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. Écrire des scripts système

2.1 Appréhender son système d'exploitation

2.1.1 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.

2.1.2 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.

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.

2.1.3 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
```

2.1.4 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', […]}
```

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)
```

2.1.5 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'
```

2.2 Gestion d'un fichier

2.2.1 Ouvrir un fichier

Python étant à l'origine conçu pour réaliser des opérations système, il est naturellement pourvu d'outils très fournis, très simples d'utilisation et très efficaces. En un mot, pythoniques. Et ils ont été améliorés tout au long de l'évolution du langage. Voici la manière basique d'ouvrir un fichier :

```
>>> with open('test.txt') as f:
...     pass # Travailler sur le contenu du fichier
...
```

Que représente **f** ?

```
>>> with open('test.txt') as f:
...     type(f)
...     type.mro(type(f))
...     dir(f)
...
<class '_io.TextIOWrapper'>
[<class '_io.TextIOWrapper'>, <class '_io._TextIOBase'>, <class
'_io._IOBase'>, <class 'object'>]
['_CHUNK_SIZE', '__class__', '__delattr__', '__doc__',
'__enter__', '__eq__', '__exit__', '__format__', '__ge__',
'__getattribute__', '__getstate__', '__gt__', '__hash__',
'__init__', '__iter__', '__le__', '__lt__', '__ne__', '__new__',
'__next__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'_checkClosed', '_checkReadable', '_checkSeekable',
'_checkWritable', 'buffer', 'close', 'closed', 'detach',
'encoding', 'errors', 'fileno', 'flush', 'isatty',
'line_buffering', 'name', 'newlines', 'read', 'readable',
'readline', 'readlines', 'seek', 'seekable', 'tell', 'truncate',
'writable', 'write', 'writelines']
```

La variable **f** est une structure qui correspond à une entrée/sortie (descripteur de fichier) et qui peut être utilisée comme un générateur.

Nous allons voir comment utiliser cette variable pour répondre à différents besoins.

Avant cela, il est utile de préciser que **open** prend deux autres paramètres qui sont le type d'accès et la taille du buffer à utiliser avec 0 pour aucun et 1 pour la valeur par défaut.

Symbole	Signification	Symbole	Signification
r	Lecture seule (défaut)	t	Mode textuel (par défaut)