import BlockingConnection
from pika.connection import URLParameters

connection = BlockingConnection(
    URLParameters('amqp://test:test@localhost:5672/vhost'))
channel = connection.channel()

channel.queue_declare(queue='test?')

def callback(ch, method, properties, body):
    print(f"Received {body!r}")

channel.basic_consume(
    queue='test_1', on_message_callback=callback, auto_ack=True)

print('Waiting for messages. To exit press CTRL+C')
channel.start_consuming()
```

Les six premières lignes sont absolument identiques au producteur. Comme on ne sait pas a priori si le producteur ou le consommateur sera lancé en premier, on déclare la file d'attente dans les deux. Le premier programme lancé la créera et le second ne se contentera que de s'en assurer.
Pour le consommateur, on enregistre à l'aide de **channel.basic_consume** une fonction de rappel qui sera lancée à chaque fois que le producteur enverra de la donnée. On précise au moment de cette déclaration le nom de la file d'attente.
Le paramètre **auto_ack** permet d'informer automatiquement le producteur que sa donnée a bien été reçue.
Le consommateur est ensuite une boucle infinie qui va attendre de recevoir un message pour lancer la fonction de rappel lorsqu'il en reçoit un. Cela se fait par la dernière ligne de code.
Vous pouvez tester ces deux codes en allant dans le répertoire distribué/amqp et l'exécuter dans deux terminaux, chacun utilisant une de ces lignes :

```
$ python receiver_1.py
$ python sender_1.py
```

Une fois que vous avez fait les tests, vous pouvez ouvrir encore plus de terminaux et lancer autant de producteurs et consommateurs que vous souhaitez. Vous verrez alors que cela se comporte comme avec le paradigme client/serveur exposé dans la section ZMQ, le producteur étant le client et le consommateur le serveur : le producteur envoie de la donnée et un des consommateurs la récupère et lui répond. À chaque requête, le consommateur qui répond change.

© Editions ENI - All rights reserved

3.4 Notions importantes

Maintenant que nous avons vu comment cela fonctionne, il est temps de revenir en détail sur certains éléments.

Lorsque le producteur envoie un message, RabbitMQ le stocke dans sa file d'attente. Les messages seront traités par ordre d'arrivée (nous ne parlerons pas ici des priorités que l'on peut rattacher à un message). RabbitMQ sait lorsqu'il donne un message à un consommateur, il ne va donc pas le donner également à un autre. Le point important est que lorsque le consommateur a terminé son travail sur un message, il envoie un ACK (*acknowledgment* ou acquittement en français). À partir de là, RabbitMQ est libre de supprimer le message de sa file d'attente.

Grâce à ce mécanisme, si un des consommateurs meurt, la tâche n'est pas perdue.

Ce que l'on a utilisé est une manière d'automatiser l'ACK, avec le **`auto_ack=True`**.

Normalement, il faudrait l'envoyer à la main, ainsi :

```
def callback(ch, method, properties, body):
    print(f"Received {body!r} ... ")
    sleep(len(body))
    print(f"... {body!r} Done.")
    ch.basic_ack(delivery_tag=method.delivery_tag)
```

On va donc déclarer cette fonction de rappel ainsi :

```
channel.basic_consume(queue='test_2', on_message_callback=callback)
```

De plus, par défaut, RabbitMQ distribue les messages de manière équilibrée (comme le distributeur de cartes), sans tenir compte de l'occupation réelle de ces consommateurs. Ainsi, dans une situation avec deux consommateurs, toutes les tâches impaires iraient à l'un et les paires à l'autre. Il est donc possible, si les tâches paires sont plus complexes, par exemple, d'avoir le premier consommateur qui soit sous une charge importante alors que le second ne fait rien. Pour éviter cela, il faut utiliser le fair dispatch :

```
channel.basic_qos(prefetch_count=1)
```

Enfin, nous avons vu que nous sommes à l'abri d'une chute d'un consommateur, mais pas d'une chute de RabbitMQ. Pour cela, il faut rendre persistante la file d'attente :

```
channel.queue_declare(queue='test_2', durable=True)
```

Cette déclaration doit être consistante dans le producteur et le consommateur.

Pour utiliser cette file persistante, il faut aussi, côté producteur, rendre le message persistant :

```
from pika import BasicProperties
channel.basic_publish(exchange='', routing_key='test_2', body=data,
    properties=BasicProperties(delivery_mode=2))
```

Vous pourrez faire fonctionner le code en utilisant les commandes suivantes :

```
$ python receiver_2.py
$ python sender_2.py
```

Je vous encourage aussi à modifier **receiver_1.py** pour rajouter des temps d'attente dans la fonction de rappel et ainsi constater les changements de comportement entre les deux codes.

3.5 Publish/subscribe

Nous avons déjà vu ce patron dans la section sur ZMQ. Ici, nous l'aborderons en trois parties. D'une part, nous verrons la notion de silo (exchange), comment les utiliser et leur différence avec les files d'attente. Ensuite, nous verrons comment faire du routage, puis comment gérer des sujets (topics).
Le début du script est identique au précédent :

```
from pika import BlockingConnection, BasicProperties
from pika.connection import URLParameters

connection = BlockingConnection(
    URLParameters('amqp://test:test@localhost:5672/vhost'))
channel = connection.channel()
```

À partir de cet instant, on déclare notre silo (exchange). Ce silo permet d'isoler les messages. Seuls ceux qui travaillent dans le silo verront les messages qui y sont émis. D'un point de vue architecture, lorsque l'on souhaite séparer les messages par typologie, alors il faut utiliser cette notion de silo.
Voici comment créer un tel silo :

```
channel.exchange_declare(exchange='tests', exchange_type='fanout')
```

Le type **fanout** permet de distribuer tous les messages dans toutes les files d'attente. Cela signifie que tous les producteurs du silo émettront des messages que tous les consommateurs recevront.
Du point de vue architecture, nous irons donc créer une file d'attente pour chaque producteur et chaque consommateur pour gérer les différents temps d'attente de chaque partie de l'application distribuée.
Dans ce cas, chaque message d'un producteur, quelle que soit la file d'attente, sera rajouté à la file d'attente des consommateurs.
Il nous faut donc un nom différent pour les files d'attente, sachant que le même programme, dans notre cas, sera lancé plusieurs fois pour faire plusieurs producteurs ou plusieurs consommateurs.
Nous demanderons donc à RabbitMQ lui-même de donner un nom unique à notre file d'attente et nous la récupérons :

```
result = channel.queue_declare(queue='', exclusive=True)
queue_name = result.method.queue
print(f"queue declared: {queue_name}")
```

En effet, si tout à l'heure il était important de connaître le nom de la file d'attente pour pouvoir connecter le consommateur et le producteur à la même file, nous sommes ici dans une configuration tout autre.
Il nous faut maintenant lier la file d'attente au silo :

```
channel.queue_bind(exchange='tests', queue=queue_name)
```

© Editions ENI - All rights reserved

Ces modifications doivent être faites sur le producteur et le consommateur.
Pour envoyer un message, le producteur utilisera :

```
channel.basic_publish(exchange='tests', routing_key='', body=data)
```

Il devra préciser le silo.
Pour le consommateur, il n'y a pas de changement particulier :

```
def callback(ch, method, properties, body):
    print(f"Received {body!r} ... ")

channel.basic_consume(
    queue=queue_name, on_message_callback=callback, auto_ack=True)
```

On continue de travailler sur la file d'attente et comme elle a été liée au silo précédemment, on travaille donc dans le silo.
Pour tester ce code, vous pourrez lancer autant d'instances de producteurs et consommateurs que vous le souhaitez en lançant ce code :

```
$ python receiver_3.py
$ python sender_3.py
```

Nous allons maintenant parler du routage, puis des sujets.

3.6 Routage et sujets

Le type de silo **fanout** que nous venons de voir diffuse tous les messages à tout le monde, au sein du silo. Si nous souhaitons travailler avec des sujets, il faut changer de type :

```
channel.exchange_declare(exchange='tests_more', exchange_type='direct')
```

Tout le reste des opérations ne change pas, à part le fait que, lorsque le producteur publie, il le fait en précisant une route :

```
channel.basic_publish(exchange='tests_more', routing_key=route, body=da-
ta)
```

D'autre part, côté consommateur, nous modifions la manière de connecter le silo à la file d'attente en rajoutant une clé de routage : ainsi, nous n'écouterons que les routes qui nous intéressent :

```
channel.queue_bind(exchange='tests_more', queue=queue_name, routing_-
key=route)
```

Vous pourrez tester ce code en lançant, par exemple :

```
$ python receiver_4.py
$ python receiver_4.py SE
$ python receiver_4.py SNW
$ python receiver_5.py SN
$ python receiver_5.py WE
$ python sender_4.py
$ python sender_4.py
$ python sender_5.py
$ python sender_5.py
```

Le premier exemple (**sender_4**) vous permet de travailler avec un producteur qui publie dans une route à la fois, tandis que l'autre (**sender_5**) permet de publier dans plusieurs routes à la fois (les routes doivent alors être précisées au début du message, avec les deux-points comme séparateur).
Les deux fonctionnent parfaitement bien ensemble.
Cela se rapproche beaucoup du patron publish/subscribe que l'on a pu voir avec ZMQ. Le dernier élément que nous allons présenter ici est la notion de sujet (topic). Cette notion est un peu plus puissante que le simple routage, car elle permet de traiter des ensembles de sujets.
Par exemple, je peux avoir un producteur qui envoie des données reliées aux sujets tels que **state.down**, **state.up**, **button.down**, **button.up** et avoir des consommateurs qui s'abonnent soit aux sujets exacts, soit à un sujet tel que ***.up** ou encore **button.***.
Voici comment déclarer un silo de type topic :

```
channel.exchange_declare(exchange='tests_topics', exchange_type='topic')
```

Par rapport à l'exemple précédent, rien d'autre ne change, si ce n'est que l'utilisateur doit saisir le nom du topic puis les deux-points avant sa donnée.
Cette donnée est toujours envoyée ainsi :

```
channel.basic_publish(exchange='tests_topics', routing_key=topic, bo-
dy=data)
```

Côté consommateur, tous les sujets sont donnés en argument de la ligne de commande et les abonnements sont réalisés comme précédemment :

```
channel.queue_bind(exchange='tests_topics', queue=queue_name, routing_-
key=topic)
```

Rien d'autre n'est modifié.
Pour tester ce code, voici quelques exemples de lignes de commande :

```
$ python sender_6.py
$ python receiver_6.py \*.\*
$ python receiver_6.py button.up
$ python receiver_6.py *.down
$ python receiver_6.py button.*
$ python receiver_6.py button.up button.down state.up state.down
```

La seconde commande permet d'avoir un consommateur qui écoute tous les topics et qui peut être pratique pour vérifier l'émission de chaque message. Voici un exemple de messages à taper dans le terminal :

```
> button.up:chose
publish chose to button.up
> top.down:truc
publish truc to top.down
> chose:machin
publish machin to chose
> button.down:bidule
publish bidule to button.down
```

© Editions ENI - All rights reserved

Vous pourrez ainsi regarder ce qu'il se passe dans les consommateurs et vérifier que seuls ceux qui ont les bons sujets reçoivent les messages.

4. Celery

4.1 Présentation

Celery est un ordonnanceur de tâche :
- il reçoit ses tâches via un système de message
- il les distribue de manière optimisée via l'utilisation des workers
- chaque tâche peut s'exécuter de manière synchrone ou asynchrone
- il est conçu pour fonctionner en temps réel, mais peut aussi gérer une planification
- chaque tâche peut renvoyer un résultat (optionnellement)

Il s'agit d'un outil de haut niveau offrant des fonctions et une syntaxe pour répondre facilement à des besoins d'architecture applicative complexe.

4.2 Dans quel cas utiliser Celery ?

Le cas classique est lorsque l'on a une fonction qui doit générer un résultat, puis qui doit faire quelque chose annexe avec ce résultat et le renvoyer à l'utilisateur. Si l'on fait ces actions annexes dans le même tunnel de traitement, alors on retarde d'autant la réponse à l'utilisateur. L'idée est donc de mettre ces actions annexes dans des tâches distribuées (exécutées à côté du processus principal). Ainsi, on lance ces tâches qui seront exécutées plus tard, mais on n'attend pas qu'elles soient terminées et l'on peut continuer le processus principal et rendre la main à l'utilisateur.

Les tâches placées dans une file d'attente sont exécutées au plus tôt. Il n'y a cependant aucune garantie de la temporalité de leur exécution ni sur le fait qu'elles réussissent ou échouent.

L'expérience utilisateur est ainsi améliorée, car l'application répond plus vite, mais il doit aussi être tenu au courant qu'il y a des tâches en attente et être notifié lorsque celles-ci se terminent.

Un exemple concret serait une tâche pour envoyer un e-mail ou générer un document à partir des résultats du processus principal.

Il est possible de prévoir un moyen pour Celery de restituer le résultat, ceci permet de résoudre toutes les situations classiques où la programmation asynchrone est efficace, sauf qu'on le fait avec une solution tierce qui permet une mise à l'échelle importante et des possibilités plus intéressantes en termes d'architecture.

4.3 Installation

Comme expliqué plus haut, Celery fonctionne en recevant des messages lui demandant de réaliser une tâche. Il faut donc choisir comment ces messages sont envoyés. Les choix les plus populaires sont RabbitMQ (la solution par défaut) ou Redis.

C'est ce que l'on nomme broker.

En fonction de votre choix, vous installerez Celery de l'une de ces deux manières :

```
$ pip install celery
$ pip install celery[redis]
```

Il faudra ensuite installer le broker lui-même. Pour RabbitMQ, nous avons déjà vu la méthode dans le chapitre précédent, mais nous la répétons pour plus de simplicité.
Pour une distribution Linux :

```
# wget -O - "https://packagecloud.io/rabbitmq/rabbitmq-server/gpgkey" |
apt-key add -
# apt update
# apt install rabbitmq-server
```

Pour les autres distributions, il faudra se référer à la documentation officielle (https://www.rabbitmq.com/download.html).
Pour Redis, la procédure est à peu près similaire et pour les mêmes raisons (paquets des distributions datés ou absents) :

```
# add-apt-repository ppa:chris-lea/redis-server
# apt update
# sudo apt install redis-server
```

4.4 Configuration

Les brokers ont besoin d'être configurés. Lorsque l'on utilise PostgreSQL, par exemple, il faut créer un utilisateur avec son mot de passe ainsi qu'une base de données. On peut alors l'utiliser dans notre programme Python à partir d'une URL telle que :

```
database_url = 'postgres://user:secret@localhost:5432/db_name'
```

Avec les brokers, il s'agit de la même chose :

```
rabbit_broker_url = 'amqp://user:secret@localhost:5672/vhost'
redis_broker_url = 'redis://:secret@hostname:port/vhost_number
```

Pour Redis, il n'est pas nécessaire d'avoir une configuration particulière, si ce n'est le fait de définir un éventuel mot de passe. La base Redis n'étant jamais exposée, elle est en général protégée par l'environnement réseau lui-même. On peut également accéder Redis directement par un socket, ce qui évite de perdre du temps en négociation réseau :

```
redis_broker_url = 'redis+socket:///path/to/redis.sock?vir-
tual_host=vhost_nb
```

En ce qui concerne RabbitMQ, il y a quelques commandes à connaître, que nous répétons ici :

```
# rabbitmqctl add_user test test
# sudo rabbitmqctl add_vhost vhost
# sudo rabbitmqctl set_permissions -p vhost test ".*" ".*" ".*"
```

Une fois que le broker est configuré, on va voir comment configurer Celery depuis notre programme Python. Pour cela, on crée un fichier dédié, usuellement nommé **celery-config.py** qui contient les lignes suivantes :

```
broker_url="amqp://test:test@localhost:5672/vhost"
```

© Editions ENI - All rights reserved

```
result_backend="rpc://"

task_serializer = 'json'
result_serializer = 'json'
accept_content = ['json']
result_accept_content = ['json']
timezone = 'Europe/Paris'
enable_utc = True
```

Vous aurez reconnu la première ligne qui permet de définir le broker. La seconde permet de définir le système de retour, sans lequel le programme Python ne peut avoir de retours sur les tâches qu'il lance (encore une fois, il est des tâches qui ne demandent pas de retour).
Dans le cas de l'utilisation de Redis en tant que broker, il aurait fallu utiliser :

```
backend='redis://localhost'
```

Les lignes suivantes permettent de spécifier le format des messages d'entrée ou de sortie et le paramétrage des fuseaux horaires.
Il existe de très nombreux autres paramètres permettant de gérer finement les brokers, backends, tâches, ou priorités, lesquels sont détaillés dans la documentation officielle (http://docs.celeryproject.org/en/latest/userguide/configuration.html).
Voici un exemple :

```
task_annotations = {
    'tasks.add': {'rate_limit': '12/m'}
}
```

Ceci ne signifie pas que l'on peut faire douze requêtes puis attendre la fin de la minute pour en lancer douze autres, mais bel et bien qu'il ne sera exécuté qu'une seule requête toutes les 5 secondes (un douzième d'une minute).
Voici maintenant comment démarrer Celery :

```
$ celery -A tasks worker --loglevel=info
```

On pourra visualiser dans le terminal la liste des tâches prises en compte par Celery (et ainsi détecter tout manquement) et visualiser tous les appels.
Il est aussi possible de lancer Celery en daemon, ce qui est indispensable sur un serveur : (http://docs.celeryproject.org/en/latest/userguide/daemonizing.html).

4.5 Utilisation

On se propose de créer :

```
>>> from tasks import add, sub, mul, div
>>> result = add.delay(2, 3)
>>> print(result)
<AsyncResult: ace21342-3861-41f4-94dd-972b884104db>
>>> result.get(timeout=1)
5
>>> result = sub.delay(2, 3)
>>> result.get(timeout=1)
-1
```

```
>>> result = mul.delay(2, 3)
>>> result.get(timeout=1)
6
>>> result = div.delay(2, 3)
>>> result.get(timeout=1)
0.6666666666666666
```

Voici maintenant un exemple qui produit une exception :

```
>>> result = div.delay(2, 0)
>>> result.get(propagate=False)
ZeroDivisionError('division by zero')
>>> print(result.traceback)
Traceback (most recent call last):
  File [...] in [...]
    return x / y
ZeroDivisionError: division by zero
```

Si on n'utilise pas **propagate=False**, alors l'exception sera jouée lors de l'appel à la méthode **get**. Le fait d'utiliser **propagate=False** permet de la gérer sans mettre en œuvre une boucle d'exception.
On se propose maintenant de tester le paramètre d'annotation. Voici un script permettant de visualiser ses effets en activant et désactivant ce paramètre :

```
from time import sleep, time
from tasks import add


def test():
    t0 = time()
    for i in range(12):
        result = add.delay(i, i)
        while not result.ready():
            sleep(0.001)
        print(time()-t0, result.get())

if __name__ == "__main__":
    test()
```

Voici un extrait du résultat sans le paramètre :

```
0.14225435256958008 0
0.1623837947845459 2
0.16633820533752441 4
0.17165613174438477 6
```

Et voici un extrait du même résultat après avoir remis le paramètre et relancé Celery :

```
0.23764896392822266 0
6.53479790687561 2
10.502197027206421 4
16.507360696792603 6
```

© Editions ENI - All rights reserved

4.6 Monitorer

L'intérêt d'utiliser un outil aussi puissant que Celery est, outre la scalabilité, les possibilités de monitorer. Ceci peut se faire en ligne de commande :

```
$ celery -A proj inspect active
$ celery -A proj inspect scheduled
$ celery -A proj inspect reserved
$ celery -A proj inspect revoked
$ celery -A proj inspect registered
$ celery -A proj inspect stats
```

Ces commandes permettent de visualiser les tâches qui sont dans un état particulier. Ces états sont, respectivement :
- actif : la tâche est en train de tourner
- programmé : la tâche a déjà reçu un ETA (estimation de fin)
- réservé : la tâche est assignée à une file d'attente (mais n'a pas encore d'ETA)
- révoqué : la tâche ne sera pas exécutée
- enregistré : la tâche est reçue, mais pas encore placée dans une file d'attente

Connaître ces statuts est indispensable pour suivre les tâches et comprendre ce qu'il se passe en cas de soucis.
Il existe également plusieurs autres façons de monitorer les tâches, mais la toute dernière, une version web, déprécie les autres. Elle s'installe ainsi :

```
$ pip install flower
```

Pour la voir en action, il faut lancer Celery de l'une de ces deux manières :

```
$ celery -A tasks flower
$ flower -A tasks
```

Pour aller plus loin avec Celery :
- http://docs.celeryproject.org/en/latest/userguide/index.html
- http://docs.celeryproject.org/en/latest/userguide/configuration.html

5. Crossbar

5.1 Présentation

Pour terminer ce chapitre sur la programmation distribuée, nous allons introduire WAMP (*Web Application Messaging Protocol*). Il s'agit d'un standard ouvert de WebSocket qui permet à deux applications de s'échanger des messages RPC (*Remote Procedure Call*, soit appel distant de procédure) ou de type Publish/Subscribe (que nous avons déjà croisé deux fois dans ce chapitre).
La conception d'origine de ce protocole a été réalisée par crossbar.io en 2012 et est relativement utile et facile à mettre en place pour des projets web.

La solution présentée ici utilise massivement la bibliothèque asynchrone de Python. Il est donc conseillé d'avoir parcouru le chapitre avancé sur l'asynchrone avant de s'attaquer à cette partie.
Pour installer les outils nécessaires :

```
$ pip install autobahn
```

5.2 WebSocket

En guise d'introduction, nous allons réaliser un WebSocket de type Echo. Ce dernier va donc recevoir de la donnée et la renvoyer telle quelle. Voici ce serveur :

```
import json
import logging
import coloredlogs
from autobahn.asyncio.websocket import (
    WebSocketServerProtocol,
    WebSocketServerFactory)


logger = logging.getLogger("notifications")
coloredlogs.install(level='DEBUG', logger=logger)


class TestServerProtocol(WebSocketServerProtocol):

    def onConnect(self, request):
        logger.info("Client connecting: {0}".format(request.peer))

    def onOpen(self):
        logger.info("WebSocket connection open.")

    def onMessage(self, payload, isBinary):
        if isBinary:
            result = self.onMessageBinary(payload)
        else:
            payload = payload.decode("utf-8")
            try:
                payload = json.loads(payload)
            except:
                result = self.onMessageText(payload)
            else:
                result = json.dumps(self.onMessageJson(payload))
            result = result.encode("utf-8")
        self.sendMessage(result, isBinary)

    def onMessageBinary(self, payload):
        logger.debug("Binary message received: {0}
bytes".format(len(payload)))
        return payload
```

© Editions ENI - All rights reserved

```
    def onMessageText(self, payload):
        logger.debug("Text message received: {0}".format(payload))
        return payload

    def onMessageJson(self, payload):
        logger.debug("JSON message received: {0}".format(payload))
        return payload

    def onClose(self, wasClean, code, reason):
        logger.info("client exiting, {0}".format(self.peer))
        logger.info("WebSocket connection closed: {0}".format(reason))


if __name__ == '__main__':
    import asyncio

    factory = WebSocketServerFactory(u"ws://127.0.0.1:9000")
    factory.protocol = TestServerProtocol

    event_loop = asyncio.get_event_loop()
    coroutine = event_loop.create_server(factory, '0.0.0.0', 9000)
    server = event_loop.run_until_complete(coroutine)

    try:
        event_loop.run_forever()
    except KeyboardInterrupt:
        pass
    finally:
        server.close()
        event_loop.close()
```

Les points clés sont les suivants : on crée un objet de type **`Protocol`** qui hérite de **`WebSocketServerProtocol`**. On n'a plus ensuite qu'à remplir les hooks **`onConnect`**, **`onOpen`**, **`onClose`** et **`onMessage`**. Pour chacun, on écrit un message dans le terminal et pour le dernier, on va analyser le contenu de la donnée. On appelle alors ensuite une autre méthode, en fonction de ce type : binaire, JSON ou texte simple.

Pour créer un client WebSocket, il faut procéder d'une manière relativement similaire :

```
import asyncio
import json
import logging
import coloredlogs

from autobahn.asyncio.websocket import (
    WebSocketClientProtocol,
    WebSocketClientFactory)
```

```
logger = logging.getLogger("notifications")
coloredlogs.install(level='DEBUG', logger=logger)


class TestClientProtocol(WebSocketClientProtocol):

    def onConnect(self, response):
        logger.info("Server connected: {0}".format(response.peer))

    async def onOpen(self):
        logger.info("WebSocket connection open.")

        while True:
            self.sendMessage(json.dumps({"k": "v"}).encode('utf8'))
            self.sendMessage("Hello, world!".encode('utf8'))
            self.sendMessage(b"\x00\x01\x03\x04", isBinary=True)
            await asyncio.sleep(5)

    def onMessage(self, payload, isBinary):
        if isBinary:
            self.onMessageBinary(payload)
        else:
            payload = payload.decode("utf-8")
            try:
                payload = json.loads(payload)
            except:
                self.onMessageText(payload)
            else:
                self.onMessageJson(payload)

    def onMessageBinary(self, payload):
        logger.debug("Binary message received: {0}
bytes".format(len(payload)))
        return payload

    def onMessageText(self, payload):
        logger.debug("Text message received: {0}".format(payload))
        return payload

    def onMessageJson(self, payload):
        logger.debug("JSON message received: {0}".format(payload))
        return payload

    def onClose(self, wasClean, code, reason):
        logger.info("WebSocket connection closed: {0}".format(reason))
```

© Editions ENI - All rights reserved

```
if __name__ == '__main__':

    factory = WebSocketClientFactory(u"ws://127.0.0.1:9000")
    factory.protocol = TestClientProtocol

    event_loop = asyncio.get_event_loop()
    coroutine = event_loop.create_connection(factory, '127.0.0.1', 9000)
    event_loop.run_until_complete(coroutine)
    event_loop.run_forever()
    event_loop.close()
```

On utilise les mêmes hooks et c'est au moment du **onOpen** que l'on va envoyer trois messages : un en JSON, un texte et un autre en binaire. Comme tout à l'heure, le **onMessage** permet de réagir à la réception d'un message, c'est-à-dire dans ce cas à la réception d'une réponse de la part du serveur.
Tout ceci est testable en utilisant le code disponible dans le répertoire wamp/wamp_proto_1_websocket/ et en ouvrant deux terminaux avec la commande :

```
$ python server.py
$ python client.py
```

Il est ainsi possible de tester ce WebSocket. Vous pourrez modifier le client pour envoyer différemment vos messages. Ceci étant dit, l'intérêt du WebSocket est surtout réalisé lorsqu'il est utilisé depuis une interface client web. C'est pourquoi il est proposé un code permettant d'arriver à ces fins. Sans détailler tout le code HTML, on va juste présenter les trois boutons :

```
<button onclick='sendText();'>Send Text Message</button>
<button onclick='sendJson();'>Send Json Message</button>
<button onclick='sendBinary();'>Send Binary Message</button>
```

Ainsi que le code JavaScript relatif à ces boutons. Pour commencer, envoyons du texte :

```
function sendText() {
    if (isopen) {
        socket.send("Hello, world!");
        console.log("Text message sent.");
    } else {
        console.log("Connection not opened.")
    }
    };
```

isopen est une variable qui permet de savoir si le WebSocket est ouvert. On utilise ensuite un socket pour envoyer un message (nous verrons en dessous comment gérer ce socket). Pour envoyer du JSON, cela est relativement similaire.

```
function sendJson() {
    if (isopen) {
        socket.send('{"k": "v"}');
        console.log('{"k": "v"}');
    } else {
        console.log("Connection not opened.")
```

```
    }
};
```

Contrairement à Python, il y a, en JavaScript, une différence entre les simples et doubles guillemets. Par contre, la syntaxe d'un dictionnaire, si l'on respecte cela, reste assez identique à ce que l'on trouve en Python. Je ne fais pas ici de la conversion entre un objet JavaScript et du JSON pour simplifier. Par contre, pour envoyer des octets, on est obligé de passer par là :

```
function sendBinary() {
    if (isopen) {
        var buf = new ArrayBuffer(32);
        var arr = new Uint8Array(buf);
        for (i = 0; i < arr.length; ++i) arr[i] = i;
        socket.send(buf);
        console.log("Binary message sent.");
    } else {
        console.log("Connection not opened.")
    }
};
```

Il faut ici créer un buffer pour le remplir octet par octet, ce qui peut être un peu complexe si l'on n'est pas familier avec le langage. Heureusement, on utilise rarement le format binaire.

Voici maintenant comment gérer la création du socket :

```
var socket = null;
var isopen = false;
window.onload = function() {
    socket = new WebSocket("ws://127.0.0.1:9000");
    socket.binaryType = "arraybuffer";

    socket.onopen = function() {
        console.log("Connected!");
        isopen = true;
}

    socket.onmessage = function(e) {
        if (typeof e.data == "string") {
            try {
                var json = JSON.parse( e.data );
                console.log("JSON message received: " + e.data);
                display_message("json_messages", json);
            } catch(e) {
                console.log("Text message received: " + e.data);
                display_message("text_messages", e.data);
            }
        } else {
            var arr = new Uint8Array(e.data);
            var hex = '';
```

© Editions ENI - All rights reserved

```
            for (var i = 0; i < arr.length; i++) {
                hex += ('00' + arr[i].toString(16)).substr(-2);
            }
            display_message("binary_messages", hex);
            console.log("Binary message received: " + hex);
        }
    }

    socket.onclose = function(e) {
        console.log("Connection closed.");
        socket = null;
        isopen = false;
    }
};
```

On retrouve ici les mêmes hooks et un code similaire au code Python précédent. La seule différence est que les messages ne sont pas envoyés automatiquement lors de l'ouverture de la connexion, mais lorsque l'utilisateur clique sur le bouton correspondant.
Vous pourrez aller dans le fichier **index.html** pour visualiser l'intégralité du code et pour l'adapter à vos besoins nécessaires.
De manière à jouer avec ce code, il est possible d'ouvrir plusieurs terminaux et plusieurs fenêtres web, et d'utiliser plusieurs clients pour voir le résultat obtenu.
Il existe aussi un exemple plus avancé qui permet d'utiliser une base de données SQLite pour garder trace de l'activité des utilisateurs. On sait quand ils arrivent sur une page et quand ils en partent juste en utilisant les différents hooks de service web.
Cet exemple plus complet n'est pas détaillé dans ce chapitre, car il n'apporte rien de plus au niveau des principes du WebSocket, mais il apporte plus au niveau de l'utilisation de SQLite et du langage JavaScript. Ce code se trouve dans le répertoire wamp_proto_2_websocket. Vous noterez qu'il faut au préalable créer la base de données :

```
$ python init_db.py
```

Il existe un autre répertoire wamp/wamp_proto_3_websocket_notifications qui, comme son nom l'indique, donne des pistes pour maintenir une notification. Dans ce cas-là, nous décidons, après l'ouverture d'une session WebSocket, d'envoyer toutes les secondes un message avec le nombre de sessions ouvertes au client. Comme chaque client garde sa session ouverte, tout le monde reçoit cette information.
Ce n'est pas à proprement parler une notification, car elle n'intervient pas au moment d'un changement, mais elle suffit à avoir l'information côté client.
Dans un cadre idéal, il faudrait utiliser cette solution en conjonction avec ce que l'on a vu précédemment, à savoir un publish/subscribe, mais l'approche reste intéressante.
Là encore, on ne détaille pas l'intégralité du code dans le livre, car il n'y a pas de nouveautés techniques, mais je vous encourage à le lire et le tester. Voici les deux étapes essentielles :

```
async def onOpen(self):
    logger.info("WebSocket connection open.")
    while True:
        conn = sqlite3.connect(DB_NAME)
        cursor = conn.cursor()
```

```
        cursor.execute(
            "SELECT count(*) AS active_count FROM sessions WHERE end is
null")
        active_count = cursor.fetchone()[0]
        conn.close()
        logger.debug("There is {} active sessions.".format(active_count))
        self.sendMessage(json.dumps({
            "topic": "active_count",
            "active_count": active_count
        }).encode("utf-8"))
        await asyncio.sleep(1)
```

Côté client Python, il n'y a pas de changement, on traite tous les messages reçus de manière identique. Au niveau JavaScript, par contre, on a ceci :

```
if (json.topic == "active_count") {
    document.getElementById("active_count").innerText=json.active_count;
} else {
    display_message(json);
}
```

Ceci suffit à afficher l'information au bon endroit.

5.3 Publish/subscribe

Pour réaliser ce patron de conception, nous allons prendre l'exemple d'une horloge. Voici un serveur qui va, toutes les secondes, publier l'heure :

```
import asyncio
from datetime import datetime
import logging
import coloredlogs
from autobahn.asyncio.wamp import ApplicationSession, ApplicationRunner


logger = logging.getLogger("notifications")
coloredlogs.install(level='DEBUG', logger=logger)


class Component(ApplicationSession):

    async def onJoin(self, details):
        while True:
            now = datetime.now().strftime("%d %B %Y, %Hh %Mm %Ss")
            logger.info("publish: com.test.topic_clock {}".format(now))
            self.publish("com.test.topic_clock", now)
            await asyncio.sleep(1)


if __name__ == '__main__':
    runner = ApplicationRunner("ws://127.0.0.1:9090/ws", "test")
    runner.run(Component)
```

© Editions ENI - All rights reserved

Pour cela, nous créons simplement un composant qui va publier une donnée (textuelle) associée à un sujet précis (dont nous choisissons librement le nom). Cette publication se fait à l'intérieur d'une boucle infinie.
Une fois ce composant réalisé, il est lancé à l'aide d'un peu de code caractéristique, présenté à la fin du fichier.
Voici maintenant un client Python permettant de recevoir cette donnée :

```
import asyncio
import logging
import coloredlogs
from autobahn.asyncio.wamp import ApplicationSession, ApplicationRunner

logger = logging.getLogger("notifications")
coloredlogs.install(level='DEBUG', logger=logger)


class Component(ApplicationSession):

    async def onJoin(self, details):

        def on_event(i):
            logger.info("Now: {}".format(i))

        await self.subscribe(on_event, u'com.test.topic_clock')

    def onDisconnect(self):
        asyncio.get_event_loop().stop()


if __name__ == '__main__':
    runner = ApplicationRunner("ws://127.0.0.1:8080/ws", "test")
    runner.run(Component)
```

Ici, au moment où la connexion au WebSocket est réalisée, on souscrit au sujet qui nous intéresse, l'horloge.
À partir de là, on reçoit l'ensemble des messages émis. On gère aussi la déconnexion de manière à fermer le WebSocket proprement et terminer le programme.
Et comme précédemment, il y a une version web du client, en JavaScript :

```
try {
    var autobahn = require('autobahn');
} catch (e) {
     console.log("Il manque une bibliothèque");
}

var connection = new autobahn.Connection({
    url: 'ws://127.0.0.1:8080/ws',
    realm: 'test'}
```

```
);

connection.onopen = function (session) {
    function on_clock_event(args) {
        document.getElementById("clock").innerText=args[0];
        console.log("Message received: ", args[0]);
    }
    session.subscribe('com.test.topic_clock', on_clock_event);
};

connection.open();
```

Là encore, on est très similaire au code client Python. Il faut utiliser une bibliothèque particulière, puis créer une connexion et souscrire au service d'horloge en passant une fonction de rappel qui va afficher l'heure dans la console JavaScript.

On a ici les bases minimales pour commencer à monter en compétences sur ce genre d'outils.

Pour aller plus loin, il faut visiter le site officiel (crossbar.io) et lire les différents codes présents dans les exemples donnés dans les dépôts GIT.

© Editions ENI - All rights reserved

!

@atexit, 144

A

Adaptateur, 26, 53
aiohttp, 374
AMP, 369
AMQP, 418
Appels à des fonctions
    planifier, 305
Appels système, 364
Asynchrone, 319, 400
    boucle événementielle, 292
    compréhension, 278
    contexte, 277
    coroutine, 270
    exécution, 400
    générateurs, 278
    HTTP, 374
    itération, 278
    parser, 327
    tâches, 274, 276
    télécharger, 326
asyncio.sleep, 276
Attente, 286
await, 272
Awaitable, 276

B

bash
    cat, 161
    cd, 158
    cp, 161, 162, 163
    cp -p, 162
    cp -r, 163
    du -sk, 160
    head, 161
    ls -R, 158
    mkdir, 158
    mv, 160, 163
    pwd, 158
    tail, 161
Boucle événementielle, 302
bsh
    rmdir, 160
Bz2, 151

C

Calcul scientifique, 227
Callbacks, 302
Celery, 425
Chaîne de responsabilité, 42
Chemin, 142
Client/Serveur TCP, 366
Commande, 43
    externe, 164
Composite, 32
Compression, 148
concurrent.futures, 402
contextvars, 289
Cookies, 200

© Editions ENI - All rights reserved

Coroutine, 270
    asyncio.sleep, 276
    communication, 313
    exécution, 271, 273
cProfile, 121
Crossbar, 429

D

Débogage, 120, 293
Décompression
    Voir Compression
Décorateur, 34
Démon, 398
Design Pattern
    Voir Motifs de conception
DNS, 363
Documentation, 114
    externe, 115
    interne, 114
    Sphinx, 115

E

etree, 66
Exception, 282, 298
Exécuteur, 329
ezodf2, 91

F

Fabrique, 17, 56
    abstraite, 20
Façade, 36

Fichier
comparer deux fichiers, 142
de configuration, 145
recherche, 163
removedirs, 160
rename, 160
rmdir, 160
rrenames, 160
touch, 160
Fil d'exécution, 263
Flux, 359
Fonction
de rappel, 52
de retour, 302
FTP, 200
Future, 295
annulation, 297

G

Générateur, 325
Gevent, 363
GIL (Global Interpreter Lock), 264
glob, 163
Groupe, 138
Gzip, 150

H

HTTP, 195

© Editions ENI - All rights reserved

I

IMAP, 205
IMAPS, 205
Introspection, 278
IRC, 212
Itérateur, 45

L

Lock, 385

M

Matrice, 236
Mémento, 47
Monitorer, 429
Monteur, 21
Motifs de conception, 15
    comportement, 42
    création, 17
    structuration, 26
multiprocessing, 390

N

NNTP, 211
NumPy, 229

O

Observateur, 49
ØMQ, 407

OpenDocument, 90
tableur, 91
texte, 91
os, 136
os.walk, 158

P

Parcourir
Voir os.walk
Passeur d'arguments, 166
PDF, 79
Pickle, 146
pipeline, 416
Poids-mouche, 38
Pont, 29
POP, 204
POPS, 204
Processus, 137, 262, 390, 393, 398
communication interprocessus, 395
démon, 398
fil d'exécution, 263
Lock, 393
paralléliser un travail, 393
partage de données, 397
synchronisation, 393
Profilage, 120
profile, 121
Programmation
asynchrone, 265, 269, 318, 331, 363, 365
bloquante, 259
concurrente, 259, 267
distribuée, 266, 407
non bloquante, 264
parallèle, 265, 318, 377
réseau, 135

© Editions ENI - All rights reserved

système, 135
Protocole, 331, 335, 372
réseau, 352
SSL, 346
TCP, 338, 359
UDP, 349
Prototype, 23
Proxy, 40
HTTP, 199
pstats, 121
Publish/Subscribe, 411, 422, 436
Push/Pull, 415
Pyro, 220

Q

Queue, 381

R

RabbitMQ, 418
Répertoire, 158
chdir, 158
getcwd, 158
makedirs, 158
mkdir, 158
ReportLab, 80
Ressources CPU, 260
REST, 217
Résultat, 282

S

Sémaphore, 388
    Voir Tâche
Serveur
    TCP, 191
    UDP, 193
Service web, 217
SFTP, 200
shutil, 153, 163
Signaux, 306
Singleton, 17
SMTP, 206
SMTPS, 206
SOAP, 218
Sphinx, 115
SSH, 202
Stratégie, 51
subprocess, 164, 353
Synchronisation, 309
    conditionnelle, 386

T

Tâche, 377, 380, 382
    annuler, 283
    file, 381
    sémaphore, 388
    synchronisation, 384, 386
    verrou, 386
Tarfile, 148
TCP, 185
Test, 103
    de non-régression, 107
    de performance, 110
    fonctionnel, 109

© Editions ENI - All rights reserved

intégration continue, 113
unitaire, 103
unittest, 106
Thread
Voir Tâche
Threading, 377
Timeouts, 286
Transport, 331
Trollius, 372
Twisted, 366

U

UDP, 189
urllib, 197
Utilisateur, 138
Utilitaire, 55

V

Visiteur, 48

W

Wake-on-LAN, 215
WebSocket, 430

X

XML
avantages, 61
DOM, 62, 66
DTD, 64

HTML, 74
inconvénients, 61
lxml, 63
SAX, 62
schéma, 60
schéma RNG, 65
Schematron, 65
terminologie, 59
valider, 63
XPath, 70
XSD, 64
XSLT, 72

Z

ZCA
Voir Zope Component Architecture
Zipfile, 151
Zope Component Architecture, 53

© Editions ENI - All rights reserved

Python 3

Traitement de données et techniques de programmation

Ce livre sur le langage Python 3 (en version 3.8 beta au moment de l'écriture) s'adresse à tout **professionnel de l'informatique**, **ingénieur**, **étudiant**, **enseignant** ou même **autodidacte** qui souhaite maîtriser ce langage très abouti. L'objectif de ce livre est de montrer au lecteur **ce que l'on peut faire avec le langage**, du traitement de données à la création d'un site web, en passant par la gestion du système et du réseau. Pour bien appréhender son contenu, la lecture du livre dont il est la suite, *Python 3 - Les fondamentaux du langage* chez le même éditeur, est recommandée.

La **première partie** couvre le **traitement de données** avec la manipulation de fichiers, les fichiers de configuration, les formats d'import/export, la gestion de la compression ou encore les flux XML ou la génération de documents.

La **seconde partie** présente la programmation **système**, la programmation **réseau**, la programmation **web** et la programmation **scientifique**, qui sont autant de niches dans lesquelles le langage Python excelle par sa simplicité d'utilisation autant que par sa couverture fonctionnelle exceptionnelle.

La **troisième partie** présente tous les outils de programmation concurrente, qu'il s'agisse de programmation **asynchrone**, **parallèle** ou **distribuée**. On notera que l'accent est mis sur de nombreuses techniques asynchrones, car il s'agit de l'un des points majeurs de l'évolution récente du langage Python.

Le **code source** des exemples du livre est **intégralement téléchargeable** sur www.editions-eni.fr pour permettre au lecteur de tester le programme et de le modifier à sa guise de manière à faire ses propres expériences.

Sur **www.editions-eni.fr** :
➜ Le code source des exemples.

Expert technique Python/Django et Web Backend/Frontend, **Sébastien CHAZALLET** intervient en tant qu'indépendant sur de longues missions de développement, d'audit, d'expertise et de formation (www.formation-python.com, www.inspyration.fr). Ses réalisations portent sur des développements autour de Python sur des projets de grande envergure, essentiellement des applications intranet sur mesure avec Django, mais également sur des applications de bureau, des scripts système, la création de sites web ou de e-commerce. À travers ce livre, il a su faire profiter le lecteur de sa parfaite maîtrise du langage Python dans sa dernière version et de l'expérience terrain acquise sur ses différentes missions.

Pour plus d'informations :

39 €

ISBN : 978-2-409-02292-0