L'art du scripting

Comprendre les langages de script Korn Shell, Perl, Python, Visual Basic Scripting et Windows PowerShell

Kais Ayari

EYROLLES

L'art du scripting

K. Ayari

PowerShell Guru et expert en scripting, **Kais Ayari** a pensé et conçu l'architecture et le prototype d'une version de PowerShell sous Linux/Unix, et il développe notamment des processus d'automatisation intelligents. Il est l'un des meilleurs experts sur la technologie PowerShell, maîtrisant parfaitement le modèle et la logique qui lui sont liés. Auteur de *Scripting avancé avec Windows PowerShell* aux éditions Eyrolles, il a formé de nombreux architectes, ingénieurs et administrateurs système et réseau.

Les langages de script permettent d'enchaîner un certain nombre d'opérations, chacune pouvant être traitée de manière isolée. Ils interagissent aussi bien avec le système d'exploitation qu'avec les applications dont ce dernier constitue le socle. En comparant les langages de script KSH, Perl, Python, VBScript et Windows PowerShell, nous verrons comment les utiliser au mieux en fonction du contexte.

Une étude comparée des cinq principaux langages de script

L'objectif premier de cet ouvrage est de souligner avec force l'importance du scripting pour que le lecteur prenne conscience de l'efficacité d'un changement de ses modes de compréhension. Le livre s'articule autour de cinq grands langages de script que sont Korn Shell, Perl, Python, Visual Basic Scripting et Windows PowerShell.

À travers leur étude comparée, les développeurs et administrateurs pourront s'approprier plus efficacement tous les langages qui sont à leur disposition, le but étant d'adopter une vision dialogique pour une meilleure efficience et l'efficacité qui en découle.

En outre, étudier plusieurs langages de script à partir d'une vingtaine de points de comparaison aidera le lecteur à comprendre la résolution de problèmes à travers une multitude d'angles de vue. La logique plurielle est celle que les informaticiens doivent définitivement adopter pour atteindre leurs objectifs.

Un livre de référence pour les administrateurs système et réseau

Quelle que soit la catégorie de lecteurs, débutants ou non, la logique de fonctionnement du code qui sera exposée ici sera expliquée et commentée. Ainsi, chacun évoluera de manière sereine, puisque chaque construction étudiée le sera à partir d'une analyse de la syntaxe ainsi que d'exemples l'illustrant.

Au sommaire

Historique des langages de script • **Les éléments de langage** • Les commentaires • Les variables • Les opérateurs • Les fonctions et commandes natives • Les structures conditionnelles • Les tableaux et dictionnaires • Les boucles • Les fonctions • Les classes et objets • Les modules • La gestion d'erreurs • **Le scripting en pratique** • Les expressions régulières • La gestion de fichiers • Windows Management Instrumentation • La gestion des services • La gestion des journaux d'événements • La gestion d'une interface réseau • Manipuler un fichier XML • La gestion de l'Active Directory.

À qui s'adresse cet ouvrage ?

– Aux administrateurs et ingénieurs système et réseau qui cherchent à résoudre les problèmes de différentes manières
– Aux développeurs qui souhaitent en savoir plus sur le domaine du scripting, et les bénéfices que cela peut leur apporter
– Aux informaticiens pour qu'ils prennent conscience des enjeux du scripting et des solutions qu'il fournit

Code éditeur : G14038
ISBN : 978-2-212-14038-5

9 782212 140385

Conception : Nord Compo

L'art du scripting

Comprendre les langages de script Korn Shell, Perl, Python, Visual Basic Scripting et Windows PowerShell

DANS LA MÊME COLLECTION

C. DELANNOY. – **Le guide complet du langage C**.
N°14012, 2014, 844 pages.

M. KABAB, R. GOETTER. – **Sass et Compass avancé**.
N°13677, 2013, 280 pages.

W. BORIES, O. MIRIAL, S.PAPP. – **Déploiement et migration Windows 8**.
N°13645, 2013, 480 pages.

W. BORIES, A. LAACHIR, D. THIBLEMONT, P. LAFEIL, F.-X VITRANT.
– **Virtualisation du poste de travail Windows 7 et 8 avec Windows Server 2012**.
N°13644, 2013, 218 pages.

J.-M. DEFRANCE. – **jQuery-Ajax avec PHP**.
N°13720, 4e édition, 2013, 488 pages.

L.G MORAND, L. VO VAN, A. ZANCHETTA.
– **Développement Windows 8 - Créer des applications pour le Windows Store**.
N°13643, 2013, 284 pages.

Y. GABORY, N. FERRARI, T. PETILLON. – **Django avancé**.
N°13415, 2013, 402 pages.

P. ROQUES. – **Modélisation de systèmes complexes avec SysML**.
N°13641, 2013, 188 pages.

SUR LE MÊME THÈME

K. AYARI. – **Scripting avancé avec Windows PowerShell**.
N°13788, 2013, 358 pages.

J.-F. BOUCHAUDY. – **Linux administration – Tome 1**.
N°14082, 3e édition, 2014, 430 pages.

L. BLOCH, C. WOLFHUGEL, N. MAKAREVITCH, C. QUEINNEC, H. SCHAUER.
– **Sécurité informatique**.
N°13737, 4e édition, 2013, 350 pages.

P.-L. REFALO. – **La sécurité numérique de l'entreprise**.
N°55525, 2012, 296 pages.

D. MOUTON. – **Sécurité de la dématérialisation**.
N°13418, 2012, 314 pages.

L'art du scripting

Comprendre les langages de script Korn Shell, Perl, Python, Visual Basic Scripting et Windows PowerShell

Kais Ayari

EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Table des matières

Avant-propos 1
Les objectifs de ce livre 1
Les prérequis 2
Qu'est-ce qu'un langage de script ? 3
L'exercice de comparaison comme vision dialogique 3
La composition de l'ouvrage 5
Remerciements 5

CHAPITRE 1
Un peu d'histoire 7
KSH 7
Perl 8
Python 8
VBS 9
Windows PowerShell 10

PARTIE 1
Les éléments de langage 11

CHAPITRE 2
Les commentaires 13
KSH 13
Commenter une ligne 13
Commenter un bloc de lignes 14
Perl 14
Commenter une ligne 15
Commenter un bloc de lignes 16
Python 17
Commenter une ligne 18
Commenter un bloc de lignes 18

VBS 19
Commenter une ligne 19
Commenter un bloc de lignes 20
Windows PowerShell 21
Commenter une ligne 21
Commenter un bloc de lignes 22

CHAPITRE 3
Les variables 23
KSH 23
Déclarer une variable 23
Les variables spéciales 24
Les variables d'environnement 24
Perl 25
Les variables scalaires 25
Les variables tableaux 26
Les variables dictionnaires 27
Les variables spéciales 28
Python 28
Créer une variable 29
Les références 29
VBS 31
Déclarer une variable 31
Les variables tableaux 32
Windows PowerShell 33
Créer une variable 33
Les variables automatiques 34
Les variables d'environnement 34
Les variables de préférence 35

CHAPITRE 4
Les opérateurs 37
KSH 37
Les opérateurs arithmétiques 38
Les opérateurs logiques 38
Les opérateurs de comparaison 38
Les opérateurs d'affectation 38
Les opérateurs de test sur des fichiers 39
Les opérateurs de redirection 39
Perl 39
Les opérateurs arithmétiques 40

Les opérateurs logiques 40
Les opérateurs de comparaison 40
Les opérateurs d'affectation 40
Python 41
Les opérateurs arithmétiques 41
Les opérateurs logiques 41
Les opérateurs de comparaison 41
Les opérateurs d'affectation 42
Les opérateurs de type 42
Les opérateurs d'appartenance 42
VBS 42
Les opérateurs arithmétiques 42
Les opérateurs logiques 43
Les opérateurs de comparaison 43
L'opérateur de concaténation 43
Windows PowerShell 43
Les opérateurs arithmétiques 43
Les opérateurs logiques 44
Les opérateurs de comparaison 44
Les opérateurs d'affectation 44
Les opérateurs de type 45
Les opérateurs de redirection 45
Les opérateurs de fractionnement et de jointure 45

CHAPITRE 5
Les fonctions et commandes natives 47
KSH 47
alias 47
cd 48
echo 48
exec 49
exit 49
export 49
jobs 49
kill 49
let 50
print 50
pwd 50
typeset 50
umask 50
unset 50

Perl 51
chomp() 51
chop() 51
chr() 51
defined() 52
hex() 52
join() 52
lc() 52
length() 53
ord() 53
q() 53
qq() 53
qw() 54
sort() 54
split() 55
uc() 55
Python 55
chr() 55
eval() 56
getattr() 56
hasattr() 56
isinstance() 56
issubclass() 57
len() 57
max() 57
min() 57
ord() 58
range() 58
round() 58
sorted() 58
sum() 59
VBS 59
Date() 59
DateAdd() 59
Day() 60
Now() 60
Asc() 60
Clng() 60
Cint() 60
Hex() 61
FormatPercent() 61

Rnd() 61
Lcase() 61
Ucase() 62
Replace() 62
Trim() 62
CreateObject() 62
Windows PowerShell 63
AddDays() 63
Round() 63
ToLower() 63
ToUpper() 64
TrimStart() 64
TrimEnd() 64
Trim() 64
Replace() 64
Split() 65
StartsWith() 65
SubString() 65
IsNullOrEmpty() 65
[char] 66
[object] 66

CHAPITRE 6
Les structures conditionnelles 67
KSH 67
La commande test 68
La structure if..then..else..fi 68
La structure if..then..elif..else..fi 69
La structure case..in..esac 70
Perl 71
La structure if..else 71
La structure if..elsif..else 72
La structure unless 73
La structure given 73
Python 74
La structure if..else 74
La structure if..elif..else 75
Pseudo switch 76
VBS 76
La structure if..then..else..end if 77
La structure if..then..elseif..then..else..end if 77

La structure select..case..end select 78
Windows PowerShell 79
La structure if..else 79
La structure if..elseif..else 80
La structure switch 81

CHAPITRE 7
Les tableaux et dictionnaires 83
KSH 83
Créer un tableau 83
Manipuler un tableau 84
Créer un dictionnaire 84
Manipuler un dictionnaire 85
Perl 85
Créer un tableau 85
Manipuler un tableau 86
Créer un dictionnaire 86
Manipuler un dictionnaire 87
Python 88
Créer un tableau 88
Manipuler un tableau 88
Créer un dictionnaire 89
Manipuler un dictionnaire 89
VBS 90
Créer un tableau 90
Manipuler un tableau 90
Créer un dictionnaire 91
Manipuler un dictionnaire 91
Windows PowerShell 93
Créer un tableau 93
Manipuler un tableau 93
Créer un dictionnaire 95
Manipuler un dictionnaire 96

CHAPITRE 8
Les boucles 99
KSH 99
La boucle for..in..do..done 99
La boucle while..do..done 100
La boucle until..do..done 101
Perl 102

La boucle for . . . 102
La boucle foreach . . . 103
La boucle while . . . 104
La boucle until . . . 104
Python . . . 105
La boucle for . . . 105
La boucle while . . . 105
VBS . . . 106
La boucle for..next . . . 106
La boucle for each..next . . . 107
La boucle do while..loop . . . 107
La boucle do until..loop . . . 108
Windows PowerShell . . . 109
La boucle for . . . 109
La boucle foreach . . . 110
La boucle while . . . 111

CHAPITRE 9
Les fonctions . . . 113
KSH . . . 113
Définir une fonction . . . 113
Une fonction par l'exemple . . . 114
Perl . . . 115
Définir une fonction . . . 115
Une fonction par l'exemple . . . 115
Python . . . 116
Définir une fonction . . . 116
Une fonction par l'exemple . . . 116
VBS . . . 117
Définir une fonction . . . 117
Une fonction par l'exemple . . . 117
Windows PowerShell . . . 119
Définir une fonction . . . 119
Une fonction par l'exemple . . . 120

CHAPITRE 10
Les classes et objets . . . 123
KSH . . . 123
Créer une classe . . . 123
Instancier et utiliser une classe . . . 124

Perl 127
Créer une classe 127
Instancier et utiliser une classe 127
Python 128
Créer une classe 129
Instancier et utiliser une classe 129
VBS 130
Créer une classe 130
Instancier et utiliser une classe 130
Windows PowerShell 133
Créer une classe 133
Instancier et utiliser une classe 134

CHAPITRE 11
Les modules **137**
KSH 137
Créer un module 137
Importer et utiliser un module 138
Perl 139
Créer un module 140
Importer et utiliser un module 140
Python 142
Créer un module 142
Importer et utiliser un module 142
Windows PowerShell 143
Créer un module 144
Importer et utiliser un module 144

CHAPITRE 12
La gestion d'erreurs **147**
KSH 147
La variable $? 147
Manipuler la variable $? 148
Perl 150
La fonction die() 150
La fonction warn() 150
Python 151
L'instruction try..except 151
try..except en action 151
VBS 152

L'instruction On Error 152
On Error en action 153
Windows PowerShell 154
L'instruction try..catch 154
try..catch en action 154

PARTIE 2
Le scripting en pratique 157

CHAPITRE 13
Les expressions régulières 159
KSH 159
La commande sed 159
La commande awk 161
Perl 162
La syntaxe 162
Quelques exemples 163
Python 164
La syntaxe 164
Le module re 165
La fonction search() 165
La fonction sub() 166
VBS 166
La syntaxe 166
L'objet RegExp 167
Windows PowerShell 168
Les opérateurs -match et -notmatch 169
La cmdlet Select-String 170

CHAPITRE 14
La gestion de fichiers 173
KSH 173
Créer un fichier 173
Lire le contenu d'un fichier 174
Modifier le contenu d'un fichier 174
Supprimer un fichier 174
Perl 174
Créer un fichier 174
Lire le contenu d'un fichier 175
Modifier le contenu d'un fichier 175

Supprimer un fichier 176
Python 176
Créer un fichier 176
Lire le contenu d'un fichier 176
Modifier le contenu d'un fichier 177
Supprimer un fichier 177
VBS 177
Créer un fichier 177
Lire le contenu d'un fichier 177
Modifier le contenu d'un fichier 178
Supprimer un fichier 178
Windows PowerShell 178
Créer un fichier 178
Lire le contenu d'un fichier 179
Modifier le contenu d'un fichier 179
Supprimer un fichier 179

CHAPITRE 15
Windows Management Instrumentation 181
KSH 181
L'outil wmic 182
WMI en pratique 182
Perl 183
Le module Win32::OLE 183
WMI en pratique 184
Python 185
Le module wmi 185
WMI en pratique 186
VBS 187
Accéder à WMI 188
WMI en pratique 188
Windows PowerShell 189
La cmdlet Get-WmiObject 189
WMI en pratique 191

CHAPITRE 16
La gestion des services 195
KSH 195
Démarrer un service 195
Arrêter un service 196
Redémarrer un service 196

Perl 196
Lister les services 196
Démarrer un service 198
Arrêter un service 198
Python 199
Lister les services 199
Démarrer un service 200
Arrêter un service 201
VBS 201
Lister les services 201
Démarrer un service 202
Arrêter un service 203
Windows PowerShell 203
Lister les services 203
Démarrer un service 204
Arrêter un service 205

CHAPITRE 17
La gestion des journaux d'événements 207
KSH 207
Les commandes head et tail 208
La commande logger 209
Perl 210
Lister les événements d'un journal 210
Lister les événements correspondant à une période spécifique 212
Python 214
Obtenir le nombre des événements contenus dans un journal 214
Obtenir la taille d'un journal d'événements 214
VBS 215
Sauvegarder un journal d'événements 215
Effacer un journal d'événements 215
Windows PowerShell 216
Lister les journaux d'événements 216
Effacer et supprimer les journaux d'événements 219

CHAPITRE 18
La gestion d'une interface réseau 221
KSH 221
Obtenir une configuration IP 221
Définir une configuration IP 222
Perl 223

Lister les adresses IP d'une machine 223
Collecter des informations de configuration réseau 224
Python 227
Définir une configuration statique 227
Désactiver IPSec 228
VBS 229
Définir un nom de domaine associé à des interfaces réseau 229
Définir l'ordre dans lequel les serveurs DNS sont interrogés 230
Windows PowerShell 230
Obtenir des informations DNS plus détaillées côté client 230
Modifier des informations DNS côté client 232

CHAPITRE 19
Manipuler un fichier XML 233
Le fichier books.xml 233
KSH 236
L'outil xmllint 236
Perl 239
Le module XML::Simple 239
Python 241
Le module xml.etree.ElementTree 241
VBS 243
L'objet Microsoft.XMLDOM 243
Windows PowerShell 244
La classe System.Xml.XmlDocument 244

CHAPITRE 20
La gestion de l'Active Directory 249
KSH 249
Avec l'outil ldapsearch 250
Perl 252
Avec le module Win32::OLE 252
Python 253
Avec le module active_directory 254
VBS 255
Avec ADSI 255
Windows PowerShell 256
Avec le module ActiveDirectory 256

Index 261

Avant-propos

L'expérience nous a appris que, en matière d'administration système et réseau, les différentes opérations peuvent être extrêmement simples ou vraiment compliquées. Et la perception de cette difficulté, très relative, est souvent liée à un autre aspect tout aussi important : celui de la répétition. L'aspect rébarbatif d'une opération particulière doit en effet être pris en compte parce que l'efficience, voire l'efficacité qui en découle, en sera affectée.

Les informaticiens doivent donc considérer la gestion de l'administration système et réseau à la lumière des enjeux soulevés par les contextes environnementaux où ils évoluent, mais aussi des aspirations qui sont les leurs. La question de l'efficacité et de l'efficience est centrale si l'on considère que les architectures réseau actuelles sont d'une complexité toujours plus forte et poussent d'une certaine façon les informaticiens à développer des méthodes dont les fondements sont là aussi toujours plus complexes et sophistiqués.

Une tendance forte est celle du scripting consistant notamment à automatiser des tâches d'administration. Sans lui, les informaticiens seraient bien moins efficaces et beaucoup auraient sans doute quitté le monde de l'informatique pour se consacrer à un autre métier, moins monotone. Si une opération d'administration venait par exemple à être répétée plusieurs fois par jour par une même personne, il ne serait pas nécessaire d'attendre très longtemps avant que l'ennui et la frustration ne se fassent ressentir. Le scripting est, selon moi, la réponse ultime à ce genre de phénomène car il augmente l'efficience, ainsi que l'efficacité et la productivité, au sens noble du terme.

Les objectifs de ce livre

Soyons clairs, ce livre est une apologie du scripting dans son acception la plus pure. Sa proposition est de modifier la perception phénoménologique la plus répandue actuellement, à savoir celle s'émancipant de la dimension d'altérité. **Ce n'est donc pas un ouvrage d'administration** au sens classique du terme. Son objectif premier est de souligner avec force l'importance du scripting pour faire prendre conscience au lecteur de l'efficacité que cela peut lui apporter s'il modifie

ses modes de compréhension. Le livre s'articule autour de cinq grands langages de script que sont Korn Shell, Perl, Python, Visual Basic Scripting et Windows PowerShell.

L'ouvrage s'adresse essentiellement aux administrateurs et ingénieurs système et réseau, mais aussi à certains programmeurs pouvant avoir recours à des langages de script. En réalité, même s'il vise particulièrement des administrateurs et ingénieurs qui souhaitent développer des logiques d'automatisation, tous les informaticiens peuvent le lire, ne serait-ce que pour prendre conscience des enjeux que le scripting peut revêtir et des solutions qu'il apporte.

Pour ceux qui en font déjà, quel que soit leur niveau, ce livre donnera sans doute une nouvelle vision de ce qu'est le scripting. Pour ceux qui débutent en la matière, il sera à tout le moins une entrée dans ce domaine si vaste.

L'axe central et méthodologique de l'ouvrage étant l'exercice de comparaison, son approche pourra au départ dérouter certains lecteurs mais ces derniers finiront, au gré des pages, par saisir sa finalité, à savoir l'utilisation plurielle des langages de script ; l'objectif est de sortir de la dimension exclusive des choix de langages qui influencent les actions des informaticiens en matière d'administration.

Les prérequis

Cet ouvrage est conçu pour être accessible à toutes et tous. En effet, compte tenu du contexte dans lequel il s'inscrit, nous pouvons distinguer deux grandes catégories de lecteurs.

- Les familiers avec le domaine du scripting : cette catégorie de lecteurs (qui est la plus visée) sera normalement plus à l'aise avec les logiques traitées dans l'ouvrage.
- Les moins familiers avec le domaine du scripting : ce type de lecteurs s'initiera alors à un nouveau domaine, apportant une assise claire et plurielle de ce qu'est le scripting.

Quelle que soit la catégorie de lecteurs, débutants ou non, l'aspect touchant à la logique de fonctionnement du code qui sera exposée à travers l'ouvrage sera expliqué et commenté. Cela permettra d'évoluer de manière sereine, puisque chaque construction étudiée le sera à partir d'une analyse de la syntaxe ainsi que d'exemples illustrant cette dernière.

Cependant, il est nécessaire d'avoir une connaissance relativement fine du fonctionnement d'un système d'exploitation, car les langages de script ont cette particularité de fournir une interaction directe avec lui. Il en va de même pour toute application susceptible d'être administrée.

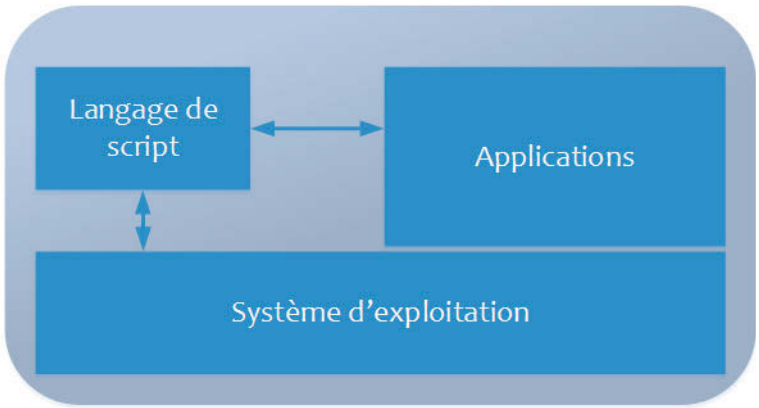
En somme, des notions de programmation de langages de script aideront certainement le lecteur dans la lecture et la compréhension de l'ouvrage, mais même si ce n'est pas le cas, ce livre restera aussi didactique que possible dans sa méthodologie.

Qu'est-ce qu'un langage de script ?

Par langage de script, j'entends un langage de programmation qui se déploie dans l'écosystème d'un interpréteur, ce dernier ayant pour objectif d'analyser et de traduire les instructions vers un langage compréhensible par une machine. Les langages de script sont souvent utilisés dans des logiques d'automatisation, et leurs interactions avec les systèmes d'exploitation sont extrêmement nombreuses.

Autrement dit, les langages de script vont permettre d'enchaîner un certain nombre d'opérations, chacune pouvant être traitée de manière isolée. Ils interagissent aussi bien avec le système d'exploitation qu'avec les applications dont ce dernier constitue le socle. Le schéma suivant illustre le fonctionnement d'un langage de script avec les autres composantes que sont le système d'exploitation et les applications qui lui sont associées.

Figure 0–1
Un langage de script peut devenir un élément central dans la gestion d'un système d'exploitation.



Bien évidemment, il s'agit d'une représentation schématique à but pédagogique. Le fonctionnement d'un langage de script, qu'il soit géré seul ou dans des processus d'interaction, est éminemment complexe et un ouvrage ne suffirait pas à décrire et expliquer toutes ses structures fondamentales.

L'exercice de comparaison comme vision dialogique

Un premier niveau de réflexion consisterait à s'interroger sur un tel exercice. En effet, les informaticiens s'occupant des questions d'administration système et réseau sont, d'une manière générale, pris dans des logiques plutôt partisanes : que ce soit dans le monde Unix/Linux ou le monde Microsoft, les automatismes sont pris beaucoup plus de l'ordre de l'exclusivité que de la pluralité.

À cette situation réelle vient s'ajouter le fait que la pratique du scripting n'est pas symétrique d'un monde à un autre. Dans le monde Unix/Linux, elle est très forte, notamment parce que les systèmes dits Unix se basent sur des architectures favorisant l'automatisation à l'aide de langages de script. Toutefois, cette dimension ne suffit pas à expliquer l'émergence d'une telle

culture. La pratique du scripting est en effet une condition sine qua non à l'accès aux métiers d'administrateurs et ingénieurs.

Le monde Microsoft, au contraire, s'est installé pendant des années dans une dichotomie séparant le programmeur de l'administrateur. Jusqu'à l'apparition de Windows PowerShell, il n'y avait pas d'intention claire de la part de Microsoft de rapprocher ces deux mondes. Tout en conservant leurs particularismes, car un administrateur n'est pas un programmeur, Microsoft s'est aujourd'hui considérablement remis en question. Et en ce qui me concerne, c'est une véritable révolution, voire une rupture épistémologique !

La dissymétrie entre ces deux mondes est donc petit à petit en train de disparaître au profit d'un nouvel équilibre qui s'articulera au service des administrateurs et des programmeurs. Plus que cela, nous allons tendre progressivement vers des modalités d'administration transenvironnementales de plus en plus homogènes ; de là naît donc la nécessité de développer une vision dialogique et non plus exclusive de nos choix d'administration en matière de scripting. Développer une vision dialogique consisterait finalement à constituer un arsenal composé de plusieurs langages, et non plus d'un seul, car malheureusement, les administrateurs d'aujourd'hui, en plus d'être réticents au scripting, sont souvent partisans lorsqu'ils font le choix d'entrer dans ce monde. « Perl vs Python », « VBS vs PowerShell », voici les débats qui animent généralement les forums sur Internet.

La question est de savoir pourquoi nous comparons de cette façon : cherchons-nous à mettre en avant un langage de script plutôt qu'un autre ? Ou alors à nous rassurer sur nos choix hélas souvent partiaux, voire partiels ? Je pense qu'il y a un peu des deux : notre façon d'apprendre les langages de script ne nous a pas enseigné l'essence des choses. Nous ne nous raccrochons qu'à la surface des éléments, sans en comprendre véritablement le sens profond.

Vous aurez sans doute saisi la finalité de cet ouvrage, à savoir la mise en avant de cette vision du monde du scripting à travers toute une série de points de comparaison, principalement les éléments fondamentaux des langages KSH, Perl, Python, VBS et PowerShell. À l'évidence, nous ne devons comparer que ce qui est comparable, au risque de tomber dans une incohérence totale.

Un certain nombre de points de comparaison seront donc étudiés dans ce livre, illustrant le concept central défendu ici, qui est celui d'une vision dialogique. Les langages sont apparemment différents dans la syntaxe, mais se rejoignent sur le fond – les langages de script ont, par exemple, tous des structures conditionnelles, des boucles, et offrent même la possibilité de créer et de manipuler des variables.

Cette base commune, que j'appellerais plutôt socle universel, doit nous permettre d'évoluer vers cette vision dialogique, c'est-à-dire finalement antagoniste, mais complémentaire. L'approche plurielle que revêt cette philosophie aidera alors à déconstruire les idées actuelles malheureusement trop répandues et bridant l'efficacité des informaticiens, pour poser les bases d'une reconstruction idéologique visant au contraire à décupler leur efficacité dans un monde où les besoins en automatisation sont de plus en plus exigeants.

La composition de l'ouvrage

Le livre est divisé en deux grandes parties.

- **Partie 1 – Les éléments de langage**

  Cette partie, qui présente brièvement les différents langages étudiés dans l'ouvrage, met en perspective un certain nombre de points de comparaison : commentaires, variables, opérateurs, fonctions ou commandes natives, instructions conditionnelles, tableaux et dictionnaires, boucles, fonctions, classes et objets, modules ainsi que la gestion d'erreurs. L'ensemble de ces points de comparaison montrera, après leur étude, un caractère universel – une universalité sémantique qui nous émancipe du singulier pour opérer un déplacement vers le pluriel.

- **Partie 2 – Le scripting en pratique**

  L'étude fondamentale des éléments de langage caractérisant la partie précédente nous amènera à analyser quelques cas pratiques : expressions régulières, gestion des fichiers et des répertoires sous Windows, *Windows Management Instrumentation* (WMI), gestion des services, gestion des logs, gestion d'une interface réseau, manipulation de structures XML et gestion de l'Active Directory. Ces cas pratiques ont tous un aspect universel dans leurs modalités d'administration, c'est-à-dire que pour chaque cas, nous pouvons utiliser au moins trois ou quatre langages de script différents, si ce ne sont pas les cinq langages étudiés ici. Néanmoins, cet ouvrage n'étant pas un guide d'administration, nous étudierons des exemples relativement simples à partir desquels le lecteur pourra rebondir à titre personnel vers des cas plus complexes.

Remerciements

J'aimerais remercier les membres de ma famille pour leurs encouragements au quotidien, ainsi que Shakira Alhamwi, Hidetoshi Nakagawa, Jonathan Lowell, Sarah Nielsen, Deha Alhamwi, Magalie Anderson, Mark James, Shinji Komano, John Davis, Katia Denat et Shams Eddine El Andalousi.

J'aimerais aussi remercier les éditions Eyrolles pour leur soutien, en particulier Muriel Shan Sei Fan, Alexandre Habian, Laurène Gibaud, Anne Bougnoux, Sophie Hincelin, Géraldine Noiret, Anne-Lise Banéath et Gaël Thomas.

1

Un peu d'histoire

Tous les langages de script ont une histoire qui leur est propre. Les circonstances dans lesquelles ils sont nés, les conditions particulières qui ont favorisé leur émergence, mais aussi l'intérêt qu'ils ont suscité lors de leur apparition constituent des singularités qu'il est fondamental de souligner. En effet, si de nombreux langages de script ont été développés ces trente dernières années, tous n'ont pas survécu, car ce sont les informaticiens eux-mêmes qui permettent à un langage de résister dans la durée. Un langage de script a des domaines d'application plus ou moins vastes ; plus ces domaines sont nombreux, plus sa capacité à être utilisé sera forte. Et, de ce point de vue, tous les langages de script ne se valent pas, les finalités n'étant évidemment pas toujours les mêmes.

Ce chapitre esquissera de manière brève l'histoire de cinq grands langages de script. D'abord, KSH (Korn Shell), utilisé majoritairement dans les environnements Unix/Linux, puis Perl et Python, que l'on peut employer dans des environnements aussi bien Unix/Linux que Microsoft. Nous apprendrons également à mieux connaître VBS (VBScript), uniquement dans le monde Microsoft, qui est d'une efficacité redoutable. Enfin, nous évoquerons Windows PowerShell, qui est bien souvent – à tort – mal défini par ceux qui l'utilisent et dont la proposition, dans les années qui viennent, sera à suivre car très audacieuse.

KSH

KSH, ou Korn Shell, est un shell Unix développé dans les années 1980 par David Korn, programmeur américain travaillant chez Bell Labs. Korn Shell est une combinaison d'un certain nombre de shells Unix ; il est comparable, dans ses éléments de langage, à BASH *(Bourne Again Shell)*, tout en incorporant les fonctionnalités de c-shell (CSH) et tc-shell (TCSH).

Korn Shell est, avec BASH, le plus utilisé des langages de script pour la programmation shell. En outre, sa syntaxe claire le distingue par exemple fortement de c-shell (CSH) et il attire de ce fait les utilisateurs Unix/Linux grâce sa simplicité et son mode d'accès moins austère.

Écrit en C, Korn Shell a influencé plusieurs langages de script comme Windows PowerShell, et ce, grâce à sa grammaire puissante. D'ailleurs, c'est cette même grammaire qui est la base du standard POSIX.2 *(Portable Operating System Interface)*.

Korn Shell, tirant parti de sa dimension essentielle et pragmatique, a encore de très nombreuses années devant lui, puisqu'il s'est imposé comme une norme universelle.

Perl

Perl est un langage de script apparu en 1987 et développé par Larry Wall. Les paradigmes sur lesquels il se base sont multiples, et Perl a influencé de nombreux langages plus modernes comme Python, JavaScript, Ruby, PHP ou même Windows PowerShell. Sa structure est inspirée des langages pour la programmation shell ainsi que des langages C et C++.

La signification du mot Perl, parmi plusieurs, est *Practical Extraction and Reporting Language*, parce que le but premier du langage était l'extraction d'informations à partir de fichiers de texte afin de créer des rapports. En effet, à l'époque, les modes d'extraction d'informations possibles en matière de scripting n'étaient pas aisés.

Parmi les nombreux domaines d'application où Perl peut être invoqué, nous pouvons citer :

- l'administration système et réseau ;
- la bio-informatique ;
- les applications financières ;
- la programmation web ;
- la programmation réseau ;
- la programmation d'interfaces graphiques.

Le langage Perl existe aujourd'hui sur de nombreuses plates-formes comme Windows, Linux ou même Mac OS X, ce qui fait de lui un langage de script universel. De plus, l'aspect multiparadigme qui le caractérise a attiré des générations entières de programmeurs de tous horizons.

Python

En 1990 apparaît un nouveau langage de script, Python, développé par Guido van Rossum et dont le paradigme de base, parmi d'autres, est l'objet. Le langage Python a été influencé par de nombreux langages de programmation comme le C, C++, Java, Lisp ou même le langage Perl. Inversement, Python a influencé des langages comme Ruby ou JavaScript.

La grande particularité de Python réside dans sa clarté. En effet, un code clair est bien mieux maintenu qu'un code qui ne l'est pas. Ce qui fait de Python un langage privilégié dans le

cadre de projets de petite ou de grande envergure est justement sa pauvreté syntaxique très appréciée par certains programmeurs.

L'objectif ultime de Python est d'afficher un code extrêmement lisible, des structures simples même dans le cas de projets complexes, ainsi qu'une flexibilité redoutable – à l'instar de Perl, c'est un langage extensible et pourvu d'une grande capacité modulaire, offrant toujours plus de créativité aux programmeurs.

Python est utilisé aussi bien dans le monde Windows que dans le monde Unix/Linux. De plus, parmi les nombreux contextes d'utilisation de ce langage, citons :

- le domaine scientifique ;
- l'enseignement ;
- l'administration système et réseau ;
- la programmation de logiciel ;
- la programmation web.

L'architecture de Python offre incontestablement aux informaticiens venant d'univers différents une puissance de productivité et d'efficacité que l'on trouve rarement dans les autres langages.

VBS

VBS, ou *Visual Basic Scripting*, est un langage de script développé par Microsoft qui est apparu en 1996. À cette époque, Microsoft ne disposait pas encore d'un véritable langage de script avec des structures plus ou moins complexes ; il n'était alors possible que d'écrire ce qui est communément appelé des fichiers *batch*, ayant des structures rudimentaires sans possibilité d'évolution en matière d'automatisation.

VBS est issu de la famille Visual Basic, ce qui ajoute souvent de la confusion dans la compréhension des uns et des autres – VBS est souvent confondu avec Visual Basic et Visual Basic Application qui, bien que fortement parents, ont des dissemblances structurelles et des finalités différentes.

Peu de monde sait que VBS est un langage dont le but premier était d'être utilisé dans des environnements web. Il devait à l'origine s'articuler autour de technologies comme ASP *(Active Server Pages)* et IIS *(Internet Information Services)*. Toutefois, Microsoft a décidé avec le temps de revoir sa copie, dans le but d'en faire un langage de script sollicité dans le domaine de l'administration système et réseau. Ce choix a d'ailleurs porté ses fruits, car il a conduit à une meilleure démocratisation de VBS, qui n'avait franchement pas réussi dans le monde du Web, face à un JavaScript plus conquérant.

VBS est bâti autour de COM *(Component Object Model)* qui est, entre autres, un ensemble d'interfaces permettant à des langages d'accéder à des composants système. D'ailleurs, la force de VBS réside en cette technologie ; sans COM, il n'aurait pas rencontré le succès qui fut le sien à partir des années 1999 et 2000.

Aujourd'hui, VBS est un langage de script largement utilisé à travers le monde. Même s'il n'est plus maintenu par Microsoft, il a encore de très nombreuses années devant lui ; mais cette présence, qui ne disparaîtra jamais complètement, sera progressivement moins marquée, car Microsoft souhaite imposer Windows PowerShell, considéré comme une voie d'évolution naturelle.

Windows PowerShell

Windows PowerShell est, à mon sens, la révolution ultime de Microsoft. Son processus d'élaboration a été enclenché à partir du début des années 2000. En effet, c'est précisément en 2001-2002 que l'idée d'une nouvelle approche de la structure du shell (au sens d'interface en ligne de commande) est née. Microsoft avait à l'époque esquissé un shell orienté objet capable de gérer les composants logiciels et matériels sur la base de ce paradigme.

Entre 2002 et 2005, le projet, nommé Monad, en référence au philosophe Leibniz et à sa *Monadologie*, a connu un grand nombre d'orientations, successivement adoptées puis abandonnées. Avant 2006 et l'apparition de Windows PowerShell, des choix particuliers liés au fonctionnement interne de PowerShell, mais aussi aux utilisateurs, ont bouleversé ses phases de développement. Il est évidemment inutile, dans le cadre de cet ouvrage, d'entrer dans les détails de ces phases de développement, mais il faut savoir qu'à l'origine, l'utilisation de PowerShell devait être autre que celle que ses utilisateurs connaissent bien.

PowerShell est donc une technologie d'automatisation dotée d'une interface en ligne de commande associée à un langage de script ; ces deux dimensions sont souvent séparées l'une de l'autre, les uns définissant Windows PowerShell comme une interface en ligne de commande, et les autres comme un langage de script. En réalité, ce n'est ni l'un, ni l'autre, mais tout simplement les deux !

PowerShell répond à une logique d'administration des ressources cohérente et systématique. À l'extraordinaire manque de cohérence qui caractérisait les outils d'administration d'une certaine époque succède aujourd'hui une cohésion très forte qui est devenue la base théorique et pratique des modes d'administration des systèmes Microsoft actuels et à venir.

Windows PowerShell est donc la voie à suivre pour les administrateurs et ingénieurs système et réseau du monde de Microsoft. Autrement dit, la cohérence offerte doit être intégrée dans l'inconscient collectif et une migration doit être opérée pour aller vers plus d'efficacité en matière d'automatisation. Cependant, cette politique d'orientation ne s'inscrit pas dans une rupture totale avec d'autres langages plus ou moins anciens, comme VBScript.

Il faut néanmoins reconnaître que cette nécessité, qui nous est présentée par Microsoft comme une étape qu'il faudra nécessairement franchir, n'est pas communément admise par l'ensemble des professionnels. Certaines personnes, pour diverses raisons, sont réticentes et préfèrent garder leur mode d'administration des ressources inhérent à leur contexte ; Microsoft a donc encore des efforts à faire pour imposer Windows PowerShell de la manière la plus naturelle qui soit.

PARTIE 1

Les éléments de langage

2

Les commentaires

Tous les langages de programmation donnent justement la possibilité d'exploiter ce sens à travers les commentaires. *J'appelle commentaire une partie d'un code source, ni interprétée ni compilée, qui donne du sens à un programme donné, de manière partielle ou non. Les commentaires ont énormément d'importance dans les processus de développement et leur sous-utilisation peut conduire à des échecs collectifs, voire des abandons de projets. Ce chapitre met précisément l'accent sur les commentaires et décrit leur utilisation dans les cinq grands langages étudiés dans cet ouvrage que sont KSH, Perl, Python, VBS et enfin PowerShell.*

KSH

Pour insérer des commentaires dans un code source, il suffit d'utiliser le caractère dièse `#`.

Commenter une ligne

En début de ligne, le signe `#` (1) permet d'ignorer l'ensemble de la ligne.

Commentaire de début de ligne

```
(1) # <code> ou <information donnant du sens au code>
```

Un commentaire peut également être placé en fin de ligne (2), après une instruction qui sera interprétée.

Commentaire de fin de ligne

```
<code source> (2) # <code> ou <information donnant du sens au code>
```

Les deux exemples qui suivent illustrent ces deux types de commentaires :

```
#!/usr/bin/ksh
# Afficher une information
echo "KSH est un superbe langage !!"

>>KSH est un superbe langage !!


#!/usr/bin/ksh
echo "KSH est un superbe langage !!" # Afficher une information

>>KSH est un superbe langage !!
```

NOTE La construction d'un script KSH

Un script KSH doit toujours commencer par la ligne `#!/chemin/vers/l'exécutable KSH`. Cela permettra au script d'être interprété et correctement exécuté. Il s'agit en réalité d'une bonne pratique qui n'est pas toujours respectée. En outre, chaque exemple est à intégrer dans un fichier portant l'extension `.ksh`. Lorsque le fichier est créé, l'utilisateur doit le rendre exécutable à l'aide de la commande `chmod +x 'nom du fichier.ksh'`. Enfin, après être rendu exécutable, le script peut, entre autres, être lancé de cette façon : `'./<nom du fichier>'` à partir du répertoire contenant le script en question.

Commenter un bloc de lignes

KSH ne propose malheureusement pas la possibilité de commenter un bloc de lignes de manière native. L'idée est donc de commenter chaque ligne séparément :

```
#!/usr/bin/ksh
# Première ligne commentée
# Deuxième ligne commentée
echo "KSH est un superbe langage !!"

>>KSH est un superbe langage !!
```

Perl

Pour commenter une ligne, Perl propose, tout comme KSH, le signe `#`. Cependant, il est aussi possible de commenter un bloc de lignes.

Commenter une ligne

L'introduction d'un commentaire en Perl a rigoureusement le même mode de fonctionnement que celui de la famille de la programmation shell : commenter l'ensemble d'une ligne consiste à placer le signe # (1) au tout début de la ligne.

Commentaire de début de ligne

```
(1) # <code> ou <information donnant du sens au code>
```

Perl offre aussi la possibilité de placer un commentaire en fin de ligne (2).

Commentaire de fin de ligne

```
<code source> (2) # <code> ou <information donnant du sens au code>
```

Vu de cette façon, Perl a, de base, les mêmes dispositions que KSH :

```
#!/usr/bin/perl
use strict;
use warnings;

# Afficher une information
print "Perl est un superbe langage !!\n";

>>Perl est un superbe langage !!


#!/usr/bin/perl
use strict;
use warnings;

print "Perl est un superbe langage !!\n"; # Afficher une information

>>Perl est un superbe langage !!
```

IMPORTANT Les pragmas

En Perl, les pragmas sont des instructions qui influencent le comportement de l'interpréteur de manière spécifique. Les pragmas `use strict` et `use warnings` indiquent respectivement à l'interpréteur de bien vérifier la déclaration d'instructions comme les variables, ainsi que de mieux contrôler les messages d'avertissement de Perl lors de son exécution. Autrement dit, ces pragmas obligent à une certaine discipline dans l'écriture du code.

Ces deux exemples mettent parfaitement en exergue les deux possibilités, qui relèvent plus de styles de programmation.

Commenter un bloc de lignes

Perl autorise à commenter des blocs de lignes. Cette possibilité n'est pas native du langage lui-même, mais est cependant très utile : POD, ou *Plain Old Documentation*, est un moyen par lequel des programmes Perl, scripts ou modules sont documentés. Il s'agira, pour un utilisateur écrivant en langage Perl, d'utiliser ce mécanisme de documentation afin de rendre son code plus clair.

Nous n'entrerons évidemment pas dans les détails de la documentation des programmes écrits à l'aide de l'outil POD, mais plutôt dans l'utilisation de base de ce dernier.

Commentaire de bloc de lignes avec les directives =for et =cut

```
=for (1)

Ligne A
Ligne B
Ligne C

=cut (2)
```

Ici, les lignes A, B et C sont délimitées par la directive `=for` (1), indiquant à Perl le début d'une région qu'il devra ignorer, ainsi que la directive `=cut` (2) qui, elle, précise la fin de cette région, à partir de laquelle l'interprétation du code source pourra reprendre. L'exemple suivant consiste en une simple addition, mais le plus intéressant est que Perl a pris en compte les trois lignes de commentaires circonscrites par les directives `=for` et `=cut`.

```
#!/usr/bin/perl
use strict;
use warnings;

my $a = 5;
my $b = 10;
my $result = $a +$b;

=for

Première ligne..
Deuxième ligne..
Troisième ligne..

=cut

print "Le résultat de l'opération est -> $result.\n";

>>Le résultat de l'opération est -> 15.
```

Une autre possibilité consiste à utiliser les directives `=begin` (1), `=end` (2) et `=cut` (3).

Commentaire de bloc de lignes avec les directives =begin, =end et =cut

```
1 =begin

Ligne A
Ligne B
Ligne C

=end 2

=cut 3
```

L'exemple précédent devient alors :

```
#!/usr/bin/perl
use strict;
use warnings;

my $a = 5;
my $b = 10;
my $result = $a +$b;

=begin

Première ligne..
Deuxième ligne..
Troisième ligne..

=end

=cut

print "Le résultat de l'opération est -> $result.\n";

>>Le résultat de l'opération est -> 15.
```

Nous pouvons observer que Perl a interprété le script exactement de la même façon. Utiliser l'une ou l'autre des combinaisons n'est ici pas critique, mais si le nombre de commentaires est abondant ou si la documentation associée est riche, une lecture plus attentive du fonctionnement de POD, sortant du cadre de cet ouvrage, aidera alors le lecteur à mieux documenter ses programmes et à mieux les structurer.

Python

Python propose nativement le commentaire d'une simple ligne et, d'une manière faussement native, le commentaire de bloc.

Commenter une ligne

Commenter une ligne en Python n'échappe pas à l'utilisation du caractère dièse #.

Commentaire de début de ligne

```
(1) # <code> ou <information donnant du sens au code>
```

Placer un commentaire en fin de ligne est évidemment possible.

Commentaire de fin de ligne

```
<code source> (2) # <code> ou <information donnant du sens au code>
```

Les règles qui s'appliquent sont identiques à celles de KSH et Perl :

```
#!/usr/bin/python

# Déclaration de variables (1)
a,b = 0,1

if a < b: # Instruction conditionnelle (2)
   print("This is true..a is less than b.")
else:
   print("This is not true.. a is not less than b.")

>>This is true..a is less than b.
```

Cet exemple intègre les deux perspectives, à savoir le commentaire placé en début de ligne (1) et celui placé en fin de ligne (2).

Commenter un bloc de lignes

Python n'a pas de mécanismes structurels pour commenter un bloc de lignes. Toutefois, sur la base d'un principe que nous n'aborderons pas dans cet ouvrage, le commentaire de plusieurs lignes est une voie possible.

Commentaire de bloc de lignes avec Python

```
(1) '''

Ligne A
Ligne B
Ligne C

''' (2)
```

Il s'agit de délimiter une ou plusieurs ligne(s) par trois apostrophes au début (1) et à la fin (2). Ces lignes ne sont pas tout à fait ignorées par l'interprétateur, mais le but est quand même atteint :

```
#!/usr/bin/python

''' (1)

Commenter un bloc de lignes
est possible avec Python

''' (2)

a,b = 0,1
if a < b:
   print("This is true..a is less than b.")
else:
   print("This is not true.. a is not less than b.")

>>This is true..a is less than b.
```

Mis à part l'aspect purement fonctionnel de la démarche, transparente du point de vue de l'utilisateur, le résultat obtenu est quasi identique.

VBS

Visual Basic Scripting rompt d'une certaine façon avec les autres langages de script en ce qui concerne le commentaire de lignes. Si les autres langages, c'est-à-dire KSH, Perl, Python et même Windows PowerShell que nous étudierons un peu plus tard, utilisent de base le caractère dièse # pour commenter une ligne, VBS propose quant à lui le caractère '.

Commenter une ligne

Comme dans les précédents langages, les commentaires sont déclarés en début ou en fin de ligne.

Commentaire de début de ligne

```
(1) ' <code> ou <information donnant du sens au code>
```

Commentaire de fin de ligne

```
<code source> (2) ' <code> ou <information donnant du sens au code>
```

Dans un même script, les deux possibilités peuvent bien sûr se combiner :

```
' List Operating System and Service Pack Information (1)

strComputer = "."
Set objWMIService = GetObject("winmgmts:" _
 & "{impersonationLevel=impersonate}!\\" & strComputer & "\root\cimv2")

Set colOSes = objWMIService.ExecQuery("Select * from Win32_OperatingSystem")
For Each objOS in colOSes
  Wscript.Echo "Computer Name: " & objOS.CSName
  Wscript.Echo "Caption: " & objOS.Caption 'Name (2)
  Wscript.Echo "Version: " & objOS.Version
  Wscript.Echo "Build Number: " & objOS.BuildNumber
  Wscript.Echo "Build Type: " & objOS.BuildType
  Wscript.Echo "OS Type: " & objOS.OSType
  Wscript.Echo "Other Type Description: " & objOS.OtherTypeDescription
  WScript.Echo "Service Pack: " & objOS.ServicePackMajorVersion & "." & _
   objOS.ServicePackMinorVersion
Next

>>Computer Name: Win8
>>Caption: Microsoft Windows 8
>>Version: 6.2.9200
>>Build Number: 9200
>>Build Type: Multiprocessor Free
>>OS Type: 18
>>Other Type Description:
>>Service Pack: 0.0
```

Nous pouvons observer dans cet exemple un script VBS effectuant une interrogation WMI afin d'obtenir des informations sur le système d'exploitation de l'ordinateur local. Une ligne est commentée au tout début du script (1) pour indiquer à l'utilisateur la démarche voulue. Puis un commentaire de fin de ligne (2) clarifie le nom d'une propriété. En effet, la propriété `Caption` correspond bien au nom du système d'exploitation.

Commenter un bloc de lignes

VBS n'offre pas de mécanismes, même indirects, pour commenter un bloc de lignes. La seule possibilité est de passer par l'utilisation du caractère ' au début de chaque ligne (1) devant être commentée :

```
''''''''''''''''''''''''''''''
' List Operating System  (1)
' and Service Pack Information
''''''''''''''''''''''''''''''

strComputer = "."
Set objWMIService = GetObject("winmgmts:" _
 & "{impersonationLevel=impersonate}!\\" & strComputer & "\root\cimv2")
```

```
Set colOSes = objWMIService.ExecQuery("Select * from Win32_OperatingSystem")
For Each objOS in colOSes
  Wscript.Echo "Computer Name: " & objOS.CSName
  Wscript.Echo "Caption: " & objOS.Caption
  Wscript.Echo "Version: " & objOS.Version
  Wscript.Echo "Build Number: " & objOS.BuildNumber
  Wscript.Echo "Build Type: " & objOS.BuildType
  Wscript.Echo "OS Type: " & objOS.OSType
  Wscript.Echo "Other Type Description: " & objOS.OtherTypeDescription
  WScript.Echo "Service Pack: " & objOS.ServicePackMajorVersion & "." & _
   objOS.ServicePackMinorVersion
Next

>>Computer Name: Win8
>>Caption: Microsoft Windows 8
>>Version: 6.2.9200
>>Build Number: 9200
>>Build Type: Multiprocessor Free
>>OS Type: 18
>>Other Type Description:
>>Service Pack: 0.0
```

Windows PowerShell

Windows PowerShell, dans l'héritage qui est le sien, dispose du caractère # pour ce qui est du commentaire de base, mais fait preuve d'innovation en ce qui concerne le commentaire de bloc de lignes.

Commenter une ligne

Commenter une ligne en PowerShell consiste à placer le signe # en début (1) ou en fin (2) de ligne.

Commentaire de début de ligne

```
(1) # <code> ou <information donnant du sens au code>
```

Commentaire de fin de ligne

```
<code source> (2) # <code> ou <information donnant du sens au code>
```

La syntaxe est donc absolument la même que pour KSH ou pour Perl, pour ne citer que ces langages de script :

```
$a = 2;
$b = 4;
```

```
$result = ($a + $b);
# Afficher le résultat
Write-Host "The result is -> $result."

>>The result is -> 6.
```

Commenter un bloc de lignes

Windows PowerShell dispose de mots-clés pour délimiter une aide qui peut être intégrée aux scripts, mais aussi aux fonctions. Cette aide est justement basée sur l'utilisation des commentaires.

Commentaire de bloc de lignes

```
1 <#
Première ligne
Deuxième ligne
Troisième ligne
2 #>
```

Le mot-clé <# 1 est placé au début ou juste avant la première ligne devant être commentée. Pour indiquer à PowerShell la fin du bloc de lignes commentées, le mot-clé #> 2 doit apparaître juste après la dernière ligne ou à la fin de cette dernière :

```
<#
Cette ligne de commande
permet d'interroger WMI
afin d'avoir plus d'informations
sur le processeur de la machine locale.
#>

Get-WmiObject -Class Win32_Processor -ComputerName 'localhost'

>>Caption           : Intel64 Family 6 Model 58 Stepping 9
>>DeviceID          : CPU0
>>Manufacturer      : GenuineIntel
>>MaxClockSpeed     : 2401
>>Name              : Intel(R) Core(TM) i7-3630QM CPU @ 2.40GHz
>>SocketDesignation : SOCKET 0
```

Ce dispositif est très précieux lorsqu'il s'agit de documenter des scripts ou des fonctions.

3

Les variables

La richesse d'un programme réside principalement dans son aspect purement variable. En effet, la prise en compte d'un certain nombre d'informations par un programme donné a comme principe la variabilité de son état. Par variable, j'entends une valeur particulière et clairement identifiée par un nom, au sein d'un programme précis, et intrinsèquement protéiforme. Les variables, dans leurs différents modes d'existence, sont les éléments les plus utilisés dans l'univers du scripting.
Dans ce chapitre, nous étudierons les variables ainsi que leurs différents modes d'existence à travers les cinq langages de script décrits dans cet ouvrage.

KSH

Déclarer une variable

On crée une variable en KSH grâce à l'opérateur d'affectation `=` (1) (que nous étudierons dans le chapitre sur les opérateurs).

Déclaration de variable

```
(2) <nom_variable> = (1) <valeur_variable> (3)
```

À gauche de l'opérateur d'affectation, le nom (2) de la variable doit être indiqué. Il doit référencer la valeur particulière (3) qu'elle contient. Une fois la variable créée, on accède à sa valeur en préfixant son nom du signe `$` :

```
#!/usr/bin/ksh

#Création de variables en KSH
var1=10
var2=20

echo "La valeur de var1 est $var1 et la valeur de var2 est $var2."

>>La valeur de var1 est 10 et la valeur de var2 est 20.
```

Ici, deux variables ont été créées, `var1` et `var2`. Puis l'accès à leur valeur s'effectue à l'aide de la combinaison `[$ + <nom_variable>]`.

IMPORTANT La création de variables

En KSH, les variables doivent être déclarées d'une manière particulière. En effet, il ne doit pas figurer d'espace autour de l'opérateur d'affectation `=`.

Les variables spéciales

KSH dispose d'un certain nombre de variables dites spéciales, prédéfinies par le langage et très utiles.

- `$0` indique le nom d'une commande.
- `$#` indique le nombre de paramètres positionnels d'une commande.
- `$$` indique le numéro de processus du shell courant.
- `$?` indique un résultat lié au déroulement de la dernière commande exécutée. En principe, si la commande a été exécutée sans erreurs, alors le résultat devrait être `0`. Dans le cas contraire, la variable renverra une valeur autre (comme `1`).
- `$*` contient l'ensemble des paramètres (au moins une valeur) d'une commande. Si cette variable est délimitée par des guillemets, la valeur retournée est unique, mais composite.
- `$@` contient l'ensemble des paramètres d'une commande. Sa signification est donc identique à `$*`. Si elle est délimitée par des guillemets, elle permet de séparer chaque argument en tant qu'élément distinct.
- `$1` à `$9` représentent des paramètres positionnels.

D'autres variables spéciales existent, mais celles que nous venons d'évoquer sont les plus utilisées.

Les variables d'environnement

Un shell se situe par définition au cœur d'un environnement qui n'a pas un caractère statique, c'est-à-dire que ses modes de configuration peuvent changer de manière dynamique. Sur cette base, tout utilisateur a la possibilité de configurer sur mesure son shell.

Pour atteindre ce résultat avec KSH, il faut personnaliser les variables d'environnement. Notez qu'elles doivent d'abord et avant tout être utilisées par le shell lui-même et partagées ensuite par l'ensemble des sous-processus.

- `PATH` représente les différents chemins où se trouvent les exécutables.
- `PS1` affiche la structure du *prompt* principal, ou celui par défaut.
- `HOME` contient le répertoire de base de l'utilisateur.
- `SHELL` indique le shell utilisé par l'utilisateur.
- `LANG` indique la langue utilisée.
- `TEMP` contient le chemin où les processus stockent les fichiers temporaires.
- `OSTYPE` indique le type de système d'exploitation.
- `LD_LIBRARY_PATH` contient les différents chemins où se trouvent les bibliothèques.
- `USER` indique l'utilisateur connecté à la session courante.
- `EDITOR` indique l'éditeur utilisé par défaut au sein de la session courante.
- `MANPATH` contient l'ensemble des répertoires où se situent les fichiers `MAN`.
- `HISTFILE` indique le nom du fichier contenant l'historique des commandes enregistrées.
- `HISTSIZE` est une valeur limitant le nombre de commandes enregistrées dans le fichier historique.
- `PWD` contient le chemin du répertoire actuel.

Ces variables sont très couramment utilisées, mais il y en a évidemment d'autres. Pour les connaître, la documentation officielle KSH est un bon point de départ.

> `NOTE` **Lister les variables d'environnement**
>
> Pour lister les variables d'environnement avec KSH, il suffit de saisir la commande `env` dans la console.

Perl

Perl étant un langage dynamique, les variables n'ont pas besoin d'être déclarées. Plus précisément, si une variable n'existe pas déjà, l'interpréteur lui affectera sa valeur ; mais si elle existe déjà, l'interpréteur utilisera sa valeur existante. Il existe plusieurs types de variables en Perl :

- les variables scalaires ;
- les variables tableaux ;
- les variables dictionnaires ;
- les variables spéciales.

Les variables scalaires

Les variables scalaires sont les plus simples qui existent. Pour en créer une, faites précéder son nom du caractère `$` (1).

Déclaration de variable

```
1 $<nom_variable> = <valeur_variable> 2 ;
```

Le nom d'une variable est composé de lettres et de chiffres. Le caractère _ peut y figurer, mais une variable ne commence pas par un chiffre. La valeur 2 d'une variable scalaire peut revêtir des types différents :

- chaînes de caractères ;
- nombres entiers ;
- valeurs octales ;
- nombres réels ;
- valeurs hexadécimales.

En ce qui concerne la définition de variables scalaires, ces types sont amplement suffisants :

```
$a = 20; # Entier
$b = "Perl"; # Chaîne de caractères
$c = 013; # Octal
$d = 0x7A; # Hexadécimal
$e = 20.7; # Réel

print "a -> $a\n";
print "b -> $b\n";
print "c -> $c\n";
print "d -> $d\n";
print "e -> $e\n";

>>a -> 20
>>b -> Perl
>>c -> 11
>>d -> 122
>>e -> 20.7
```

Les variables tableaux

Une variable scalaire ne peut contenir qu'une seule valeur, d'où une approche simplifiée de sa construction. Pour les valeurs composites, Perl intègre la notion de tableau.

Un tableau, nous le verrons en détail plus tard, est une série de valeurs, de types similaires ou différents. Pour être plus clair, un tableau est une collection de valeurs.

Variable tableau en Perl

```
1 @<nom_variable_tableau> = (<valeur>,<valeur>,<valeur>) 2 ;
```

Une variable tableau commence par le caractère @ (1) et est caractérisée par un ensemble d'une ou plusieurs valeur(s) de types éventuellement différents (2). L'accès à l'une de ces valeurs se fait sur la base d'un indice mis entre crochets [] et compris entre 0 et (taille du tableau-1) :

```
# Variable tableau en Perl
@Tableau = ("Perl","Python","Lisp",17,23);

# Obtenir le premier élément de la variable tableau
print($Tableau[0],"\n");

# Obtenir le quatrième élément de la variable tableau
print($Tableau[3]);

>>Perl
>>17
```

Cet exemple met en évidence la création d'une variable nommée @Tableau qui contient cinq valeurs : trois chaînes de caractères ("Perl", "Python", "Lisp") et deux entiers (17, 23). Puis, la fonction print() est utilisée afin d'afficher le premier et le quatrième élément du tableau ($Tableau[0] et $Tableau[3]). Notez que pour l'obtention d'un élément, la variable commence par le caractère $ et non @, car la valeur extraite est unique.

> **ASTUCE** **Obtenir le dernier élément d'une variable tableau**
>
> L'indice [-1] retourne le dernier élément d'une variable tableau.

Les variables dictionnaires

Les éléments d'une variable tableau sont référencés par un indice numérique commençant à 0. Cependant, une valeur peut aussi être identifiée par une clé ou une chaîne de caractères lui étant associée. Ce procédé, qui est celui des variables dictionnaires, permet alors une association entre une clé donnée et une valeur particulière.

Variable dictionnaire en Perl

```
(1) %<nom_variable> = (<clé> (2) => (3) <valeur> (4), [<clé> => <valeur>]);
```

Le nom d'une variable dictionnaire est précédé du caractère % (1). La construction d'une telle variable consiste en la création de clés (2) auxquelles sont affectées les valeurs (4) correspondantes, au moyen du signe => (3). Enfin, l'accès aux éléments d'une variable dictionnaire se fait par l'intermédiaire des clés qui la constituent et à l'aide du caractère $:

```
# Variable dictionnaire en Perl
%Dict = ('SRV1' => '10.0.2.17', 'SRV2' => '10.0.2.22', 'SRV3' => '10.0.2.25');
```

```
# Premier élément de la variable dictionnaire
print("Premier élément de la variable dictionnaire -> ", $Dict{'SRV1'}, "\n");

# Deuxième élément de la variable dictionnaire
print("Deuxième élément de la variable dictionnaire -> ", $Dict{'SRV2'}, "\n");

# Troisième élément de la variable dictionnaire
print("Troisième élément de la variable dictionnaire -> ", $Dict{'SRV3'}, "\n");

>>Premier élément de la variable dictionnaire -> 10.0.2.17
>>Deuxième élément de la variable dictionnaire -> 10.0.2.22
>>Troisième élément de la variable dictionnaire -> 10.0.2.25
```

Dans cet exemple, une variable dictionnaire `%Dict` est créée. Elle est constituée de trois clés représentant des noms de serveurs. Les valeurs, quant à elles, figurent des adresses IP. La fonction `print()` est de nouveau utilisée pour collecter les différents éléments de la variable dictionnaire à l'aide des signes `{}` indiquant le nom d'une clé.

Lorsqu'une valeur est composite dans son mode de compréhension, ou si une série de valeurs doit être renforcée par une séparation et une identification de ses éléments, le recours aux variables dictionnaires est alors particulièrement utile.

Les variables spéciales

Comme KSH, Perl contient un certain nombre de variables dites spéciales, prédéfinies et extrêmement utiles dans le processus de développement. En voici quelques-unes.

- `$_` représente en quelque sorte la valeur courante correspondant à une entrée standard. Elle est très utilisée dans le cadre de manipulation de collections d'objets ou de fonctions.
- `@_` est une variable tableau contenant l'ensemble des arguments d'une fonction.
- `@ARGV` contient l'ensemble des arguments d'un script.
- `%ENV` est une variable dictionnaire contenant les variables d'environnement du shell.
- `@INC` contient l'ensemble des répertoires où se trouvent les bibliothèques.

Python

La compréhension des variables avec le langage Python s'articule autour de celle du paradigme objet. En effet, en Python, tout est objet. Cela signifie notamment que la valeur d'une variable précise est un objet, ce qui produit un effet de contraste avec des langages comme Perl.

Créer une variable

Python est un langage doté d'une syntaxe volontairement dépouillée. Cette simplicité est appréciée de beaucoup de monde. Créer une variable est donc très aisé et, ici, affectation veut aussi dire création.

Créer une variable en Python

```
1 <nom_variable> = <valeur>
```

Le nom d'une variable 1 ne nécessite aucun caractère particulier et l'instruction n'a pas besoin d'être terminée par un point-virgule `;`. L'exemple suivant illustre la création de deux variables, `a` et `b`, dont les valeurs sont obtenues à l'aide de la fonction `print()` :

```
a = 50
b = 250

print("La valeur de a est -> ", a)
print("La valeur de b est -> ", b)

>>La valeur de a est -> 50
>>La valeur de b est -> 250
```

Les références

En Python, toute valeur d'une variable est un objet. Chaque objet a un identifiant, un type et une valeur associée. Une variable est donc une référence vers sa valeur. Considérons la création d'une simple variable :

```
x = 19
```

Nous avons simplement affecté la valeur `19` à la variable `x`, à l'aide de l'opérateur d'affectation `=`. Sur cette base, créons une autre variable, que nous nommerons `b` et qui sera égale à `x` :

```
b = x
```

Maintenant que les deux variables ont été créées, utilisons la fonction `print()` afin de vérifier qu'elles sont égales :

```
print(x)
print(b)

>>19
>>19
```

Le résultat nous montre bien qu'elles ont la même valeur, mais sont-elles différentes ? A priori oui : `x` étant `x` et `b` étant `b`, elles n'ont en commun que la valeur, résidant dans des espaces mémoire différents. Cette réponse, qui est de l'ordre de l'intuition, est fausse en Python. Pour le prouver, nous allons utiliser la fonction `id()`, qui va nous renvoyer l'identifiant de ces deux variables :

```
print(id(x))
print(id(b))

>>505911248
>>505911248
```

L'identité d'un objet n'étant que l'adresse mémoire où il réside, on peut constater que les variables `x` et `b` pointent vers la même adresse mémoire. Donc, `x` et `b` référencent un même objet, tout en étant indépendantes l'une de l'autre :

```
x = 17

print(x)
print(b)

>>17
>>19
```

Ici, la valeur de `x` a été changée et la fonction `print()` nous a renvoyé des valeurs différentes.

Comme évoqué précédemment, un objet a un type, en plus d'une valeur spécifique et d'un identifiant qui lui sont propres :

```
print(type(x))
print(type(b))

>><class 'int'>
>><class 'int'>
```

Reprenons les deux variables que nous avons créées ; leur type a été déduit en utilisant la fonction `type()`. Les variables `x` et `b` sont donc de type `[int]`, comme le montre le résultat : `<class 'int'>`. `[int]` veut dire un entier et correspond bien à ce que nous avons affecté à ces variables.

Le fonctionnement des variables sous Python est donc différent de celui de VBS ou Windows PowerShell.

VBS

Visual Basic Scripting considère les variables sur une base beaucoup plus homogène que Python, pour ne citer que celui-ci.

Déclarer une variable

En VBS, la gestion des variables est très simple. En effet, une variable est unique et n'a qu'un seul type, `[variant]`. Le type `[variant]` est un conteneur ayant toutes sortes d'autres types comme `[String]`, `[Long]`, `[Integer]` ou `[Boolean]`. En réalité, le programmeur n'a pas à se préoccuper du type des variables qu'il crée, puisque c'est l'interpréteur qui gère la conversion dynamique des données des variables.

Déclarer une variable en VBS

```
1 Dim <nom_variable>
```

D'une manière générale, la déclaration d'une variable se fait à l'aide de l'instruction `Dim` 1, non obligatoire, mais permettant une déclaration de type explicite. Voici la création explicite d'une variable en VBS :

```
Dim Age
```

La création de plusieurs variables à l'aide de l'instruction `Dim` nécessite l'utilisation de virgules :

```
Dim Age, Name, Position
```

L'affectation de valeurs associées nécessite, comme pour les autres langages de script étudiés dans cet ouvrage, l'opérateur d'affectation `=` :

```
Dim Age, Name, Position

Age = 31
Name = "Aurora"
Position = "Striker"

WScript.Echo "The name is " & Name
WScript.Echo "The age is " & Age
WScript.Echo "The position is " & Position

>>The name is Aurora

>>The age is 31

>>The position is Striker
```

Ici, trois variables ont été créées avec leurs valeurs respectives. Puis l'objet `WScript` a été invoqué afin d'utiliser leur contenu.

> **NOTE L'objet WScript**
>
> `WScript` est l'objet racine de la hiérarchie 'Windows Script Host Object Model' et la méthode `Echo` est essentielle pour afficher du texte.

Les variables tableaux

Déclarer une variable tableau en VBS passe aussi par le mot-clé `Dim` (1).

Déclarer une variable tableau en VBS

```
(1) Dim <nom_variable_tableau>(<éléments>) (2)
```

La taille du tableau (unidimensionnel ici) est indiquée entre parenthèses (2) ; plus exactement, il s'agit de l'indice du dernier élément, sachant que le premier élément porte l'indice `0`. C'est pourquoi le nombre figurant entre les parenthèses de l'exemple suivant est `7` alors qu'il s'agit de déclarer une variable tableau à huit éléments :

```
Dim var(7)
```

Cette variable `var` peut donc accepter huit valeurs affectées comme suit :

```
Dim var(7)

var(0) = 1
var(1) = 2
var(2) = 3
var(3) = 4
var(4) = 5
var(5) = 6
var(6) = 7
var(7) = 8
```

L'accès aux différentes valeurs se fait toujours à l'aide d'un indice correspondant à l'élément devant être récupéré :

```
WScript.Echo var(0)
WScript.Echo var(5)

>>1
>>6
```

En VBS, l'utilisation des variables tableaux est très utile pour manipuler des valeurs composites, bien que leur gestion soit vraiment stricte.

Windows PowerShell

PowerShell dispose de plusieurs types de variables, chacun remplissant une fonction bien particulière.

Créer une variable

Comme pour le langage Perl, le symbole dollar `$` (1) sert à désigner une variable en PowerShell.

Déclarer une variable en PowerShell

```
(1) $<nom_variable> = <valeur_variable> ;
```

Et, comme pour l'ensemble des langages étudiés dans cet ouvrage, déclaration et affectation signifient la même chose. Toutefois, Windows PowerShell a une particularité vis-à-vis des autres langages en ce sens que tout type de données peut être encapsulé dans une variable :

```
PS> $num = 5
PS> $string = 'PowerShell'
PS> $services = (Get-Service)
PS> $num.GetType()

>>IsPublic IsSerial Name              BaseType
>>-------- -------- ----              --------
>>True     True     Int32             System.ValueType

PS> $string.GetType()

>>IsPublic IsSerial Name              BaseType
>>-------- -------- ----              --------
>>True     True     String            System.Object

PS> $services.GetType()

>>IsPublic IsSerial Name              BaseType
>>-------- -------- ----              --------
>>True     True     Object[]          System.Array
```

Cet exemple illustre d'abord la création de plusieurs variables : `$num`, `$string` et `$services`. Puis la méthode `GetType()` de ces différentes variables a été instanciée afin de déduire leur type, ce qui nous permet d'examiner le résultat ; la colonne `Name` indique que la variable `$num` est de type `[int32]`, `$string` de type `[String]` et `$services` de type `[Object[]]`. Le typage, dynamique, pré-

sente clairement le caractère protéiforme d'une variable. Windows PowerShell autorise donc une gestion des variables des plus riches, grâce à la puissance de .NET.

Les variables automatiques

Dans le contexte de Windows PowerShell, variable automatique signifie information créée de manière transparente et dynamiquement changeante au cours de l'exécution de l'interpréteur. Ces informations sont donc modifiées automatiquement en mémoire et sont bien sûr exploitables par l'utilisateur à tout moment.

- `$_` contient l'objet d'entrée en provenance du pipeline.
- `$$` contient le dernier jeton de la dernière ligne de commande au cours d'une session Windows PowerShell.
- `$?` contient une valeur de sortie indiquant l'état d'exécution de la dernière ligne de commande : `$True` si l'opération s'est bien passée et `$False` dans le cas contraire.
- `$^` contient le premier jeton de la dernière ligne de commande au cours d'une session Windows PowerShell.
- `$Error` est une variable tableau contenant l'ensemble des erreurs rencontrées au cours d'une session particulière.
- `$Matches` contient les chaînes de correspondance lorsque les opérateurs `-match` et `-notmatch` sont invoqués.
- `$Profile` indique les chemins d'accès des profils Windows PowerShell de l'utilisateur ainsi que de l'application PowerShell.
- `$PSHome` contient le chemin d'accès au répertoire d'installation de Windows PowerShell.
- `$Host` donne accès aux propriétés de l'hôte Windows PowerShell.
- `$True` contient la valeur `true`.
- `$False` contient la valeur `false`.

Les variables d'environnement

Windows PowerShell a accès, tout comme KSH, à un certain nombre de variables d'environnement. Il s'agit de variables spécifiques contenant des informations en lien avec, par exemple, le système d'exploitation, le(s) processeur(s), divers chemins d'accès, des espaces de stockage temporaire ou même le nom de la machine locale. Leur fonction est précise et leur utilisation très contextualisée. PowerShell dispose d'un fournisseur, programme donnant accès à des magasins de données, nommé `env`.

```
PS> (Get-ChildItem env:).Name

ALLUSERSPROFILE
APPDATA
CommonProgramFiles
CommonProgramFiles(x86)
```

```
CommonProgramW6432
COMPUTERNAME
ComSpec
configsetroot
FP_NO_HOST_CHECK
HOMEDRIVE
HOMEPATH
LOCALAPPDATA
LOGONSERVER
NUMBER_OF_PROCESSORS
OS
Path
PATHEXT
PROCESSOR_ARCHITECTURE
PROCESSOR_IDENTIFIER
PROCESSOR_LEVEL
PROCESSOR_REVISION
ProgramData
ProgramFiles
ProgramFiles(x86)
ProgramW6432
PSModulePath
PUBLIC
SAL_ACCESSIBILITY_ENABLED
SESSIONNAME
SystemDrive
SystemRoot
TEMP
TMP
USERDOMAIN
USERDOMAIN_ROAMINGPROFILE
USERNAME
USERPROFILE
VBOX_INSTALL_PATH
Windir
```

En listant l'ensemble des variables d'environnement contenues dans le fournisseur `env`, on s'aperçoit que ce dernier reprend les variables d'environnement au niveau système en y ajoutant `PSModulePath`, ayant le statut de variable d'environnement, mais aussi de variable de préférence.

Les variables d'environnement sont aussi un des moyens de communication entre processus par convention.

Les variables de préférence

Les variables automatiques, associées au fonctionnement de PowerShell, ont des valeurs constantes, ce qui peut paraître paradoxal, je l'avoue. Cependant, l'environnement fourni aux utilisateurs par Windows PowerShell définit un certain nombre de données modifiables par ces derniers à travers les variables de préférence.

- `$ErrorActionPreference` fournit un certain nombre de réponses aux erreurs ne nécessitant pas l'arrêt de l'exécution du pipeline.
- `$WarningPreference` répond aux messages d'avertissement pouvant survenir lors de l'exécution d'une commande.
- `$DebugPreference` répond aux messages de débogage pouvant survenir lors de l'exécution d'une commande quelconque.
- `$ErrorView` structure le format d'affichage des erreurs.
- `$ConfirmPreference` détermine la façon dont Windows PowerShell demande confirmation avant l'exécution d'une commande.
- `$MaximumAliasCount` indique le nombre d'alias autorisés lors d'une session Windows PowerShell.
- `$MaximumErrorCount` indique le nombre d'erreurs enregistrées dans l'historique.
- `$PSEmailServer` spécifie le serveur de messagerie utilisé pour l'envoi des messages électroniques.
- `$VerbosePreference` indique comment PowerShell doit répondre aux messages commentés d'une commande.
- `$OFS` indique le caractère de séparation des éléments d'un tableau lorsque ce dernier est transtypé en chaîne de caractères.

Évidemment, il en existe d'autres, mais celles-ci sont les plus fréquemment rencontrées.

> **NOTE Les variables de préférence**
>
> Les variables de préférence ont une portée globale, mais certaines d'entre elles, à l'aide de paramètres de commandes, sont désactivées au profit de valeurs fournies par ces dernières.

4

Les opérateurs

La manipulation de valeurs est un élément fondamental à prendre en compte avant de faire son choix sur tel ou tel langage de script. En outre, c'est souvent la base d'une orientation caractéristique du code ; cela confère de la puissance aux opérateurs, car d'eux découle une certaine flexibilité. Du calcul arithmétique à la comparaison, de l'affectation à la redirection, les opérateurs constituent un des principes actifs d'un code source. Les maîtriser revient donc à maîtriser le code lui-même, toutes proportions gardées par ailleurs.

Ce chapitre énumère l'ensemble des opérateurs associés aux cinq langages traités depuis le début de l'ouvrage. Qu'il s'agisse de KSH, Perl, Python, VBS ou Windows PowerShell, les opérateurs existants mettent en exergue une très forte richesse que les utilisateurs, quel que soit le monde auquel ils appartiennent, se doivent de connaître dans le but de mieux cerner les possibilités qui s'offrent à eux.

KSH

Korn Shell a inspiré de nombreux langages, notamment en ce qui concerne les opérateurs. En effet, il dispose d'un panel d'opérateurs extrêmement fourni.

Les opérateurs arithmétiques

Tableau 4–1 Opérateurs arithmétiques dans KSH

Opérateur	Description	Exemple
+	Ajoute des valeurs.	`echo $((2+2))`
*	Multiplie des valeurs.	`echo $((2*2))`
-	Soustrait une valeur à une autre.	`echo $((4-2))`
/	Divise des valeurs.	`echo $((4/2))`
%	Renvoie le reste d'une division.	`echo $((10%3))`

Les opérateurs logiques

Tableau 4–2 Opérateurs logiques dans KSH

Opérateur	Description	Exemple
&&	ET logique : le résultat est vrai lorsque les deux expressions le sont.	`[[5 -eq 5 && 6 -eq 7]]`
||	OU logique : le résultat est vrai lorsqu'au moins une des deux expressions est vraie.	`[[5 -eq 5 || 7 -lt 5]]`

Les opérateurs de comparaison

Tableau 4–3 Opérateurs de comparaison dans KSH

Opérateur	Description	Exemple
==	Égal à (concerne les chaînes de caractères).	`"string" == "strings"`
!=	Non égal à (concerne les chaînes de caractères).	`"string" != "strings"`
-eq	Vrai si deux valeurs sont égales (comparaison numérique).	`[[5 -eq 7]]`
-gt	Supérieur à (comparaison numérique).	`[[7 -gt 5]]`
-ge	Supérieur ou égal à (comparaison numérique).	`[[10 -ge 5]]`
-lt	Inférieur à (comparaison numérique).	`[[13 -lt 5]]`
-le	Inférieur ou égal à (comparaison numérique).	`[[23 -le 19]]`
-ne	Différent de (comparaison numérique).	`[[19 -ne 777]]`

Les opérateurs d'affectation

Tableau 4–4 Opérateurs d'affectation dans KSH

Opérateur	Description	Exemple
++	Incrémente une valeur d'une unité.	`print $((var++)) $var`
--	Décrémente une valeur d'une unité.	`print $((var--)) $var`
*=	Multiplie une valeur par une autre.	`print $((var*=3))`

Tableau 4–4 Opérateurs d'affectation dans KSH *(suite)*

Opérateur	Description	Exemple
`+=`	Incrémente une valeur par une autre.	`print $((var+=3))`
`-=`	Décrémente une valeur par une autre.	`print $((var-=3))`
`/=`	Divise une valeur par une autre.	`print $((var/=3))`

Les opérateurs de test sur des fichiers

Tableau 4–5 Opérateurs de test les plus utilisés dans KSH

Opérateur	Description	Exemple
`-x`	Vrai si le fichier est exécutable.	`[[-x /bin/ls]]`
`-r`	Vrai si le fichier est accessible en lecture.	`[[-r /etc/shadow]]`
`-d`	Vrai si le fichier est un répertoire.	`[[-d /etc]]`
`-s`	Vrai si le fichier a une taille non nulle.	`[[-s /tmp/logs.txt]]`
`-e`	Vrai si le fichier existe.	`[[-e /tmp/logs.txt]]`
`-h`	Vrai si le fichier est un lien symbolique.	`[[-h /misc/symlink]]`
`-w`	Vrai si le fichier est accessible en écriture.	`[[-w /etc/passwd]]`
`-k`	Vrai si le *sticky bit* est actif.	`[[-x /misc/inventory]]`

Les opérateurs de redirection

Tableau 4–6 Opérateurs de redirection les plus utilisés dans KSH

Opérateur	Description	Exemple
`>`	Envoie le flux de sortie à un fichier.	`ps > processes.txt`
`>>`	Ajoute du contenu à un fichier déjà existant. Crée le fichier s'il n'existe pas.	`ps -ax >> file.txt`
`<`	Lit l'entrée standard à partir d'un fichier.	`grep word < logs.txt`
`2>`	Redirige le flux d'erreurs vers un fichier.	`ps 2> errors.txt`
`2>>`	Redirige et ajoute le flux d'erreurs vers un fichier existant. Crée le fichier s'il n'existe pas.	`ps 2>> errors.txt`

Perl

Les opérateurs qu'englobe le langage Perl sont très similaires à ceux que l'on peut trouver dans KSH. Cependant, il y a quand même des spécificités, parfois des pièges, en passant d'une utilisation à une autre, ou d'un langage à un autre.

Les opérateurs arithmétiques

Tableau 4–7 Opérateurs arithmétiques dans Perl

Opérateur	Description	Exemple
+	Ajoute des valeurs.	`print 3+2;`
*	Multiplie des valeurs.	`print 3*2;`
-	Soustrait une valeur à une autre.	`print 3-2;`
/	Divise des valeurs.	`print 6/2;`
%	Renvoie le reste d'une division.	`print 10%2;`

Les opérateurs logiques

Tableau 4–8 Opérateurs logiques dans Perl

Opérateur	Description	Exemple
&& ou and	ET logique : le résultat est vrai lorsque les deux expressions le sont.	`if (($x && $y) == 17) {…}`
|| ou or	OU logique : le résultat est vrai lorsqu'au moins une des deux expressions est vraie.	`if (($x || $y) == 17) {…}`

Les opérateurs de comparaison

Tableau 4–9 Opérateurs de comparaison dans Perl

Opérateur	Description	Exemple
== ou eq	Égal à.	`(7 == 7)`
!= ou ne	Non égal à.	`(9 != 7)`
> ou gt	Supérieur à.	`(9 > 7)`
>= ou ge	Supérieur ou égal à.	`(23 >= 5)`
< ou lt	Inférieur à.	`(5 < 23)`
<= ou le	Inférieur ou égal à.	`(87 <= 87)`

Les opérateurs d'affectation

Tableau 4–10 Opérateurs d'affectation dans Perl

Opérateur	Description	Exemple
++	Incrémente une valeur d'une unité.	`$c++ ;`
--	Décrémente une valeur d'une unité.	`$c-- ;`
*=	Multiplie une valeur par une autre.	`print $c *= 3 ;`
+=	Incrémente une valeur par une autre.	`print $c += 3 ;`
-=	Décrémente une valeur par une autre.	`print $c -= 3 ;`
/=	Divise une valeur par une autre.	`print $c /= 3 ;`

Python

Vis-à-vis des opérateurs, Python ne se différencie que très peu des autres langages comme KSH ou Perl. Toutefois, on peut observer la présence d'opérateurs en lien avec les collections.

Les opérateurs arithmétiques

Tableau 4–11 Opérateurs arithmétiques dans Python

Opérateur	Description	Exemple
`+`	Ajoute des valeurs.	`print (3+2)`
`*`	Multiplie des valeurs.	`print (3*2)`
`-`	Soustrait une valeur à une autre.	`print (3-2)`
`/`	Divise des valeurs.	`print (6/2)`
`%`	Renvoie le reste d'une division.	`print (10%3)`

Les opérateurs logiques

Tableau 4–12 Opérateurs logiques dans Python

Opérateur	Description	Exemple
`and`	ET logique : le résultat est vrai lorsque les deux expressions le sont.	`x==n and y==n`
`or`	OU logique : le résultat est vrai lorsqu'au moins une des deux expressions est vraie.	`x==n or y==n`
`not`	NON logique : le résultat est vrai lorsque l'expression est fausse.	`not c==n`

Les opérateurs de comparaison

Tableau 4–13 Opérateurs de comparaison dans Python

Opérateur	Description	Exemple
`==`	Égal à.	`7 == 7`
`!=`	Non égal à.	`9 != 7`
`>`	Supérieur à.	`9 > 7`
`>=`	Supérieur ou égal à.	`23 >= 5`
`<`	Inférieur à.	`5 < 23`
`<=`	Inférieur ou égal à.	`87 <= 87`
`<>`	Différent de.	`17 <> 98`

Les opérateurs d'affectation

Tableau 4–14 Opérateurs d'affectation dans Python

Opérateur	Description	Exemple
*=	Multiplie une valeur par une autre.	c *= 3
+=	Incrémente une valeur par une autre.	c += 3
-=	Décrémente une valeur par une autre.	c -= 3
/=	Divise une valeur par une autre.	c /= 3

Les opérateurs de type

Tableau 4–15 Opérateurs de type dans Python

Opérateur	Description	Exemple
is	Renvoie True si les deux variables pointent vers le même objet.	c is d
isnot	Renvoie True si les deux variables pointent vers des objets différents.	c isnot d

Les opérateurs d'appartenance

Tableau 4–16 Opérateurs d'appartenance dans Python

Opérateur	Description	Exemple
in	Vérifie l'existence d'une valeur au sein d'une séquence d'objets.	list=('blue','yellow','red') if 'red' in list :(..)
not in	Vérifie qu'une valeur n'existe pas au sein d'une séquence d'objets.	list=('blue','yellow','red') if 'green' not in list :(..)

VBS

De tous les langages étudiés dans cet ouvrage, Visual Basic Scripting est celui dont le nombre d'opérateurs est le moins important. Cependant, ceux existants pourraient être qualifiés de fondamentaux.

Les opérateurs arithmétiques

Tableau 4–17 Opérateurs arithmétiques dans VBS

Opérateur	Description	Exemple
+	Ajoute des valeurs.	Wscript.Echo (3+2)
*	Multiplie des valeurs.	Wscript.Echo (3*2)
-	Soustrait une valeur à une autre.	Wscript.Echo (3-2)
/	Divise des valeurs.	Wscript.Echo (6/2)
%	Renvoie le reste d'une division.	Wscript.Echo (10 Mod 3)

Les opérateurs logiques

Tableau 4–18 Opérateurs logiques dans VBS

Opérateur	Description	Exemple
`and`	ET logique : le résultat est vrai lorsque les deux expressions le sont.	`x=n and y=n`
`or`	OU logique : le résultat est vrai lorsqu'au moins une des deux expressions est vraie.	`x=n or y=n`
`not`	NON logique : le résultat est vrai lorsque l'expression est fausse.	`not c=n`

Les opérateurs de comparaison

Tableau 4–19 Opérateurs de comparaison dans VBS

Opérateur	Description	Exemple
`=`	Égal à.	`7 = 7`
`>`	Supérieur à.	`9 > 7`
`>=`	Supérieur ou égal à.	`23 >= 5`
`<`	Inférieur à.	`5 < 23`
`<=`	Inférieur ou égal à.	`87 <= 87`
`<>`	Différent de.	`17 <> 98`

L'opérateur de concaténation

Tableau 4–20 Opérateur de concaténation dans VBS

Opérateur	Description	Exemple
`&`	Concatène des valeurs.	`'Visual Basic ' & ' Scripting'`

Windows PowerShell

La liste des opérateurs qu'englobe Windows PowerShell est longue. Dans cet ouvrage, nous ne vous proposerons que les opérateurs essentiels les plus fréquemment utilisés.

Les opérateurs arithmétiques

Tableau 4–21 Opérateurs arithmétiques dans Windows PowerShell

Opérateur	Description	Exemple
`+`	Ajoute des valeurs.	`3+2`
`*`	Multiplie des valeurs.	`3*2`
`-`	Soustrait une valeur à une autre.	`3-2`
`/`	Divise des valeurs.	`6/2`
`%`	Renvoie le reste d'une division.	`10%3`

Les opérateurs logiques

Tableau 4–22 Opérateurs logiques dans Windows PowerShell

Opérateur	Description	Exemple
`-and`	ET logique : le résultat est vrai lorsque les deux expressions le sont.	`(2 -eq 3) -and (12 -ne 25)`
`-or`	OU logique : le résultat est vrai lorsqu'au moins une des deux expressions est vraie.	`(2 -eq 3) -or (12 -ne 25)`
`-not`	NON logique : le résultat est vrai lorsque l'expression est fausse.	`-not (6 -eq 7)`

Les opérateurs de comparaison

Tableau 4–23 Opérateurs de comparaison dans Windows PowerShell

Opérateur	Description	Exemple
`-eq`	Égal à.	`7 -eq 7`
`-gt`	Supérieur à.	`9 -gt 7`
`-ge`	Supérieur ou égal à.	`23 -ge 5`
`-lt`	Inférieur à.	`5 -lt 23`
`-le`	Inférieur ou égal à.	`87 -le 87`
`-ne`	Différent de.	`17 -ne 98`
`-like`	Établit une correspondance à l'aide du caractère générique `*`.	`'powershell' -like '*shell'`
`-notlike`	Inverse de l'opérateur `-like`.	`'powershell' -notlike '*Basic'`
`-match`	Établit une correspondance à l'aide des expressions régulières.	`'powershell' -match '^\w'`
`-notmatch`	Inverse de l'opérateur `-match`.	`'powershell' -notmatch '^\d'`
`-in`	Vérifie si une valeur se trouve dans une collection.	`'yellow' -in 'red', 'green','blue'`
`-notin`	Inverse de l'opérateur `-in`.	`'yellow' -notin 'red', 'green','blue'`
`-replace`	Remplace une valeur par une autre.	`'MonadShell' -replace 'Monad', 'Power'`

Les opérateurs d'affectation

Tableau 4–24 Opérateurs d'affectation dans Windows PowerShell

Opérateur	Description	Exemple
`++`	Incrémente une valeur d'une unité.	`$c++`
`--`	Décrémente une valeur d'une unité.	`$c--`
`*=`	Multiplie une valeur par une autre.	`$c *= 3`

Tableau 4–24 Opérateurs d'affectation dans Windows PowerShell *(suite)*

Opérateur	Description	Exemple
`+=`	Incrémente une valeur par une autre.	`$c += 3`
`-=`	Décrémente une valeur par une autre.	`$c -= 3`
`/=`	Divise une valeur par une autre.	`$c /= 3`

Les opérateurs de type

Tableau 4–25 Opérateurs de type dans Windows PowerShell

Opérateur	Description	Exemple
`-is`	Vérifie que le type spécifié est bien celui correspondant à l'objet d'entrée.	`5 -is [int]`
`-isnot`	Inverse de l'opérateur `-is`.	`5 -isnot [string]`
`-as`	Convertit l'objet d'entrée vers le type spécifié.	`"05/04/1998" -as [datetime]`

Les opérateurs de redirection

Tableau 4–26 Opérateurs de redirection les plus utilisés dans Windows PowerShell

Opérateur	Description	Exemple
`>`	Envoie le flux de sortie à un fichier.	`Get-Service > services.txt`
`>>`	Ajoute du contenu à un fichier déjà existant. Crée le fichier s'il n'existe pas.	`Get-Service | Sort-Object -Property Status >> svc_status.txt`
`2>`	Redirige le flux d'erreurs vers un fichier.	`Get-Process -Name 'NotHere' 2> errors.txt`
`2>>`	Redirige et ajoute le flux d'erreurs vers un fichier existant. Crée le fichier s'il n'existe pas.	`Get-Process -Name 'NotHere' 2>> errors.txt`

Les opérateurs de fractionnement et de jointure

Tableau 4–27 Opérateurs de fractionnement et de jointure dans Windows PowerShell

Opérateur	Description	Exemple
`-split`	Fractionne des chaînes en sous-chaînes.	`-split 'Windows PowerShell'`
`-join`	Combine des sous-chaînes en une seule.	`-join ('Windows',' PowerShell')`

5

Les fonctions et commandes natives

Plusieurs éléments concourent à juger de la qualité d'un langage de script. Parmi ceux-ci, il y a les perspectives proposées de façon native, autrement dit les possibilités que ce langage offre à l'utilisateur sans que ce dernier sente la nécessité d'ajouter des extensions qu'il aura lui-même développées. Cette dimension native s'articule donc elle-même autour de multiples composants dont font partie ce que l'on nomme les fonctions et commandes natives, programmes effectuant des opérations de base comme la conversion de valeurs, la manipulation de chaînes de caractères, la création de séquences d'objets ou même la gestion de fichiers.

Nous ne nous en rendons généralement pas compte, mais l'utilité de ces fonctions et commandes natives, et ce, quel que soit le langage, conduit à une efficacité redoutable en matière de scripting. À partir de ce constat évident et afin de mieux les connaître, ce chapitre propose d'aborder les plus utilisées.

KSH

Avec KSH, nous aurons surtout affaire aux commandes natives. Elles ont la particularité d'être exécutées directement par le shell lui-même sans créer de processus supplémentaires, ce qui en augmente la rapidité d'exécution. Voici les plus usitées.

alias

La commande `alias` est utilisée pour créer un alias (raccourci vers une commande) et lister ceux déjà existants.

Lister les alias

```
$ alias

>>2d='set -f;_2d'
>>autoload='typeset -fu'
>>command='command '
>>compound='typeset -C'
>>fc=hist
>>float='typeset -lE'
>>functions='typeset -f'
>>hash='alias -t --'
>>history='hist -l'
>>integer='typeset -li'
>>nameref='typeset -n'
>>nohup='nohup '
>>r='hist -s'
>>redirect='command exec'
>>source='command .'
>>stop='kill -s STOP'
>>suspend='kill -s STOP $$'
>>times='{ { time;} 2>&1;}'
>>type='whence -v'
```

Créer un alias

```
$ alias c='clear'
```

cd

La commande `cd` *(Change Directory)* change le répertoire courant pour celui passé en argument.

Changer de répertoire courant

```
$ cd /home/kais
```

echo

La commande `echo` écrit des caractères dans le flux de sortie standard.

Écrire des caractères avec la commande echo

```
$ echo 'Voici du JavaScript !!'
```

exec

La commande `exec` permet entre autres de remplacer le processus actuel par un autre.

Remplacer BASH par KSH

```
$ exec /usr/bin/ksh
```

exit

`exit` sort du shell courant en envoyant un code de sortie au shell parent.

Sortir du shell courant avec un code de retour de 2

```
$ exit 2
```

export

La commande `export` crée des variables d'environnement.

Créer une variable d'environnement

```
$ export PSX='/engine/devel'
```

jobs

La commande `jobs` liste les différents états des *jobs* actuellement exécutés dans le shell courant.

Lister l'état des jobs existants dans l'environnement actuel

```
$ jobs -l
```

kill

Envoie un signal à un processus en cours d'exécution.

Terminer un processus avec le signal SIGTERM

```
$ kill 1325
```

let

Évalue une expression particulière.

Créer une variable à l'aide de la commande let

```
$ let "c = 2 * 17"
```

print

Affiche des chaînes de caractères au niveau de la sortie standard.

Afficher le contenu de la variable $PWD

```
$ print $PWD
```

pwd

pwd affiche le répertoire courant.

Afficher le répertoire courant

```
$ pwd
```

typeset

Définit des attributs de variables.

Définir une variable en lecture seule

```
$ typeset -r x=10
```

umask

Fixe le masque de création de fichiers et de répertoires.

Définition des permissions par défaut des fichiers et des répertoires créés

```
$ umask 0002
```

unset

Détruit l'implémentation des variables et fonctions.

Détruire une variable nommée intvar

```
$ unset intvar
```

Perl

Perl dispose d'un nombre impressionnant de fonctions natives. Cette section ne listera que certaines d'entre elles, car il faudrait un chapitre entier pour toutes les énumérer ! Cependant, dans les chapitres qui vont suivre, nous en mettrons beaucoup d'autres en avant.

chomp()

La fonction `chomp()` supprime d'une chaîne l'expression spécifiant un retour à la ligne.

Supprimer le caractère signifiant un retour à la ligne (correspondant à la touche Entrée)

```
print "User name: ";
$name = <STDIN>;
chomp $name;
print "Your user name is -> $name\n";

>>User name: kais ayari
>>Your user name is -> kais ayari
```

chop()

La fonction `chop()` supprime quant à elle le dernier caractère d'une chaîne, quel qu'il soit.

Supprimer le dernier caractère d'une chaîne

```
$name = "Perld";
chop $name;
print "Your language is -> $name\n";

>>Your language is -> Perl
```

chr()

`chr()` est utile pour convertir des valeurs ASCII ou Unicode en leur caractère équivalent.

Obtenir le mot « KAIS » à partir des valeurs ASCII correspondantes

```
$k = chr(75);
$a = chr(65);
```

```
$i = chr(73);
$s = chr(83);
print $k,$a,$i,$s;

>>KAIS
```

defined()

La fonction `defined()` détermine si une fonction ou une variable est implémentée.

Vérifier qu'une variable est bien définie

```
$a = "defined";
if(defined($a)) {
  print "a is defined.";
} else {
  print "a is not defined";
}

>>a is defined.
```

hex()

La fonction `hex()` convertit une valeur hexadécimale en la valeur décimale qui lui correspond.

Convertir une valeur hexadécimale

```
print hex("0x4E9");

>>1257
```

join()

`join()` concatène des valeurs provenant d'une séquence pour en former une chaîne en ajoutant un séparateur.

Joindre deux chaînes de caractères en utilisant -> comme séparateur

```
print join "->", ("Perl has a very good join "," function.");

>> Perl has a very good join -> function.
```

lc()

`lc()` *(lowercase)* retourne une chaîne de caractères en minuscules.

Convertir une chaîne en minuscules

```
print lc("PERL");

>> perl
```

length()

La fonction `length()` retourne le nombre de caractères d'une chaîne.

Afficher le nombre de caractères d'une chaîne

```
print length("PERL");

>> 4
```

ord()

La fonction `ord()` convertit un caractère en sa valeur ASCII ou Unicode.

Obtenir la valeur ASCII de la lettre K

```
print ord("K");

>> 75
```

q()

La fonction `q()` délimite une chaîne par des guillemets simples (*quotes* en anglais).

Utiliser la fonction q est identique à l'emploi des guillemets simples

```
$with = 'Perl';
$without = q(Perl);

print "$with\n";
print $without;

>> Perl
>> Perl
```

qq()

La fonction `qq()` évalue les expressions à l'intérieur de chaînes, mais garde les guillemets.

Afficher une chaîne de caractères en gardant les guillemets

```
$name = qq("Perl");

print "The language is -> $name";

>>The language is -> "Perl"
```

qw()

La fonction `qw()` divise une expression en plusieurs éléments, avec un espace comme séparateur par défaut.

Créer un tableau de chaînes

```
@elements = qw/Perl is an awesome language/;

foreach my $item(@elements) {
  print "$item\n";
}

>>Perl
>>is
>>an
>>awesome
>>language
```

sort()

La fonction `sort()` est utile pour trier des valeurs sur une base alphabétique ou numérique.

Trier une liste de valeurs numériques

```
@list = (1,3,5,9,6,8,7,4,2);

foreach my $item(sort(@list)) {
  print "$item\n";
}

>>1
>>2
>>3
>>4
>>5
>>6
>>7
>>8
>>9
```

split()

La fonction `split()` est utile pour créer des listes sur la base de simples caractères ou de modèles issus d'expressions régulières en tant que séparateurs.

Créer une liste à partir d'une chaîne avec l'espace comme séparateur

```
@elements = split(" ", "Perl is an awesome language");

foreach my $item(@elements) {
  print "$item\n";
}

>>Perl
>>is
>>an
>>awesome
>>language
```

uc()

`uc()` *(uppercase)* retourne une chaîne de caractères en majuscules.

Convertir une chaîne en majuscules

```
print uc("perl");

>> PERL
```

Python

Tout comme Perl, Python dispose nativement de nombreuses fonctions aux finalités variées. Cette section n'en listera que quelques-unes, mais les chapitres à venir mettront en exergue d'autres fonctions natives extrêmement utiles.

chr()

`chr()` convertit une valeur ASCII en son caractère équivalent.

Obtenir le caractère K à partir de sa valeur ASCII

```
k = chr(75)

print(k)

>>K
```

eval()

La fonction `eval()` est utile pour évaluer une expression.

Évaluer une expression à partir d'une chaîne

```
expression = '2 + 7'

print(eval(expression))

>>9
```

getattr()

La fonction `getattr()` retourne la valeur d'un attribut d'un objet.

Obtenir la valeur de l'attribut x de la variable object

```
print(getattr(object, 'x'))

>>37
```

hasattr()

La fonction `hasattr()` retourne `True` si un objet a l'attribut spécifié.

Savoir si la variable object a bien un attribut nommé x

```
print(hasattr(object, 'x'))

>>True
```

isinstance()

`isinstance()` indique si l'objet passé en argument est bien du type demandé.

Savoir si la variable var est bien du type str

```
var = 'kais'

print(isinstance(var, str))

>>True
```

issubclass()

`issubclass()` indique si une classe B est bien une sous-classe de A.

Savoir si la classe Lion est une sous-classe de Animal

```
print(issubclass(Lion.__class__, Animal))

>>True
```

len()

La fonction `len()` retourne le nombre d'éléments d'une séquence d'objets.

Obtenir le nombre d'éléments d'un tuple

```
x = (1,2,3)

print(len(x))

>>3
```

max()

La fonction `max()` retourne le plus grand élément d'une séquence d'objets.

Obtenir le plus grand élément d'un tuple

```
x = (1,2,3)

print(max(x))

>>3
```

min()

La fonction `min()` retourne le plus petit élément d'une séquence d'objets.

Obtenir le plus petit élément d'un tuple

```
x = (1,2,3)

print(min(x))

>>1
```

ord()

La fonction `ord()` convertit un caractère en sa valeur ASCII.

Obtenir la valeur ASCII de la lettre K

```
print(ord("K"))

>>75
```

range()

La fonction `range()` retourne une liste d'éléments à partir d'une valeur de début de liste et d'une autre valeur (non comprise), qui indique la fin de liste.

Créer une rangée de 10 éléments (0 à 9)

```
r = range(10)
for item in r:
    print(item)

>>0
>>1
>>2
>>3
>>4
>>5
>>6
>>7
>>8
>>9
```

round()

`round()` arrondit des valeurs mathématiques.

Arrondir à deux décimales la valeur 10.869

```
print(round(10.869, 2))

>>10.87
```

sorted()

La fonction `sorted()` trie une liste d'éléments fournie en argument.

Trier une séquence de valeurs

```
print(sorted((1,2,5,6,9,7,3,8,4)))

>>[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

sum()

La fonction `sum()` retourne la somme des éléments passés en arguments.

Calculer la somme des éléments d'une séquence de valeurs

```
print(sum((1,2,5,6,9,7,3,8,4)))

>>45
```

VBS

Les fonctions natives fournies par Visual Basic Scripting sont aussi nombreuses que celles de Perl ou de Python. Tout comme pour ces derniers, nous n'en verrons donc que quelques-unes dans cette section, mais d'autres seront abordées dans les chapitres suivants.

Date()

Cette fonction retourne la date du système.

Obtenir la date du système

```
Wscript.Echo(Date)

>>23/02/2014
```

DateAdd()

La fonction `DateAdd()` ajoute un intervalle de temps, positif ou négatif, à une date spécifique.

Soustraire un mois à la date 27/04/98

```
WScript.Echo(DateAdd("m",-1,"27-avr-98"))

>>27/03/1998
```

Day()

La fonction `Day()` retourne le jour de la date passée en argument.

Obtenir uniquement le jour de la date 27/04/98

```
WScript.Echo(Day("27-avr-98"))

>>27
```

Now()

La fonction `Now()` retourne la date et l'heure système.

Obtenir la date et l'heure système

```
WScript.Echo(Now)

>>23/02/2014 13:15:22
```

Asc()

La fonction `Asc()` retourne la valeur ASCII correspondant au caractère passé en argument.

Obtenir la valeur ASCII de la lettre K

```
WScript.Echo(Asc("K"))

>>75
```

Clng()

La fonction `Clng()` convertit une expression en type `long`.

Convertir une chaîne en type Long

```
WScript.Echo(Clng("100000"))

>>100000
```

Cint()

La fonction `Cint()` convertit une expression en type `int`.

Convertir un nombre à virgule en type int

```
WScript.Echo(Cint(22.569))

>>23
```

Hex()

La fonction `Hex()` retourne la valeur hexadécimale d'un nombre passé en argument.

Obtenir la valeur hexadécimale du nombre 1997

```
WScript.Echo(Hex(1997))

>>7CD
```

FormatPercent()

La fonction `FormatPercent()` convertit une expression en pourcentage.

Convertir l'expression 25/100 en pourcentage

```
WScript.Echo(FormatPercent(25/100))

>>25.00%
```

Rnd()

La fonction `Rnd()` retourne une valeur aléatoire entre 0 et 1.

Obtenir une valeur aléatoire

```
' L'instruction [Randomize] évite que
' la fonction rnd ne renvoie systématiquement la
' même valeur.
Randomize
WScript.Echo(Rnd())

>>0,4354212
```

Lcase()

La fonction `Lcase()` convertit une chaîne de caractères en minuscules.

Convertir une chaîne en minuscules

```
WScript.Echo(LCase("VISUAL BASIC SCRIPTING"))

>>visual basic scripting
```

Ucase()

La fonction `Ucase()` convertit une chaîne de caractères en majuscules.

Convertir une chaîne en majuscules

```
WScript.Echo(UCase("visual basic scripting"))

>>VISUAL BASIC SCRIPTING
```

Replace()

La fonction `Replace()` trouve et remplace des occurrences d'une chaîne.

Trouver le mot Program et le remplacer par le mot Scripting

```
text = "Visual Basic Program"
WScript.Echo(Replace(text, "Program", "Scripting"))

>>Visual Basic Scripting
```

Trim()

La fonction `Trim()` élimine les espaces des deux côtés d'une chaîne.

Supprimer les espaces d'une chaîne

```
text = " Visual Basic Scripting     "
WScript.Echo(Trim(text))

>>Visual Basic Scripting
```

CreateObject()

La fonction `CreateObject()` crée un objet du type passé en argument.

Créer une référence à l'objet FileSystem

```
Set objFSO = CreateObject("Scripting.FileSystemObject")
```

Windows PowerShell

PowerShell ne dispose pas de fonctions natives. Étant donné qu'il est entièrement basé sur la technologie .NET, il nécessite de penser ses actions différemment d'avec les autres langages de script. Le concept approprié est celui de *cmdlet*. Une cmdlet, ou applet de commande, est un objet de programmation accomplissant une ou plusieurs tâches bien spécifiques. Cependant, afin de nous rapprocher au maximum de l'aspect natif du fonctionnement de PowerShell, nous présentons dans cette section quelques méthodes et classes .NET (au cœur du concept de cmdlet) permettant d'accomplir des opérations de base rencontrées dans les sections précédentes.

AddDays()

`AddDays()` ajoute un intervalle de temps à une date donnée.

Ajouter 30 jours à la date actuelle

```
PS> $date = [System.DateTime]::Now
$date.AddDays(30)

>>Vendredi 28 mars 2014 15:15:53
```

Round()

`Round()` arrondit un nombre passé en argument.

Arrondir un nombre à deux décimales

```
PS> [System.Math]::Round(67.7589, 2)

>>67,76
```

ToLower()

`ToLower()` convertit une chaîne en minuscules.

Convertir la chaîne POWERSHELL en minuscules

```
PS> "POWERSHELL".ToLower()

>>powershell
```

ToUpper()

`ToUpper()` convertit une chaîne en majuscules.

Convertir la chaîne powershell en majuscules

```
PS> "powershell".ToUpper()

>>POWERSHELL
```

TrimStart()

La méthode `TrimStart()` supprime les espaces au début d'une chaîne.

Supprimer les espaces de début de chaîne

```
PS> "   powershell".TrimStart()

>>powershell
```

TrimEnd()

La méthode `TrimEnd()` supprime les espaces à la fin d'une chaîne.

Supprimer les espaces de fin de chaîne

```
PS> "   powershell    ".TrimEnd()

>>   powershell
```

Trim()

La méthode `Trim()` supprime les espaces au début et à la fin d'une chaîne.

Supprimer les espaces de début et de fin de chaîne

```
PS> "   powershell    ".Trim()

>>powershell
```

Replace()

La méthode `Replace()` remplace une occurrence d'une chaîne par une autre.

Remplacer le mot monad par le mot power

```
PS> "monadshell".Replace('monad','power')

>>powershell
```

Split()

La méthode `Split()` divise une chaîne en plusieurs parties.

Diviser la chaîne Windows PowerShell en deux mots

```
PS> "Windows PowerShell".Split(' ')

>>Windows
>>PowerShell
```

StartsWith()

La méthode `StartsWith()` détermine si le début d'une chaîne correspond bien à la chaîne spécifiée en paramètre.

Vérifier si la chaîne mon est bien au début de la chaîne monadshell

```
PS> "monadshell".StartsWith('mon')

>>True
```

SubString()

La méthode `SubString()` obtient une sous-chaîne à partir d'une chaîne spécifiée.

Obtenir le mot monad à partir du mot monadshell

```
PS> "monadshell".SubString(0,5)

>>monad
```

IsNullOrEmpty()

La méthode `IsNullOrEmpty()` est utile pour vérifier si une variable est nulle ou vide.

Savoir si une variable $c est nulle ou vide

```
PS> [string]::IsNullOrEmpty($c)

>>False
```

[char]

La classe [char] convertit une valeur ASCII en son caractère équivalent.

Obtenir la lettre K à partir de sa valeur ASCII

```
PS> [char]75

>>K
```

[object]

La classe [object] indique si une valeur est un objet (ce qui, dans le contexte de Windows PowerShell, sera toujours le cas, mais une variable ne référence pas toujours un objet).

Savoir si une variable $c contient bien un objet

```
PS> $c = 4
PS> $c -is [object]

>>True
```

6

Les structures conditionnelles

Les structures conditionnelles sont l'un des piliers de tout algorithme car elles orientent le code selon le résultat d'un test. Si une condition est remplie, alors telle partie du code sera exécutée, sinon c'est une autre partie du code qui s'appliquera.
L'utilisation des structures conditionnelles est extrêmement courante dans le monde de la programmation en général et du scripting en particulier. Les connaître pour mieux les maîtriser est donc une condition sine qua non d'un code bien écrit. La pluralité des structures conditionnelles qu'offrent les cinq langages étudiés dans cet ouvrage aidera le lecteur à mieux comprendre, mais aussi à mieux évaluer les richesses offertes afin de s'orienter plus aisément vers une utilisation spécifique de ces technologies.

KSH

Korn Shell dispose principalement de quatre structures conditionnelles :

- `test` ;
- `if..then..else..fi` ;
- `if..then..elif..else..fi` ;
- `case..in..esac`.

La commande test

La commande `test` est très connue des utilisateurs de KSH ou de BASH. Son but, comme son nom l'indique, est de tester une expression : si cette dernière est vraie, la commande `test` retourne la valeur `0`, sinon elle retourne la valeur `1`.

Syntaxe de la commande test

```
test <expression> (1)
```

Ici, l'expression (1) évaluée par la commande `test` représente toute expression valide. Le tableau suivant esquisse quelques opérateurs susceptibles d'être utilisés par la commande `test`.

Tableau 6–1 Opérateurs de base utilisés avec la commande test

Opérateur	Description	Exemple
`-d`	Vérifie si le fichier est un répertoire.	`test -d /etc`
`-e`	Vérifie si le fichier existe.	`test -e /etc/password`
`-x`	Vérifie si le fichier est un exécutable.	`test -x /bin/ls`
`-w`	Vérifie que le fichier est accessible en écriture.	`test -w /etc/shadow`
`-s`	Vérifie que le fichier a une taille supérieure à zéro.	`test -s /misc/script.sh`

> **NOTE Au sujet de la commande test**
>
> Pour plus d'informations, tapez `man test` dans la console.

La structure if..then..else..fi

L'instruction `if..then..else..fi` est utilisée pour orienter le code en fonction de la réalisation d'une ou plusieurs conditions.

Syntaxe de la structure if..then..else..fi

```
if (1) [[ <expression conditionnelle> ]] then (2)
   <liste d'instructions> (3)
[else (4)
   <liste d'instructions> (5)]
fi (6)
```

Dans cette structure, KSH évalue d'abord une expression conditionnelle (1) délimitée par les éléments `[[ ]]`. Si le résultat est `true`, alors (2) une liste d'instructions (3) est exécutée et KSH quitte la structure. Si le résultat de l'évaluation est `false`, alors (4) KSH exécute une autre liste d'instructions (5) et quitte la structure. Enfin, le mot-clé `fi` (6) marque la fin de la structure `if..then..else..fi`.

L'exemple suivant met en évidence la vérification de la supériorité (en termes de valeur numérique) d'une variable sur l'autre.

```
a=4
b=23

if [[ $a -gt $b ]] then
   echo "a is greater than b."
else
   echo "b is greater than a."
fi

>>b is greater than a.
```

Cet autre exemple compare deux chaînes :

```
a="KSH"
b="BASH"

if [[ $a != $b ]] then
   echo "a and b are different."
else
   echo "a and b are not different."
Fi

>>a and b are different.
```

La structure if..then..elif..else..fi

Lorsque plusieurs conditions doivent être évaluées de manière exclusive, l'instruction `if..then..else..fi` n'est alors plus considérée comme adéquate ; cette dernière est en effet idéale lorsqu'il s'agit d'évaluer une seule condition. KSH propose une autre construction pour évaluer plusieurs conditions : la structure `if..then..elif..else..fi`.

Syntaxe de la structure if..then..elif..else..fi

```
if ❶ [[ <expression conditionnelle> ]] then ❷
   <liste d'instructions> ❸
[elif ❹ [[ <expression conditionnelle> ]] then ❺
   <liste d'instructions> ❻]
[else ❼
   <liste d'instructions> ❽]
fi
```

KSH évalue d'abord la première expression ❶. Si celle-ci est vraie, alors ❷ KSH exécute la liste d'instructions correspondantes ❸ et quitte la structure. Dans le cas contraire, KSH évalue la condition suivante ❹ et exécute alors ❺ la liste d'instructions correspondante ❻ si

le résultat de l'évaluation est `true`. Après avoir vérifié l'ensemble des conditions, et si toutes retournent `false`, alors (7) KSH exécute le dernier bloc d'instructions (8).

```
a=5
b=7

if [[ $a -eq $b ]] then
    echo "a and b are equal."
elif [[ $a -ne $b ]] then
    echo "a and b are not equal."
else
    echo "nothing else."
fi

>>a and b are not equal.
```

Dans cet exemple, deux variables sont comparées dans une instruction `if..then..elif..else..fi`. KSH vérifie si les variables `$a` et `$b` sont égales. Si les conditions retournent toutes deux `false`, KSH exécutera alors un bloc d'instructions par défaut.

La structure case..in..esac

Nous avons pu remarquer que la structure `if..then..elif..else..fi` n'est autre qu'une extension de la structure `if..then..else..fi`, la première permettant de concentrer une série de conditions dans une seule structure. Cependant, lorsque le nombre de conditions est élevé, le code peut devenir illisible et difficile à maintenir. Pour pallier ce problème, KSH dispose d'une structure très intéressante qui prend en charge cette complexité : `case..in..esac`.

Syntaxe de la structure case..in..esac

```
case (1) <valeur> (2) in
[PATTERN [| PATTERN]) (3) <liste d'instructions> (4) ;; (5)]
esac (6)
```

Une valeur (2) est passée à la structure `case..in..esac` (1), qui la compare à une série de valeurs de référence (3). Lorsqu'il y a correspondance, KSH exécute alors une liste d'instructions (4), puis l'opérateur `;;` (5) force KSH à quitter la structure. Le mot-clé `esac` (6) marque la fin de l'instruction.

La structure suivante effectue trois évaluations sur une variable :

```
a="KSH"

case $a in
   "BASH") echo "The language is BASH.";;
   "KSH") echo "The language is KSH.";;
```

```
   "PERL") echo "The language is PERL.";;
   * ) echo "It's another language.";;
esac

>>The language is KSH.
```

La variable est ici une chaîne passée à la structure `case..in..esac` qui effectue de simples tests de correspondance à l'aide de valeurs de référence. Le résultat renvoyé par la structure conditionnelle indique que la deuxième valeur de référence, `KSH`, correspond bien à la valeur de la variable `$a`. Toutefois, si aucune correspondance n'avait pu être effectuée, KSH aurait exécuté l'instruction correspondant à la valeur `*`, cette dernière représentant une orientation par défaut.

La structure `case..in..esac` nous montre bien l'efficacité produite lorsqu'il s'agit de sérier de multiples conditions.

Perl

Perl dispose quant à lui d'un nombre de structures conditionnelles supérieur à ce que proposent des langages comme KSH. Dans cette section, nous verrons les principales structures conditionnelles proposées par Perl :

- `if..else` ;
- `if..elsif..else` ;
- `unless` ;
- `given`.

La structure if..else

L'instruction `if..else` est une structure essentielle de la gestion des conditions.

Syntaxe de l'instruction if..else

```
if (1) (<expression conditionnelle>)
{
    <liste d'instructions> (2)
}
[else (3)
{
    <liste d'instructions> (4)
}]
```

Ici, une expression conditionnelle (1) entre parenthèses est évaluée. Si le résultat est `true`, une liste d'instructions est exécutée (2) et Perl quitte l'instruction. Le bloc `else` (3) oriente le code d'une autre façon (4) si le résultat de l'expression est `false`.

La structure `if..else` suivante définit la valeur d'une variable en fonction de la comparaison de deux autres variables :

```
$continue;
$a = 5;
$b = 10;

if($a > $b){
    $continue = 'false';
} else {
    $continue = 'true';
}
```

La structure if..elsif..else

Tout comme Korn Shell, Perl a une structure conditionnelle permettant de multiplier les conditions au sein d'une même structure.

Syntaxe de l'instruction if..elsif..else

```
if (1) (<expression conditionnelle>)
{
    <liste d'instructions_1> (2)
}
[elsif (3) (<expression conditionnelle>)
{
    <liste d'instructions_2> (4)
}]
[else (5)
{
    <liste d'instructions_3> (6)
}]
```

La structure `if..elsif..else` est très utile lorsqu'il s'agit d'évaluer quelques conditions successives. Perl évalue tout d'abord une expression (1), puis exécute une liste d'instructions (2) si celle-ci est vraie. Si cette dernière est fausse (3), la condition suivante est évaluée et Perl agit exactement de la même manière (4) que pour la première condition. Enfin, si aucune des conditions évaluées n'est vraie, alors (5) un bloc d'instructions (6) sera exécuté par défaut.

```
$a = 5;
$b = 10;

if($a == $b){
    print("a and b are equal.");
} elsif($a > $b) {
    print("a is greater than b.");
} else {
    print("nothing else.");
```

```
}

>>nothing else.
```

Cet exemple met en évidence deux conditions évaluées l'une après l'autre. La première teste l'égalité de deux variables et la seconde si l'une est supérieure à l'autre. Si aucune des conditions testées n'est vraie, alors le bloc d'instructions `else` est exécuté. C'est d'ailleurs ce que nous montre le résultat : la variable `$a` n'est ni égale ni supérieure à la variable `$b`.

La structure unless

La structure `unless` est une instruction vraiment originale. En effet, un bloc d'instructions (2) est exécuté uniquement si la condition évaluée (1) est fausse et non pas vraie comme c'est le cas dans les autres structures conditionnelles.

Syntaxe de l'instruction unless

```
unless (1) (<expression conditionnelle>)
{
    <liste d'instructions> (2)
}
```

La structure `unless` suivante vérifie qu'une variable ne se situe pas dans un certain intervalle :

```
$x = 19;
unless ($x > 35 and $x < 200) {
    print("Number is out of the range.")
}

>>Number is out of the range.
```

La structure given

`given` est une instruction émulant le comportement d'une structure `switch…` à la manière de la structure `case..in..esac` vue précédemment avec le langage KSH. Cependant, il s'agit pour le moment d'un processus expérimental.

Syntaxe de l'instruction given

```
given (1) (<valeur>)
{
    [when(<valeur>) (2) {<liste d'instructions>}] (3)
    [default (4) {<liste d'instructions>}] (5)
}
```

Le principe est simple : une valeur (1) est passée en argument à l'instruction `given`, qui l'évalue une ou plusieurs fois. Le mot-clé `when` (2) introduit une évaluation plus ou moins complexe de la valeur passée en argument et un bloc d'instructions (3) est exécuté si le résultat de cette évaluation est vrai. Le nombre de blocs `when` n'a pas de limite précise. Si aucune des évaluations n'est vraie, Perl exécute alors le bloc d'instructions (5) issu de la clause `default` (4).

L'instruction suivante effectue plusieurs tests sur une variable donnée :

```
$name = "Perl";

given($name) {
    when(undef) {say '$name is undefined'}
    when(/^\d/) {say '$name begins with a digit.'}
    default {say 'nothing else.'}
}

>>nothing else.
```

Le premier test, à l'aide de la fonction `undef`, précise le caractère défini ou non de la variable `$name`. Puis, si la variable `$name` est définie, une autre évaluation a lieu, consistant simplement à vérifier que la variable commence bien par un chiffre. Le contenu de la variable `$name` étant une chaîne, elle est donc bien définie et ne commence pas par un chiffre ; Perl exécute logiquement le bloc d'instructions lié à la clause `default`.

Python

En Python, la syntaxe des structures conditionnelles (comme les autres instructions) est particulièrement simple et claire, ce qui lui confère un excellent avantage par rapport aux autres langages. Nous présentons trois de ces structures :

- `if..else` ;
- `if..elif..else` ;
- le pseudo `switch`.

La structure if..else

Syntaxe de l'instruction if..else

```
if (1) <expression conditionnelle>:
    <liste d'instructions> (2)
[else: (3)
    <liste d'instructions> (4)]
```

Dans cette structure, et comme pour les autres langages, une expression (1) est évaluée par Python. Si le résultat est vrai, alors le bloc d'instructions correspondant (2) est exécuté, mais si

le résultat est faux ❸, c'est l'autre bloc ❹ qui s'applique ; Python quitte ensuite l'instruction conditionnelle.

```
a,b = 0,1
if a < b:
    print("a is less than b.")
else:
    print("a is greater than b.")

>>a is less than b.
```

Cet exemple met en exergue la comparaison de deux variables, `a` et `b`. Ces dernières sont définies en une seule fois et l'expression conditionnelle n'est pas entre parenthèses ; elle doit cependant se terminer par `:`.

> **IMPORTANT** **Syntaxe des structures du langage Python**
>
> La syntaxe de Python est assez naturelle, mais une attention très importante doit être portée sur l'indentation du code, qui est porteuse de sens et non pas seulement esthétique.

La structure if..elif..else

Dans la continuité de la section précédente, la structure `if..elif..else` étend les capacités de `if..else` en multipliant les expressions conditionnelles.

Syntaxe de l'instruction if..elif..else

```
if ❶ <expression conditionnelle>:
    <liste d'instructions 1> ❷
[elif ❸ <expression conditionnelle>:
    <liste d'instructions_2> ❹]
[else ❺
    <liste d'instructions_3> ❻]
```

La clause `if` ❶ évalue une première condition et exécute au besoin le bloc d'instructions ❷ qui lui correspond. La clause `elif` ❸ (contraction de *else if*) est une alternative ❹ en cas de résultat négatif de la première condition. Python exécutera le bloc `else` ❺ ❻ uniquement si toutes les conditions évaluées sont fausses.

Reprenons l'exemple précédent :

```
a,b = 0,1

if a == b:
    print("a and b are equal.")
elif a < b:
    print("a is less than b.")
```

```
else:
    print("nothing else.")

>>a is less than b.
```

La structure conditionnelle `if..else` a été simplement étendue à l'aide du bloc `elif`. Le résultat attendu est évidemment le même, mais l'introduction d'une autre évaluation a été possible grâce à ce bloc, ce qui nous a évité de créer deux structures conditionnelles `if..else`.

Pseudo switch

Tout comme avec Perl, Python n'a pas de structure `switch`. Si l'utilisateur veut avoir recours à une construction similaire, il devra user d'un peu d'ingéniosité. Voici par exemple une structure pseudo `switch` :

```
Languages = dict(
    one = 'Python',
    two = 'Perl',
    three = 'KSH',
    four = 'VBS',
    five = 'PowerShell'
)
Language = 'four'
print(Languages.get(Language, 'default -> other language'))

>>VBS
```

Cette structure consiste à définir une variable nommée `Languages`. Au sein de cette variable est déclaré un dictionnaire à l'aide de la fonction `dict()`. Puis cinq paires clés/valeurs sont créées où chaque valeur est une chaîne. Une autre variable, nommée `Language`, est ensuite définie ; elle sert de référence dans la structure pseudo `switch`. Enfin, la méthode `get()` de l'objet `Languages` est instanciée et deux arguments lui sont passés : le premier est l'objet de référence, `Language` et le second est la clause `default`, sous forme de chaîne. Si l'objet de référence se trouve bien dans la structure pseudo `switch`, la valeur correspondante, ici `VBS`, est alors retournée. Dans le cas contraire, le second argument est retourné par Python.

VBS

Visual Basic Scripting est le langage de script dont les structures sont sans conteste les plus complexes, par opposition à d'autres langages comme Python. Pour preuve :

- `if..then..else..end if` ;
- `if..then..elseif..then..else..end if` ;
- `select..case..end select`.

La structure if..then..else..end if

Comme les autres langages de script étudiés dans cet ouvrage, VBS dispose d'une structure exécutant une liste d'instructions si une expression conditionnelle est vraie et une autre liste si l'expression est fausse.

Syntaxe de la structure if..then..else..end if

```
if (1) <expression conditionnelle> then (2)
   <liste d'instructions>
[else (3)
   <liste d'instructions>]
end if (4)
```

Cette structure conditionnelle ressemble beaucoup à son équivalent en Korn Shell. Une clause `if` (1) marque le début de la structure conditionnelle. Si la condition est vraie, un bloc d'instructions est alors (2) exécuté. Si la condition est fausse, (3) un autre bloc est appliqué. La fin d'une structure conditionnelle en VBS est marquée par l'existence de la clause `end if` (4).

Voici un exemple simple de structure `if..then..else..end if` testant l'égalité de deux variables :

```
var1 = 10
var2 = 20

If var1 = var2 Then
   WScript.Echo "var1 and var2 are equal."
Else
   WScript.Echo "var1 and var2 are not equal."
End If

>>var1 and var2 are not equal.
```

Notez que l'expression conditionnelle `var1=var2` peut être délimitée par des parenthèses.

La structure if..then..elseif..then..else..end if

Si plusieurs conditions doivent être testées de manière exclusive, VBS propose une extension de la structure précédente.

Syntaxe de la structure if..then..elseif..then..else..end if

```
if <expression conditionnelle> then
   <liste d'instructions>
[elseif (1) <expression conditionnelle> then
   <liste d'instructions>]
```

```
[else
   <liste d'instructions>]
end if
```

Reprenons l'exemple précédent :

```
var1 = 10
var2 = 20

If (var1 = var2) Then
   WScript.Echo "var1 and var2 are equal."
ElseIf (var1 < var2) Then
   WScript.Echo "var1 is less than var2."
Else
   WScript.Echo "nothing else."
End If

>>var1 is less than var2.
```

Nous pouvons observer qu'un bloc `elseif` (1) a été intégré. Cela nous a permis d'insérer une autre condition, pour savoir si `var1` est strictement inférieure à `var2`.

La structure select..case..end select

Lorsque plusieurs conditions doivent être réalisées sur une seule valeur, la structure `select..case..end select` peut être d'un excellent recours.

Syntaxe de la structure select..case..end select

```
select case <valeur> (1)
    [case (2) <valeur>
        <liste d'instructions> (3)]
    [case else (4)
        <liste d'instructions> (5)]
end select (6)
```

Une valeur (1) est passée en argument à la structure conditionnelle, puis plusieurs conditions sont testées (2). Si plusieurs conditions sont vraies, VBS exécute alors la liste d'instructions (3) correspondant à la première condition évaluée comme vraie. À l'inverse, si toutes les conditions sont fausses, VBS exécute, s'il existe, le bloc optionnel d'instructions (5) correspondant à la clause `case else` (4). L'ensemble des conditions étant vérifié, VBS continue l'exécution de l'algorithme après la clause `end select` (6).

Créons une variable, que nous nommerons `var` :

```
var = "Visual"
```

À présent, utilisons une structure conditionnelle `select..case..end select` pour tester le contenu de cette variable :

```
Select Case var
   Case "Visual"
      WScript.Echo "the value of var is 'Visual'."
   Case "KSH"
      WScript.Echo "the value of var is 'KSH'."
   Case Else
      WScript.Echo "nothing else."
End Select

>>the value of var is 'Visual'.
```

Deux tests sont réalisés à l'aide de la clause `case` et la clause `case else` existe pour le cas où toutes les conditions testées sont fausses. Comme prévu, le résultat renvoyé par la structure conditionnelle indique que l'instruction issue de la première clause `case` a été exécutée.

Sur cette base, si d'autres tests doivent être effectués sur la variable `var`, il suffira d'insérer des clauses `case` supplémentaires au sein de la structure conditionnelle `select..case..end select`.

Windows PowerShell

Les principales structures conditionnelles dans l'écosystème de Windows PowerShell sont au nombre de trois :

- `if..else` ;
- `if..elseif..else` ;
- `switch`.

La structure if..else

L'instruction `if..else` a une syntaxe identique à son équivalent en langage Perl.

Syntaxe de l'instruction if..else

```
if (<expression conditionnelle>) 1
{
    <liste d'instructions> 2
}
[else 3
{
    <liste d'instructions> 4
}]
```

Une expression conditionnelle (1), dont le contenu peut être extrêmement riche, est évaluée. En fonction du résultat de l'évaluation, un bloc d'instructions est exécuté (2) (4), le bloc `else` (3) permettant justement une certaine alternative.

La structure suivante, dans la continuité des exemples de ce chapitre, compare deux variables afin de vérifier si l'une, `$var1`, est inférieure ou égale à une autre, `$var2` :

```
$var1 = 35
$var2 = 70

if ($var1 -le $var2) {
    Write-Host "var1 is 'less or equal' than var2."
} else {
    Write-Host "var1 is not 'less or equal' than var2."
}

>>var1 is 'less or equal' than var2.
```

La structure if..elseif..else

PowerShell, lui aussi, propose une extension de l'instruction conditionnelle `if..else`.

Syntaxe de l'instruction if..elseif..else

```
if (<expression conditionnelle>)
{
    <liste d'instructions_1>
}
[elseif (1) (<expression conditionnelle>)
{
    <liste d'instructions_2>
}]
[else
{
    <liste d'instructions_3>
}]
```

Le bloc `elseif` (1) donne la possibilité de « brancher » plusieurs conditions les unes à la suite des autres, comme le montre l'exemple suivant :

```
$var = "PowerShell"

if ($var.Length -eq 12) {
    Write-Host "The number of characters is 12."
} elseif ($var.Length -eq 11) {
    Write-Host "The number of characters is 11."
} elseif ($var.Length -eq 10) {
    Write-Host "The number of characters is 10."
```

```
} else {
    Write-Host "Nothing else."
}

>>The number of characters is 10.
```

Cet exemple met surtout en évidence le caractère pluriel de l'existence de la clause `elseif` au sein d'une structure conditionnelle.

La structure switch

Au contraire de Perl et Python, Windows PowerShell dispose d'une véritable instruction `switch`.

Syntaxe de l'instruction switch.

```
switch [-regex|-wildcard|-exact] (2) (valeur) (1)
{
   chaîne|nombre|variable|{expression} {liste d'instructions}
   default (3) {liste d'instructions}
}
```

La structure conditionnelle `switch` dans l'écosystème PowerShell est complexe ; une valeur (1) lui est passée en argument, qui peut être traitée de différentes manières (2) :

- une chaîne `regex[-regex]` ;
- une chaîne contenant un caractère générique `[-wildcard]` ;
- une chaîne avec correspondance exacte `[-exact]`.

Puis des conditions sont évaluées et un bloc d'instructions est exécuté en cas de résultat vrai. Si aucune des conditions n'est vraie, PowerShell exécute, s'il existe, le bloc d'instructions lié à la clause `default` (3).

Voici un exemple de structure conditionnelle `switch` :

```
$var = "PowerShell"

switch -regex ($var) {
    'Shell$' {
        "La valeur se termine par le mot 'Shell'."
    }
    '^P' {
        "La valeur commence par la lettre 'P'."
    }
    default {
        "Autre chose."
    }
}
```

```
>>La valeur se termine par le mot 'Shell'.
>>La valeur commence par la lettre 'P'.
```

Une variable nommée `$var` y est d'abord définie en tant que chaîne. Elle est ensuite passée en argument à la structure conditionnelle `switch` qui effectue à son tour plusieurs évaluations avant de repasser le contrôle à l'interpréteur.

NOTE Évaluations de conditions au sein de l'instruction switch

Les évaluations de conditions au sein de l'instruction `switch` ne sont pas mutuellement exclusives par défaut.

7

Les tableaux et dictionnaires

Au cours du développement du code, un programmeur ou un administrateur est amené à manipuler des séquences de valeurs pour réaliser des opérations très particulières. Ces données sont souvent complexes et les stocker dans de simples scalaires n'est pas une méthode adaptée.

Tous les langages de script ont dans leur proposition algorithmique des constructions dédiées à ce genre de manipulation. Il faut cependant souligner que toutes ne sont pas égales en termes de marges de manœuvre. La gestion des collections de valeurs n'est, par exemple, pas la même en Perl et en KSH, car le niveau de flexibilité offert varie très sensiblement. Ce chapitre présente deux constructions majeures en matière de collection de valeurs : les tableaux et les dictionnaires.

KSH

Même si les perspectives sont limitées, KSH offre la possibilité de créer des tableaux ainsi que des dictionnaires.

Créer un tableau

Créer un tableau avec Korn Shell se fait, entre autres, à l'aide de la commande `typeset`. Commençons par créer un tableau nommé `tab` :

```
typeset -a tab[0]='ksh' tab[1]='csh' tab[2]='bash' tab[3]='zsh'
```

Ce tableau contient quatre éléments. Il est construit sur la base d'un index commençant à zéro, grâce à l'option `-a`. La quatrième valeur a donc l'index `3`.

Manipuler un tableau

Essayons tout d'abord de lister tous les éléments du tableau `tab` :

```
echo "Array elements -> ${tab[@]}"

>>Array elements -> ksh csh bash zsh
```

Obtenir la liste des éléments d'un tableau peut s'effectuer soit par la notation `${tab[@]}`, soit par la notation `${tab[*]}`.

Pour obtenir le premier ou le troisième élément, il faut passer par l'utilisation des index :

```
echo "First element -> ${tab[0]}"
echo "Third element -> ${tab[2]}"

>>First element -> ksh
>>Third element -> bash
```

Pour obtenir le nombre d'éléments du tableau `tab`, il faut utiliser la notation donnant la liste des éléments, précédée du caractère `#` :

```
echo "Numbers of elements -> ${#tab[@]}"

>>Numbers of elements -> 4
```

Créer un dictionnaire

La commande `typeset` sert également à créer des dictionnaires, à l'aide de l'option `-A` :

```
typeset -A dict
```

Ici, le dictionnaire s'appelle `dict`, mais ne contient rien. Pour créer les clés et les valeurs associées, le recours à l'opérateur d'affectation `=` est indispensable :

```
dict["First"]="ksh"
dict["Second"]="bash"
dict["Third"]="zsh"
dict["Fourth"]="csh"
```

Manipuler un dictionnaire

Contrairement aux tableaux, qui sont basés sur la notion d'index, les dictionnaires le sont sur la notion d'association. Voici la liste des clés constituant le dictionnaire de l'exemple précédent :

```
echo "List of elements -> ${!dict[@]}"

>>List of elements -> First Fourth Second Third
```

Obtenir la liste des clés d'un dictionnaire est très utile dans le cadre de l'écriture de scripts. Si la totalité des valeurs doit être obtenue sans les clés associées, le nom du dictionnaire doit être utilisé avec le caractère * :

```
echo "List of elements -> ${dict[*]}"

>>List of elements -> ksh csh bash zsh
```

Enfin, trouver la valeur associée à une clé particulière s'opère de la façon suivante :

```
echo "First element -> ${dict['First']}"
echo "Third element -> ${dict['Third']}"

>>First element -> ksh
>>Third element -> zsh
```

Perl

Perl offre des perspectives plus larges que celles de KSH en matière de gestion de tableaux et de dictionnaires.

Créer un tableau

Définir un tableau en Perl débute avec le caractère @ :

```
@arraylist = (1,2,3,4,5,6,7,8,9,10);
```

Les éléments du tableau sont déclarés entre parenthèses et séparés par des virgules. Il existe d'autres façons de déclarer un tableau en Perl mais, dans le cadre de cet ouvrage, nous nous contenterons de cette approche de base.

Manipuler un tableau

Accéder aux éléments d'un tableau nécessite le caractère `$` et non plus le caractère `@` :

```
print("The first array element is -> $arraylist[0]\n");

>>The first array element is -> 1
```

Comme en KSH, l'accès à un élément utilise un index qui commence à `0`.

La boucle suivante énumère l'ensemble des éléments du tableau `@arraylist` de manière ordonnée :

```
foreach my $i (@arraylist) {
    print("$i is the value..\n");
}

>>1 is the value..
>>2 is the value..
>>3 is the value..
>>4 is the value..
>>5 is the value..
>>6 is the value..
>>7 is the value..
>>8 is the value..
>>9 is the value..
>>10 is the value..
```

La structure en question est une boucle `foreach` que nous détaillerons dans un prochain chapitre. Elle nous a permis de parcourir la totalité du tableau `@arraylist` afin d'intégrer chaque élément au sein d'une autre instruction.

Pour obtenir le nombre d'éléments d'un tableau, une des approches possibles consiste à utiliser la fonction `scalar()` :

```
print(scalar(@arraylist));

>>10
```

Cette fonction est très utile en ce sens qu'elle évalue un tableau passé en argument en contexte de scalaires. Autrement dit, `10`, qui est le résultat de l'appel de cette fonction, représente le nombre de scalaires contenus dans le tableau `@arraylist`.

Créer un dictionnaire

Créer un dictionnaire en Perl commence par le caractère `%` :

```
%hash = ("Name" => "AlKindi", "Age" => "31", "Job" => "Kernel Devel");
```

Chaque valeur d'un dictionnaire est associée à une clé. Dans le dictionnaire `%hash`, trois associations ont donc été établies avec le caractère `=>`.

Manipuler un dictionnaire

Là aussi, on accède aux éléments d'un dictionnaire à l'aide du caractère `$`, comme pour les tableaux, Perl considérant chaque élément comme un scalaire. Une valeur est obtenue via sa clé associée :

```
print($hash{'Job'});

>>Kernel Devel
```

Dans cet exemple, afin d'obtenir la valeur `Kernel Devel`, la clé `Job` doit être délimitée entre accolades `{}`.

La fonction `keys()` affiche l'ensemble des clés d'un dictionnaire :

```
print(keys(%hash));

>>NameAgeJob
```

Le résultat nous montre bien l'ensemble des clés du dictionnaire `%hash`, mais l'affichage n'est guère agréable. Séparons-les d'abord par un espace. Pour ce faire, la fonction `join()` est invoquée :

```
print(join(' ', keys(%hash)));

>>Name Age Job
```

Si ce sont les valeurs, et non les clés, qui doivent être affichées, alors il faut faire appel à la fonction `values()` :

```
print(join(' ', values(%hash)));

>>AlKindi 31 Kernel Devel
```

Bien évidemment, les valeurs peuvent être triées, grâce à la fonction `sort()` :

```
print(join(' ', sort(values(%hash))));

>>31 AlKindi Kernel Devel
```

La sortie affiche d'abord la valeur `31` au lieu de `AlKindi`.

Python

La gestion des tableaux et dictionnaires en Python est aussi flexible qu'en langage Perl.

Créer un tableau

La création d'un tableau en Python consiste à lister ses éléments entre crochets `[]`, séparés par des virgules :

```
tab = [1,2,3,4,5,6,7,8,9,10]
```

Ce tableau, nommé `tab`, contient exactement dix valeurs de même type.

Manipuler un tableau

L'accès aux éléments d'un tableau est extrêmement simple. Par exemple, le nom du tableau suffit pour afficher tous ses éléments :

```
print(tab)

>>[1,2,3,4,5,6,7,8,9,10]
```

Afficher un élément de manière isolée nécessite de passer par son index, qui commence à `0`. Affichons le troisième élément du tableau `tab` :

```
print(tab[2])

>>3
```

Si cette valeur doit être modifiée, une des méthodes possibles est d'utiliser l'opérateur `=` :

```
tab[2] = 11
print(tab[2])

>>11
```

Le nombre d'éléments d'un tableau est obtenu via la fonction `len()` :

```
print(len(tab))

>>10
```

Les tableaux créés avec Python sont, pour certains d'entre eux, dynamiques. Il existe en effet plusieurs genres de tableaux dans ce langage. Par dynamisme, nous entendons une potentielle modification de sa structure.

```
tab.append(11)
print(tab)

>>[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Dans cet exemple, la méthode `append()` a ajouté un élément au tableau `tab`. La suppression est aussi possible, via la méthode `remove()` :

```
tab.remove(7)
print(tab)

>>[1, 2, 3, 4, 5, 6, 8, 9, 10, 11]
```

Ici, la valeur `7` a été supprimée du tableau `tab`.

Créer un dictionnaire

Pour créer un dictionnaire, nous utilisons la fonction `dict()` :

```
d = dict(one=1, two=2, three=3)
print(d)

>>{'two': 2, 'one': 1, 'three': 3}
```

Manipuler un dictionnaire

L'affichage précédent indique que les clés n'ont pas été affichées dans l'ordre de création.

Nous pouvons les trier dans l'ordre alphabétique, à l'aide de la fonction `sorted()` :

```
print(sorted(d.keys()))

>>['one', 'three', 'two']
```

Les clés sont correctement triées, mais les valeurs correspondantes, elles, ne sont pas du tout affichées : la méthode `keys()` ne cible que les clés d'un dictionnaire et non les valeurs associées. Ceci n'est pas un problème, une boucle peut très bien accomplir cet objectif double :

```
for k in sorted(d.keys()):
    print(k, d[k])

>>one 1
>>three 3
>>two 2
```

La boucle `for` s'inscrit ici dans une logique de continuité vis-à-vis de la fonction `sorted()`.

Tout comme un certain type de tableaux, les dictionnaires sont des objets dont la structure peut évoluer. Modifions la valeur associée à la clé `one` :

```
d['one'] = 4
print(d)

>>{'two': 2, 'three': 3, 'one': 4}
```

La chaîne `'one'`, au lieu d'un index numérique, permet d'accéder à sa valeur.

VBS

Grâce à COM *(Component Object Model)*, Visual Basic Scripting se trouve très bien doté en matière de création de tableaux et de dictionnaires.

Créer un tableau

La fonction `Array()` est l'outil majeur pour créer des tableaux dans l'écosystème VBS :

```
Dim tab
tab = Array(10,20,30,40,50,60,70,80,90,100)
```

Dans cet exemple, dix valeurs numériques séparées par des virgules ont été passées en arguments à la fonction `Array()`.

Manipuler un tableau

En VBS, les valeurs d'un tableau ne sont pas figées. Par exemple, changeons la valeur du quatrième élément de l'exemple précédent :

```
tab(3) = 110
WScript.Echo tab(3)

>>110
```

Parcourir l'ensemble des éléments d'un tableau est une opération idéalement accomplie à l'aide d'une boucle. Commençons par créer un compteur :

```
c = 0
```

La boucle `for each` va itérer à travers chaque valeur du tableau `tab` :

```
For Each v In tab
   WScript.Echo "The value " & v & " has an index of " & c & "."
   c = c + 1
Next
```

Une variable, ici nommée `v`, représente de manière dynamique chaque élément parcouru. À chaque tour, le compteur `c` est incrémenté d'une unité et l'instruction est répétée dix fois :

```
>>The value 10 has an index of 0.
>>The value 20 has an index of 1.
>>The value 30 has an index of 2.
>>The value 110 has an index of 3.
>>The value 50 has an index of 4.
>>The value 60 has an index of 5.
>>The value 70 has an index of 6.
>>The value 80 has an index of 7.
>>The value 90 has an index of 8.
>>The value 100 has an index of 9.
```

Créer un dictionnaire

Il existe un objet COM dédié à la création de dictionnaires :

```
Dim dict
Set dict = CreateObject("Scripting.Dictionary")
```

`Scripting.Dictionary` est un objet COM natif servant essentiellement à la création de dictionnaires et qui fournit les méthodes de base pour manipuler ces derniers.

Manipuler un dictionnaire

Le dictionnaire que nous venons de créer ne contient aucune clé. Nous utiliserons donc la méthode `add()` du dictionnaire `dict` en lui fournissant en arguments une clé ainsi que sa valeur associée :

```
dict.Add "a", "KSH"
dict.Add "b", "Perl"
dict.Add "c", "Python"
dict.Add "d", "VBS"
dict.Add "e", "PowerShell"
```

Notre dictionnaire contient à présent cinq paires clés/valeurs. Si nous voulons obtenir la valeur correspondant à la clé `d`, cette dernière doit être précisée entre parenthèses :

```
WScript.Echo dict("d")

>>VBS
```

La méthode `keys()` retourne un tableau contenant toutes les clés d'un dictionnaire :

```
For Each v In dict.Keys()
   WScript.Echo v
Next

>>a
>>b
>>c
>>d
>>e
```

Il en est de même en ce qui concerne les valeurs correspondantes, retournées par la méthode `items()` :

```
For Each v In dict.Items()
   WScript.Echo v
Next

>>Python
>>KSH
>>Perl
>>Python
>>VBS
>>PowerShell
```

Enfin, il est possible de supprimer une clé d'un dictionnaire via la méthode `remove()`. Supprimons la clé `e` :

```
dict.Remove("e")
```

Après l'exécution de cette méthode, VBS ne nous a pas retourné de valeur indiquant si l'opération s'est déroulée correctement. Cependant, une autre méthode nommée `exists()` renvoie `true` si une clé particulière subsiste encore au sein d'un dictionnaire :

```
If dict.Exists("e") = True  Then
   WScript.Echo "e exists."
Else
   WScript.Echo "e doesn't exist."
End If

>>e doesn't exist.
```

Windows PowerShell

La création de tableaux et de dictionnaires avec Windows PowerShell est évidemment possible. La technologie .NET étant sa substance principale, ces structures n'en sont pas moins riches de possibilités.

Créer un tableau

Plusieurs approches sont possibles concernant la création de tableaux, mais la plus recommandée consiste à utiliser l'expression `@()` :

```
PS> $tab = @(1,2,3,4,5,6,7,8,9,10)
```

La variable `$tab` est un tableau contenant exactement dix valeurs :

```
PS> $tab

>>1
>>2
>>3
>>4
>>5
>>6
>>7
>>8
>>9
>>10
```

Manipuler un tableau

Un tableau étant un objet .NET, il est nécessaire de le manipuler avec précaution pour qu'il soit exploité correctement :

```
PS> Get-Member -InputObject $tab

   TypeName : System.Object[]

Name           MemberType           Definition
----           ----------           ----------
Count          AliasProperty        Count = Length
Add            Method               int IList.Add(System.Object value)
Address        Method               System.Object&, mscorlib, Version=4.
Clear          Method               void IList.Clear()
Clone          Method               System.Object Clone(), System.Object
```

```
CompareTo       Method                int IStructuralComparable.CompareTo(
Contains        Method                bool IList.Contains(System.Object va
CopyTo          Method                void CopyTo(array array, int index),
Equals          Method                bool Equals(System.Object obj), bool
Get             Method                System.Object Get(int )
GetEnumerator   Method                System.Collections.IEnumerator GetEn
GetHashCode     Method                int GetHashCode(), int IStructuralEq
GetLength       Method                int GetLength(int dimension)
GetLongLength   Method                long GetLongLength(int dimension)
GetLowerBound   Method                int GetLowerBound(int dimension)
GetType         Method                type GetType()
GetUpperBound   Method                int GetUpperBound(int dimension)
GetValue        Method                System.Object GetValue(Params int[]
IndexOf         Method                int IList.IndexOf(System.Object valu
Initialize      Method                void Initialize()
Insert          Method                void IList.Insert(int index, System.
Remove          Method                void IList.Remove(System.Object valu
RemoveAt        Method                void IList.RemoveAt(int index)
Set             Method                void Set(int , System.Object )
SetValue        Method                void SetValue(System.Object value, i
ToString        Method                string ToString()
Item            ParameterizedProperty System.Object IList.Item(int index)
IsFixedSize     Property              bool IsFixedSize {get;}
IsReadOnly      Property              bool IsReadOnly {get;}
IsSynchronized  Property              bool IsSynchronized {get;}
Length          Property              int Length {get;}
LongLength      Property              long LongLength {get;}
Rank            Property              int Rank {get;}
SyncRoot        Property              System.Object SyncRoot {get;}
```

La cmdlet `Get-Member` nous a permis ici de nous familiariser avec les membres du tableau `$tab`, mais commençons par l'utiliser sur la base de position d'index :

```
PS> $tab[5]
>>6
PS> $tab[5] = 19
PS> $tab[5]
>>19
```

Dans cet exemple, le sixième élément du tableau a d'abord été affiché, puis modifié. Cette manipulation est classique. Intéressons-nous maintenant à certaines méthodes et propriétés de l'objet `$tab`. Premièrement, combien d'éléments composent le tableau `$tab` ? La réponse est du côté de la propriété `count` :

```
PS> $tab.Count
>>10
```

Précédemment, la sixième valeur a été modifiée à partir de son index, mais cette modification est aussi possible à l'aide de la méthode `SetValue()`, qui prend en arguments la nouvelle valeur ainsi que la position d'index héritant de cette dernière :

```
PS> $tab.SetValue(17,5)
PS> $tab[5]

>>17
```

L'ajout d'un nouvel élément se fait à l'aide de l'opérateur d'affectation (`+=`) :

```
PS> $tab += 11
PS> $tab

>>1
>>2
>>3
>>4
>>5
>>17
>>7
>>8
>>9
>>10
>>11
```

Et le nombre d'éléments devrait être maintenant de onze :

```
PS> $tab.Count
>>11
```

Créer un dictionnaire

En PowerShell, un dictionnaire est défini à l'aide de l'expression `@{}` :

```
PS> $hash = @{'one'=1 ; 'two'=2 ; 'three'=3}
```

Chaque clé est séparée de sa valeur par le symbole `=` et chaque paire clé/valeur est séparée d'une autre par un point-virgule. La variable `$hash` contient ici trois clés :

```
PS> $hash

>>Name                           Value
>>----                           -----
>>one                            1
>>three                          3
>>two                            2
```

Manipuler un dictionnaire

Vous l'aurez sans doute deviné, un dictionnaire est aussi un objet dans l'écosystème PowerShell :

```
PS> $hash | Get-Member

   TypeName : System.Collections.Hashtable

Name              MemberType            Definition
----              ----------            ----------
Add               Method                void Add(System.Object
Clear             Method                void Clear(), void IDi
Clone             Method                System.Object Clone(),
Contains          Method                bool Contains(System.O
ContainsKey       Method                bool ContainsKey(Syste
ContainsValue     Method                bool ContainsValue(Sys
CopyTo            Method                void CopyTo(array arra
Equals            Method                bool Equals(System.Obj
GetEnumerator     Method                System.Collections.IDi
GetHashCode       Method                int GetHashCode()
GetObjectData     Method                void GetObjectData(Sys
GetType           Method                type GetType()
OnDeserialization Method                void OnDeserialization
Remove            Method                void Remove(System.Obj
ToString          Method                string ToString()
Item              ParameterizedProperty System.Object Item(Sys
Count             Property              int Count {get;}
IsFixedSize       Property              bool IsFixedSize {get;
IsReadOnly        Property              bool IsReadOnly {get;}
IsSynchronized    Property              bool IsSynchronized {g
Keys              Property              System.Collections.ICo
SyncRoot          Property              System.Object SyncRoot
Values            Property              System.Collections.ICo
```

La propriété `Keys` liste toutes les clés d'un dictionnaire :

```
PS> $hash.Keys

>>one
>>two
>>three
```

Les valeurs sont quant à elle listées grâce à la propriété `Values` :

```
PS> $hash.Values

>>1
>>3
>>2
```

La méthode `Add()` ajoute une clé à un dictionnaire :

```
PS> $hash.Add('four',4)
PS> $hash

>>Name                          Value
>>----                          -----
>>four                          4
>>one                           1
>>three                         3
>>two                           2
```

Si l'accent doit être mis sur une clé, une des méthodes possibles est alors d'utiliser une indexation basée sur les chaînes :

```
PS> $hash['three']
>>3
```

Une autre consiste à référencer directement la clé elle-même :

```
PS> $hash.three
>>3
```

Un dictionnaire est une structure très flexible. Le problème est qu'elle est souvent mal maîtrisée par ceux qui y ont recours. Un entraînement est donc sans nul doute indispensable.

8

Les boucles

En matière de programmation, répéter une action un certain nombre de fois est souvent une démarche inévitable. Le phénomène d'itération est toutefois un processus redoutablement efficace, car une instruction particulière peut s'appliquer à une multitude de valeurs issues, par exemple, d'une collection.

Tous les langages de script, sans exception, disposent de structures permettant d'effectuer des itérations. Les étudier revient à développer un code plus concis et plus efficace.

KSH

Trois boucles constituent le socle fondamental d'itération dans le langage Korn Shell :

- `for` ;
- `while` ;
- `until`.

La boucle for..in..do..done

La boucle `for` est idéale pour parcourir un tableau de valeurs.

Syntaxe de la boucle for en KSH

```
for c (1) in 1 2 3 4 5 6 7 8 9 10 (2)
 (3) do echo $c
done

>>1
>>2
>>3
>>4
>>5
>>6
>>7
>>8
>>9
>>10
```

Cet exemple met en évidence un tableau de dix valeurs (2) parcouru par une boucle `for`. Une variable d'initialisation (1) prend successivement chacune de ces valeurs. Le corps de la boucle `for` est constitué, quant à lui, d'un bloc d'instructions (3) exécuté pour chacune des valeurs.

Le résultat d'une commande peut très bien être traité comme un tableau de valeurs et donc parcouru par une boucle `for`. Dans ce cas, la commande doit figurer entre deux symboles ` :

```
for c in `seq 1 10`
   do echo $c
done

>>1
>>2
>>3
>>4
>>5
>>6
>>7
>>8
>>9
>>10
```

La boucle while..do..done

Si une instruction doit être exécutée tant qu'une condition est vraie, la boucle `while` est alors un outil privilégié. Tant qu'une condition (1) est vraie, la boucle `while` répète un bloc d'instructions (2). Il faut donc bien veiller à ce que l'itération ne soit pas effectuée à l'infini.

Syntaxe de la boucle while..do..done en KSH

```
a=10
c=0
```

```
while ❶ [[ $c -le $a ]]; do ❷
   echo $c
   ((c++))
done

>>0
>>1
>>2
>>3
>>4
>>5
>>6
>>7
>>8
>>9
>>10
```

Ici, deux variables `a` et `c` sont définies et initialisées respectivement à `10` et `0`. Puis une boucle `while` vérifie si `c` est inférieure ou égale à `a`. Tant que le résultat de l'évaluation est vrai, la valeur de `c` est affichée et incrémentée d'une unité. Comme attendu, la boucle `while` s'est bien arrêtée lorsque la valeur de `c` a atteint `10`.

La boucle until..do..done

La boucle `until` propose exactement l'inverse de la boucle `while` : tant qu'une condition est fausse ❶, un bloc d'instructions est exécuté ❷. La boucle s'arrêtera donc quand la condition évaluée sera vraie.

Syntaxe de la boucle until..do..done en KSH

```
a=10
c=0

until ❶ [[ $c -gt $a ]]; do ❷
   echo $c
   ((c++))
done
```

Reprenons l'exemple précédent :

```
a=10
c=0

until [[ $c -gt $a ]]; do
   echo $c
   ((c++))
done
```

```
>>0
>>1
>>2
>>3
>>4
>>5
>>6
>>7
>>8
>>9
>>10
```

Comme nous pouvons le constater, la sortie est exactement la même que pour la boucle `while`. Ici, onze tours ont été nécessaires à la boucle `until`. Donc, à onze reprises, la condition était fausse.

Perl

Le langage Perl compte quatre boucles principales :

- `for` ;
- `foreach` ;
- `while` ;
- `until`.

La boucle for

La structure `for` est une boucle très proche de son équivalent en langage C.

Syntaxe de la boucle for en Perl

```
for ($i = 0 (1); $i < 10 (2); $i++ (3))
{
    print "$i\n"; (4)
}
```

Dans son fonctionnement classique, une variable d'initialisation est créée et initialisée (1). Elle est incrémentée d'une unité (3) à chaque tour tant qu'une condition (2) est vraie. En outre, un bloc d'instructions (4) est exécuté par Perl si la condition évaluée est vraie.

Réadaptons l'exemple utilisé dans la section dédiée à KSH :

```
for ($i = 0; $i <= 10; $i++) {
    print("$i\n");
}
```

```
>>0
>>1
>>2
>>3
>>4
>>5
>>6
>>7
>>8
>>9
>>10
```

NOTE **Utilisation de la boucle for**

La boucle `for` est, dans certaines dimensions liées à son utilisation, une structure identique à la boucle `foreach`.

La boucle foreach

La structure `foreach` sert à parcourir une collection de valeurs.

Syntaxe de la boucle foreach en Perl

```
foreach my $i (1) (1..10) (2)
{
    print "$i is the value..\n"; (3)
}

>>1 is the value..
>>2 is the value..
>>3 is the value..
>>4 is the value..
>>5 is the value..
>>6 is the value..
>>7 is the value..
>>8 is the value..
>>9 is the value..
>>10 is the value..
```

Une collection de valeurs (2) parcourue par une boucle `foreach` peut être de n'importe quel type (liste, tableau, dictionnaire, etc.). Chaque valeur est contenue dans une variable (1) et une instruction (3) est exécutée pour chaque valeur.

La boucle while

En Perl comme en KSH, une boucle `while` exécute un bloc de code tant qu'une condition est vraie (1).

Syntaxe de la boucle while en Perl

```
$c = 0;
$x = 10;

while (1) ($c <= $x) {
    print("$c\n");
    $c++;
}

>>0
>>1
>>2
>>3
>>4
>>5
>>6
>>7
>>8
>>9
>>10
```

La boucle until

La boucle `until` se déploie dans une logique exactement inverse de celle de la boucle `while` : tant qu'une condition est fausse (1), un bloc d'instructions (2) est exécuté.

Syntaxe de la boucle until en Perl

```
$c = 0;
$x = 10;

until (1) ($c > $x) {
    print("$c\n"); (2)
    $c++;
}

>>0
>>1
>>2
>>3
>>4
>>5
```

```
>>6
>>7
>>8
>>9
>>10
```

REMARQUE **Boucles until et while**

Les boucles `until` et `while` sont à la fois classiques en raison de leur universalité, mais aussi très puissantes lorsqu'il s'agit de parcourir tous types de séquences de valeurs.

Python

Il y a essentiellement deux façons d'exploiter des boucles dans l'arsenal du langage Python :

- `for` ;
- `while`.

La boucle for

La structure `for` du langage Python ressemble beaucoup à la boucle `foreach` que nous pouvons trouver dans d'autres langages de script. Une variable (1) prend successivement chacune des valeurs d'une liste (2). Une instruction (3) est exécutée pour chaque valeur parcourue.

Syntaxe de la boucle for en Python

```
for (1) i in (2) [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    print("{} is the value.".format(i)) (3)

>>1 is the value.
>>2 is the value.
>>3 is the value.
>>4 is the value.
>>5 is the value.
>>6 is the value.
>>7 is the value.
>>8 is the value.
>>9 is the value.
>>10 is the value.
```

La boucle while

La boucle `while` exécute une instruction (2) tant qu'une condition (1) est vraie. Transformons l'exemple de la section précédente en une boucle `while` (tant que `a` est inférieure ou égale à `b`, on l'affiche et on l'incrémente d'une unité).

Syntaxe de la boucle while en Python

```
a, b = 0, 10

while (1) a <= b:
    print("{} is the value.".format(a)) (2)
    a += 1

>>0 is the value.
>>1 is the value.
>>2 is the value.
>>3 is the value.
>>4 is the value.
>>5 is the value.
>>6 is the value.
>>7 is the value.
>>8 is the value.
>>9 is the value.
>>10 is the value.
```

VBS

VBS dispose de plus de boucles que Python :

- `for..next` ;
- `for each..next` ;
- `do while..loop` ;
- `do until..loop`.

La boucle for..next

La boucle `for..next` est utile si un bloc de code doit être exécuté un certain nombre de fois. Le principe de fonctionnement est simple : une variable d'initialisation (1) est définie en tant que compteur, dont on précise une valeur de départ ainsi qu'une valeur de fin (2). Cette variable augmente d'une unité à chaque tour et une instruction est exécutée (3).

Syntaxe de la boucle for..next en VBS

```
For (1) c = 0 To 10 (2)
   WScript.Echo c, "is the value." (3)
Next

>>0 is the value.
>>1 is the value.
>>2 is the value.
>>3 is the value.
```

```
>>4 is the value.
>>5 is the value.
>>6 is the value.
>>7 is the value.
>>8 is the value.
>>9 is the value.
>>10 is the value.
```

La boucle for each..next

Si un bloc de code doit s'appliquer plusieurs fois sur une collection de valeurs (2), l'instruction `for each..next` est alors la structure la plus adaptée.

Un bloc d'instructions (3) est exécuté pour chaque élément (1) de la collection (2). L'exemple suivant consiste à créer un tableau de six éléments, dont on affiche la valeur.

Syntaxe de la boucle for each..next en VBS

```
Dim coll(5)
coll(0)="zero"
coll(1)="one"
coll(2)="two"
coll(3)="three"
coll(4)="four"
coll(5)="five"

For Each (1) i In coll (2)
  WScript.Echo i (3)
Next

>>zero
>>one
>>two
>>three
>>four
>>five
```

La boucle do while..loop

Répéter l'exécution d'un bloc d'instructions (2) tant qu'une condition est vraie (1) est possible avec la boucle `do while..loop`.

Syntaxe de la boucle do while..loop en VBS

```
Do While (1) c <= d
   WScript.Echo c (2)
    c = c + 1
Loop
```

Les conditions évaluées peuvent également être plus complexes :

```
c = 0
d = 6

Do While c <= 10 And d > 5
   WScript.Echo c
   c = c + 1
Loop

>>0
>>1
>>2
>>3
>>4
>>5
>>6
>>7
>>8
>>9
>>10
```

La nécessité de maîtriser le niveau d'exécution d'une boucle `do while..loop` est donc primordiale, afin d'éviter les erreurs de logique.

La boucle do until..loop

La boucle `do until..loop` répond à une logique exactement inverse de celle de `do while..loop` : un bloc d'instructions (2) est exécuté jusqu'à ce qu'une condition (1) soit vraie.

Syntaxe de la boucle do until..loop en VBS

```
Do Until (1) c > d
   WScript.Echo c (2)
    c = c + 1
Loop
```

Un branchement de plusieurs conditions est ici aussi possible, mais attention à garder la maîtrise de leurs mécanismes :

```
c = 0
d = 6

Do Until c > 10 Or d < 5
   WScript.Echo c
   c = c + 1
Loop
```

```
>>0
>>1
>>2
>>3
>>4
>>5
>>6
>>7
>>8
>>9
>>10
```

Windows PowerShell

Les boucles existant dans l'écosystème de Windows PowerShell sont nombreuses. Nous n'en verrons que trois :

- `for` ;
- `foreach` ;
- `while`.

La boucle for

La boucle `for` sert à répéter un bloc d'instructions plusieurs fois. Un compteur (1) est défini et repris au sein d'une condition (2) dans son processus d'évaluation. Tant que cette condition est vraie, un bloc d'instructions (4) est exécuté. Il est important d'incrémenter le compteur (3) afin que la boucle ne soit pas infinie.

La boucle `for` suivante parcourt un tableau de services tournant dans un système d'exploitation et affiche leurs noms.

Syntaxe de la boucle for en PowerShell

```
$services = @(Get-Service)

for (1 $i=0; 2 $i -lt $services.Length; 3 $i++) {
   Write-Host $services[$i].ServiceName 4
}

>>AdobeARMservice
>>AeLookupSvc
>>ALG
>>AppIDSvc
>>Appinfo
>>AppReadiness
>>AppXSvc
>>AudioEndpointBuilder
```

```
>>Audiosrv
>>AVerRemote
>>AVerScheduleService
>>AxInstSV
>>BcmBtRSupport
>>BDESVC
>>BFE
>>BITS
>>BrokerInfrastructure
>>Browser
>>
...
```

La boucle foreach

PowerShell propose la boucle `foreach`, utile lorsqu'il s'agit d'itérer à travers une séquence de valeurs. Un même bloc d'instructions (3) est exécuté successivement sur chaque élément (1) issu d'une collection de valeurs (2).

L'exemple de la section précédente est repris ici via une simple traduction d'une boucle `for` en une boucle `foreach`. Autrement dit, ces deux boucles peuvent servir les mêmes objectifs, même si elles ont chacune un dessein qui leur est propre.

Syntaxe de la boucle foreach en PowerShell

```
$services = @(Get-Service)

foreach (1 $service in 2 $services) {
   Write-Host $service.ServiceName 3
}

>>AdobeARMservice
>>AeLookupSvc
>>ALG
>>AppIDSvc
>>Appinfo
>>AppReadiness
>>AppXSvc
>>AudioEndpointBuilder
>>Audiosrv
>>AVerRemote
>>AVerScheduleService
>>AxInstSV
>>BcmBtRSupport
>>BDESVC
>>BFE
>>BITS
```

```
>>BrokerInfrastructure
>>Browser
...
```

La boucle while

Comme dans les autres langages, tant qu'une condition est vraie (1), un bloc d'instructions (2) est exécuté. Toujours dans l'esprit du précédent exemple, voici une boucle `while` affichant le nom de tous les services tournant dans un système d'exploitation.

Syntaxe de la boucle while en PowerShell

```
$services = @(Get-Service)
$i = 0

while (1) ($i -lt $services.Length) {
   Write-Host $services[$i].ServiceName (2)
   $i++
}

>>AdobeARMservice
>>AeLookupSvc
>>ALG
>>AppIDSvc
>>Appinfo
>>AppReadiness
>>AppXSvc
>>AudioEndpointBuilder
>>Audiosrv
>>AVerRemote
>>AVerScheduleService
>>AxInstSV
>>BcmBtRSupport
>>BDESVC
>>BFE
>>BITS
>>BrokerInfrastructure
>>Browser
...
```

La sortie, tronquée, nous montre que le résultat est bien le même que dans les sections précédentes. Notez que le compteur `$i` a été créé à l'extérieur de la boucle `while`, mais incrémenté au sein de cette dernière.

9

Les fonctions

Dans le monde des langages de script, les fonctions aident les programmeurs à être plus créatifs et à mieux appréhender l'extension des fonctionnalités natives. Elles accroissent sensiblement l'efficacité et la productivité des scripts écrits. D'un langage à un autre, le statut d'une fonction n'est pas le même ; maîtriser leurs différents modes de fonctionnement est donc un savoir fondamental.

KSH

Définir une fonction

La création d'une fonction en KSH est on ne peut plus simple. Une fonction a un nom (1), ainsi qu'un bloc de code qui lui est associé (2).

Définition d'une fonction en Korn Shell

```
func() (1) {
   echo "This is a statement." (2)
}
```

L'instanciation d'une fonction s'effectue par son nom ; son appel avec passage de paramètres se réalise à l'aide des variables spéciales `$n`.

Une fonction par l'exemple

Créons une fonction capable de vérifier si un fichier passé en argument existe ou non. Pour cela, nous aurons besoin de la commande `test` et de son argument `-f` :

```
test -f /etc/passwd
```

Ensuite, la valeur de sortie de la commande nous sera indispensable afin de savoir si un fichier existe :

```
if (test -f /etc/passwd) then
    echo "The file /etc/passwd exists."
else
    echo "The file /etc/passwd doesn't exist."
fi
```

Si la valeur de sortie est `0`, cela signifie que le fichier existe, mais si la valeur est autre (`1`), le fichier n'existe pas.

L'essentiel du code étant défini, intégrons-le au sein d'une fonction :

```
func_fexists(){
   filename=$1
   if (test -f $filename) then
      echo "The file $filename exists."
   else
      echo "The file $filename doesn't exist."
   fi
}
```

Pour que la fonction soit véritablement utile, il faut que le nom du fichier soit une variable. C'est une variable spéciale, ici `$1`, qui joue ce rôle.

La fonction `func_fexists()` peut donc être instanciée de la manière suivante :

```
func_fexists "/etc/passwd"

>>The file /etc/passwd exists.

func_fexists "/etc/notonline"

>>The file /etc/notonline doesn't exist.
```

Selon les objectifs recherchés, les fonctionnalités de `func_fexists()` sont évidemment extensibles à souhait.

Perl

Le concept de fonction, en Perl, est souvent une notion mal comprise des programmeurs. En effet, dans l'absolu, il faudrait plutôt parler de procédure, à savoir un programme comportant un bloc de code susceptible d'être appelé plusieurs fois au cours de l'exécution de ce dernier. Toutefois, lorsqu'une procédure renvoie une valeur, alors ladite procédure devient une fonction.

Définir une fonction

En Perl, une fonction est définie à l'aide du mot-clé `sub` (1).

Syntaxe d'une fonction en Perl

```
sub (1) func
{
    print("This is a statement."); (3)
    return 7; (2)
}
```

Le mot-clé `return` permet à une fonction de retourner une valeur (2). Le bloc d'instructions (3) associé à la fonction doit être délimité par des accolades `{}`.

Une fonction par l'exemple

La fonction que nous allons définir consiste en une simple opération mathématique. Pour être plus précis, la fonction créée additionnera trois arguments qui lui auront été passés. Les paramètres d'une fonction sont concentrés dans la variable tableau `@_`. Il suffit donc d'affecter les trois valeurs du tableau à trois variables, via l'opérateur d'affectation `=` (1).

Ces trois variables sont ensuite additionnées et la somme est affectée à la variable `$sum` (2).

Enfin, le résultat est retourné grâce au mot-clé `return` (3).

```
sub func_add
{
    ($arg1, $arg2, $arg3) = @_; (1)
    $sum = ($arg1 + $arg2 + $arg3); (2)
    return $sum; (3)
}
```

L'appel d'une fonction se fait entre autres à l'aide de l'opérateur `&`. Les arguments sont passés à `func_add` entre parenthèses et séparés par des virgules (4).

```
print(&func_add(7,8,9)); (4)

>>24
```

> **IMPORTANT** **L'appel des fonctions en Perl**
>
> Lors de l'appel d'une fonction, n'oubliez pas de respecter la casse.

Python

Les fonctions, comme le reste des structures dans le langage Python, sont extrêmement simples à mettre en œuvre.

Définir une fonction

Le mot-clé `def` précède le nom de la fonction (1) suivi de parenthèses contenant les éventuels arguments, puis du symbole deux-points. Le bloc d'instructions (2) appartenant à la fonction est caractérisé par une indentation.

Syntaxe d'une fonction en Python

```
def (1) func():
    print("This is a statement.") (2)
```

Une fonction par l'exemple

La fonction suivante accepte un argument de type numérique (1) et l'utilise pour limiter le processus d'exécution d'une boucle (2) :

```
def func_loop(num) (1):
    print('The limit of the loop is..', num)
    for c in range(0, num): (2)
        print(c, 'is the value.')
```

Il ne reste plus qu'à appeler la fonction `func_loop` en précisant en argument la valeur de `num` :

```
func_loop(10)

>>The limit of the loop is.. 10
>>0 is the value.
>>1 is the value.
>>2 is the value.
>>3 is the value.
>>4 is the value.
>>5 is the value.
>>6 is the value.
```

```
>>7 is the value.
>>8 is the value.
>>9 is the value.
```

Au sein de la boucle `for`, la fonction `range()` sert de générateur de tableau en utilisant `0` comme valeur de départ et la variable `num` comme valeur de fin.

VBS

Définir une fonction

En VBS, la définition d'une fonction est comprise entre les mots-clés `Function` (1) et `End Function` (2). Naturellement, une fonction a un nom (3) suivi de parenthèses (même sans arguments), puis un corps (4), bloc d'instructions exécuté par l'interpréteur lorsque la fonction est appelée.

Syntaxe d'une fonction en VBS

```
Function (1) func() (3)
    Wscript.Echo "This is a statement." (4)
End Function (2)
```

Une fonction par l'exemple

VBS est un langage très puissant, entre autres lorsqu'il s'agit d'interroger WMI *(Windows Management Instrumentation)*. WMI est une base de données très riche, permettant notamment de consulter les informations liées au système d'exploitation ou au matériel. Parmi les données importantes, il y a les journaux d'événements.

Tout administrateur système sait que les journaux d'événements sont organisés par catégories. Une démarche efficace concernant la gestion des journaux d'événements nécessite, tout d'abord, de bien maîtriser les niveaux de lecture de ces journaux qui, pour rappel, sont dans l'immense majorité des cas très nombreux. Ensuite, faire en sorte que cette démarche soit automatisée est une condition sine qua non pour atteindre une certaine efficacité.

Être conforme à ces logiques revient donc à développer une fonction capable de lire et d'analyser différentes catégories de journaux. Pour ce faire, il faut que nous avancions étape par étape. Premièrement, étant donné que la fonction sera exécutée en local ou à distance, une variable contenant le nom de la machine (1) est nécessaire (dans le code qui suit, `"."` correspond à la machine locale)

WMI étant la base de données contenant les informations qui nous intéressent, il faut créer une connexion à l'espace de nom `\root\cimv2` (2).

Cet espace de nom contient la classe WMI `Win32_NTLogEvent` (3), qui est utilisée pour obtenir des instances de journaux d'événements Windows. La liste des journaux est longue, ce qui nous amène à recourir à un filtre (clause `Where` (4)). La variable `LogType` (5) permettra au filtre de spécifier la catégorie du journal d'événements.

La variable `colLoggedEvents` (6) est une collection d'événements : elle doit être parcourue à l'aide d'une boucle (7) afin d'en extraire les événements de manière isolée.

Grâce à la boucle `For Each`, chaque événement peut être manipulé sur la base d'une sélection de propriétés en adéquation avec les objectifs recherchés. Ici, plusieurs propriétés ont été retenues, comme la catégorie de l'événement, sa source ou son type.

Nous encapsulons l'ensemble de ce code dans une fonction que nous nommerons `Get_Logs()` (8) :

```
Function Get_Logs(LogType) (8)

   strComputer = "." (1)
   Set objWMIService = GetObject("winmgmts:" _
      & "{impersonationLevel=impersonate}!\\" & strComputer & "\root\cimv2") (2)

   Set colLoggedEvents (6) = objWMIService.ExecQuery _
      ("Select * from Win32_NTLogEvent Where (4) Logfile='" & LogType (5) & "'") (3)

   For Each objEvent in colLoggedEvents (7)
      Wscript.Echo "Category: " & objEvent.Category
      Wscript.Echo "Computer Name: " & objEvent.ComputerName
      Wscript.Echo "Event Code: " & objEvent.EventCode
      Wscript.Echo "Record Number: " & objEvent.RecordNumber
      Wscript.Echo "Source Name: " & objEvent.SourceName
      Wscript.Echo "Time Written: " & objEvent.TimeWritten
      Wscript.Echo "Event Type: " & objEvent.Type
      Wscript.Echo
   Next

End Function
```

La fonction `Get_Logs()`, via son argument (`LogType`), procédera à une connexion au service WMI ainsi qu'à la collecte des informations de journaux d'événements à partir de l'argument spécifié. Il ne nous reste plus qu'à tester la fonction :

```
Get_Logs("Application")

>>Category: 0
>>Computer Name: win8
>>Event Code: 903
>>Record Number: 472
>>Source Name: Microsoft-Windows-Security-SPP
>>Time Written: 20140327215341.000000-000
>>Event Type: Information
```

```
>>Category: 0
>>Computer Name: win8
>>Event Code: 16384
>>Record Number: 471
>>Source Name: Microsoft-Windows-Security-SPP
>>Time Written: 20140327215341.000000-000
>>Event Type: Information

>>Category: 0
>>Computer Name: win8
>>Event Code: 902
>>Record Number: 470
>>Source Name: Microsoft-Windows-Security-SPP
>>Time Written: 20140327215314.000000-000
>>Event Type: Information
...
```

La sortie est longue et a volontairement été tronquée. Cependant, l'affichage met bien en évidence une disposition correspondant exactement à la requête effectuée par la fonction. Cette dernière pourra donc être utilisée un nombre illimité de fois, pour interroger tous types de journaux (`Application`, `System`, `Security`, etc.).

Windows PowerShell

PowerShell, en matière de définition de fonctions, propose une marge de manœuvre vraiment très importante. Cette section mettra en évidence la façon dont s'articule une fonction à partir de sa structure de base.

Définir une fonction

Comme avec Visual Basic Scripting, on utilise le mot-clé `function` (1) pour la déclaration.

Syntaxe d'une fonction en PowerShell

```
function (1) func {
    Write-Host "This is a statement."
}
```

En réalité, il y a plusieurs façons de déclarer une fonction (avec ou sans paramètres, communicative ou non, etc.), mais cette syntaxe constitue le socle fondamental en la matière.

Une fonction par l'exemple

Voici une fonction effectuant une requête WMI afin de consulter l'état des services existant au niveau du système d'exploitation :

```
function Get-Svc {

   Param( 1
          [String]$ComputerName,
          [String]$State
        )

   PROCESS { 2
          Get-WmiObject -ComputerName $ComputerName -Class Win32_Service -Filter
"State='$State'"
   }

}
```

Son principe est très simple : établir une connexion au service WMI pour y collecter les services sur la base de deux états. Le premier état (`Stopped`) représente les services arrêtés et le second (`Running`) ceux en cours d'exécution. D'autres valeurs d'état existent, mais les deux susmentionnées sont les plus utilisées.

La fonction `Get-Svc` dispose de deux paramètres : `$ComputerName` spécifie un nom de machine locale ou distante et `$State` est l'état des services collectés. Ces deux paramètres sont contenus au sein d'un bloc `Param` 1 spécialement dédié.

Ensuite vient la clause `PROCESS` 2 : non obligatoire, elle est censée contenir le code dit principal. En effet, la cmdlet `Get-WmiObject` associée à ses paramètres `-ComputerName`, `-Class` et `-Filter` y est invoquée pour effectuer une requête WMI. Une lecture de la fonction nous montre que les valeurs des paramètres `$ComputerName` et `$State` sont passés en tant que valeurs des paramètres `-ComputerName` et `-Filter` de la cmdlet `Get-WmiObject`.

> **CODAGE Pourquoi ne pas utiliser la cmdlet Get-WmiObject directement ?**
>
> Intégrer cette cmdlet au sein d'une fonction revient à ne fournir que les éléments les plus importants, tout en s'émancipant de la syntaxe compliquée.

Testons dès à présent notre fonction sans oublier ses deux paramètres :

```
PS> Get-Svc -ComputerName 'localhost' -State 'Running'

>>ExitCode  : 0
>>Name      : AdobeARMservice
>>ProcessId : 1744
>>StartMode : Auto
>>State     : Running
>>Status    : OK

>>ExitCode  : 0
>>Name      : Appinfo
>>ProcessId : 1012
>>StartMode : Manual
>>State     : Running
>>Status    : OK

>>ExitCode  : 0
>>Name      : AudioEndpointBuilder
>>ProcessId : 1116
>>StartMode : Auto
>>State     : Running
>>Status    : OK
…
```

Le résultat a été tronqué. La fonction `Get-Svc` a effectué une requête WMI afin de collecter uniquement les services en cours d'exécution. Pour collecter uniquement les services arrêtés, il aurait fallu indiquer la valeur `Stopped` en lieu et place de la valeur `Running`.

Comme les besoins évoluent très vite, la fonction `Get-Svc` pourra se complexifier au gré des besoins des administrateurs système.

10

Les classes et objets

Un problème particulier peut être résolu de plusieurs manières. Parmi les orientations paradigmatiques possibles se trouve le modèle objet ; il s'agit d'une méthode de résolution de problèmes à partir d'une vision des éléments sous forme d'entités ou de structures appelées classes. *Une classe est une structure représentant une entité particulière, qui pourra être incarnée de manières différentes sous forme d'*objets, *ou* instances. *Tous les objets issus d'une classe sont distincts, même s'ils partagent une structure commune provenant de leur classe.*

Voyons dans ce chapitre comment s'implémente le modèle objet à travers nos cinq grands langages. Certains lecteurs seront peut-être étonnés d'apprendre qu'il est possible d'implémenter des classes avec Korn Shell.

KSH

La version `ksh93` de Korn Shell offre la possibilité aux programmeurs de développer à partir du paradigme objet. Cette perspective, peu connue des informaticiens, mérite que nous nous y attardions quelque peu, car il s'agit là d'une fonctionnalité très intéressante.

Créer une classe

Il existe plusieurs manières de définir une classe en Korn Shell. Parmi elles, on relève la commande `typeset` (1) que nous avons déjà croisée dans un précédent chapitre.

Déclaration d'une classe en KSH

```
typeset (1) -T (2) Name_t=( (5)

    typeset -h 'First argument.' Property='value' (3)

    Method() (4) {
  (6) _.a=$1
    }

) (5)
```

La commande `typeset`, à l'aide de l'option `-T`, sert à déclarer une classe (2). Par convention, le nom d'une classe est suivi des caractères `_t`. Le code encapsulé par une classe est délimité par des parenthèses (5). L'instruction `typeset` sert aussi à la création de propriétés (3) ; son option `-h` est utile pour documenter ces dernières. Les méthodes de classe (4) peuvent être implémentées de la même façon que des fonctions.

Le symbole `_` (6) est une référence à la classe elle-même : lorsqu'il s'agit de modifier ses propres éléments comme les propriétés, l'objet créé fait ainsi référence à lui-même.

Instancier et utiliser une classe

Créer une classe avec le langage Korn Shell n'est pas très compliqué. Dans cette section, nous allons en créer une, que nous nommerons `Math_t` et qui réalisera de simples additions et multiplications à partir de deux nombres passés en arguments. Commençons par lui affecter deux propriétés `a` et `b` (1), respectivement de valeur `5` et `7`, ce qui signifie que la classe `Math_t` est utilisable dès son instanciation. Ces deux valeurs doivent être modifiables, ce que réalise la méthode `SetArgs()` (2) grâce aux deux arguments `$1` et `$2`. À présent, il faut donner la possibilité à la classe `Math_t` d'additionner (3) et de multiplier (4) les valeurs `a` et `b` :

```
typeset -T Math_t=()

    typeset -i -h 'First argument.' a=5 (1)
    typeset -i -h 'Second argument.' b=7 (1)

    SetArgs() { (2)
        _.a=$1; _.b=$2
    }

    Add() { (3)
        printf "%d\n" $((_.a +_.b))
    }

    Mul() { (4)
        printf "%d\n" $((_.a *_.b))
    }

)
```

Notre classe est prête ; elle peut donc être instanciée :

```
Math_t obj
```

L'instanciation d'une classe se réalise à l'aide du nom de la classe suivi du nom de l'objet (ici, `obj`). L'instruction suivante affiche les valeurs des propriétés `a` et `b` de l'objet `obj` :

```
echo "The values of a and b are -> ${obj.a} and ${obj.b}."

>>The values of a and b are -> 5 and 7.
```

Les méthodes `Add()` et `Mul()` effectuent respectivement l'addition et la multiplication de ces valeurs :

```
obj.Add

>>12

obj.Mul

>>35
```

Si nous voulons modifier les propriétés `a` et `b`, nous devons fournir les nouvelles valeurs à la méthode `SetArgs()` :

```
obj.SetArgs 7 10
```

Et nous nous apercevrons que les valeurs en question ont bien été modifiées :

```
echo "The values of a and b are -> ${obj.a} and ${obj.b}."

>>The values of a and b are -> 7 and 10.
```

Korn Shell, de manière automatique, documente la classe créée :

```
$ Math_t -man

NAME
  Math_t - set the type of variables to Math_t

SYNOPSIS
  Math_t [ options ] [name[=value]...]

DESCRIPTION
  Math_t sets the type on each of the variables specified by name to Math_t. If =value
is specified, the variable name is set to value before the variable is converted to
Math_t.
```

```
  If no names are specified then the names and values of all variables of this type
are written to standard output.

  Math_t is built-in to the shell as a declaration command so that field splitting and
pathname expansion are not performed on the arguments. Tilde expansion occurs on
value.

OPTIONS
  -r       Enables readonly. Once enabled, the value cannot be changed or
unset.
  -a[type] Indexed array. Each name will converted to an index array of       type
Math_t. If a variable already exists, the current value will become index 0. If
[type] is specified, each subscript is interpreted as a                   value of
enumeration type type. The option value may be omitted.
  -A       Associative array. Each name will converted to an associate array of type
Math_t. If a variable already exists, the current value will become subscript 0.
  -h string Used within a type definition to provide a help string for variable name.
Otherwise, it is ignored.
  -S         Used with a type definition to indicate that the variable is shared by
each instance of the type. When used inside a function defined
with the function reserved word, the specified variables will have
function static scope. Otherwise, the variable is unset prior to
processing the assignment list.

DETAILS
  Math_t defines the following fields:
    _     string.
    a     integer, default value is 5.
    b     integer, default value is 7. First argument..

  Math_t defines the following discipline functions:
    SetArgs
    Add
    Mul

EXIT STATUS
    0     Successful completion.
    >0    An error occurred.

SEE ALSO
   readonly(1), typeset(1)

IMPLEMENTATION
  version         type (AT&T Labs Research) 2008-07-01
  author          David Korn <dgk@research.att.com>
  copyright       Copyright (c) 1982-2012 AT&T Intellectual Property
  license         http://www.eclipse.org/org/documents/epl-v10.html
```

Cette fonctionnalité est très utile lors de l'analyse d'une classe donnée (méthodes, propriétés, etc.).

Perl

La programmation orientée objet n'est pas une perspective native du langage Perl. En effet, son implémentation a pris forme uniquement à partir de la version 5, en 1994.

Créer une classe

En Perl, une classe est un paquet (ou *package*) contenant des propriétés et méthodes. Voici une syntaxe classique de création d'une classe.

Syntaxe de création d'une classe en Perl

```
package (1) name;

sub new (2)
{
    my $class = shift; (3)
    my $self (4) = {
        _FirstArg => shift,
        _SecondArg => shift,
    };
    bless $self, $class; (5)
    return $self; (6)
}
```

Le mot-clé `package` (1) sert ici à définir une classe. Pour que celle-ci puisse être instanciée, un constructeur (2) doit exister au sein de la structure, ce qui est, entre autres, le rôle du mot-clé `sub`. Au sein du constructeur, deux éléments doivent être liés : la classe elle-même (3), ainsi que la référence à un tableau anonyme ou à un dictionnaire anonyme (4). Cette liaison permet de marquer une structure anonyme (tableau ou dictionnaire) comme faisant partie intégrante de la classe déclarée.

C'est la fonction `bless()` (5) qui réalise cette liaison. Enfin, l'instruction `return` (6) renvoie la référence créée, celle-ci appartenant à la classe définie.

Instancier et utiliser une classe

Dans le même esprit que l'exemple pour Korn Shell, voici une classe effectuant des opérations arithmétiques de base :

```
package Math;
sub new
{
    my $class = shift;
    my $self = {
        _a => shift,
        _b => shift,
```

```
    };
    bless $self, $class;
    return $self;
}

sub Add {
      my( $self ) = @_;
      return $self->{_a} + $self->{_b};
}

sub Mul {
      my( $self ) = @_;
      return $self->{_a} * $self->{_b};
}
```

Cette classe, nommée `Math`, contient deux propriétés `a` et `b`, qui sont initialisées lors de l'instanciation à l'aide du constructeur `new`. Puis deux méthodes `Add()` et `Mul()` effectuent respectivement des additions et des multiplications à partir des variables `a` et `b`.

Instancions notre classe et vérifions si l'objet créé fonctionne comme attendu :

```
$object = new Math(7, 10);

print "The addition result is -> ", $object->Add(), "\n";

>>The addition result is -> 17

print "The multiplication result is -> ", $object->Mul(), "\n";

>>The multiplication result is -> 70
```

Parmi les méthodes d'instanciation existantes, il y a celle utilisant la fonction `new()` qui nous a servi pour créer l'objet `$object`. Deux valeurs (`7` et `10`) ont été passées en arguments à la classe `Math`, ce qui autorise un fonctionnement normal des méthodes `Add()` et `Mul()`, parfaitement interprétées par la fonction `print()`. Pour instancier une méthode, l'opérateur de déréférencement (`->`) doit être ajouté à l'objet créé et précéder la méthode appelée.

HISTORIQUE Perl et le paradigme objet

Le paradigme objet est une couche superposée à ce langage ; sur un plan architectural, Perl n'a pas été conçu, à l'origine, pour fournir cette perspective-là. En conséquence, certaines personnes habituées au modèle objet pourraient y trouver une philosophie autre que celle qu'ils ont eu l'habitude de pratiquer jusque-là.

Python

Contrairement à Perl, l'architecture de Python a comme noyau central le paradigme objet ; ceci a comme effet une intégration plus naturelle de cette philosophie par les programmeurs.

Créer une classe

Le mot-clé utilisé pour créer une classe en Python est `class` (1).

Syntaxe d'une classe en langage Python

```
class (1) Name:
   (2) def __init__(self (3), value):
          self._val = value (4)
```

La définition du constructeur de classe est établie à l'aide du mot-clé `def` (2). `__init__` est une méthode spéciale permettant la construction d'un objet. Elle prend pour premier argument le mot-clé `self` (3) (qui est en réalité une convention) afin de référencer l'objet en tant que tel et donc accéder à ses membres. Naturellement, lors de l'instanciation d'une classe, des valeurs (4) peuvent être initialisées et constituer les données de base de l'objet créé.

Instancier et utiliser une classe

Dans cette section, nous allons définir une classe composée de deux propriétés et de deux méthodes :

```
class Dev:
    def __init__(self, value, color = 'Blue'):
        self._val = value
        self._color = color

    def get_color(self):
        return self._color

    def set_color(self, color):
        self._color = color
```

Nous pouvons distinguer au sein de la classe `Dev` son constructeur (`__init__`), deux attributs (`_val` et `_color`) et deux méthodes (`get_color()` et `set_color()`). L'argument `color` a une valeur par défaut (`'Blue'`), qui s'applique lorsqu'aucune autre couleur n'est spécifiée lors de l'instanciation. Les arguments `color` et `value` ne doivent pas être amalgamés avec les attributs `_color` et `_val`.

La méthode `get_color` retourne la valeur de l'attribut `_color`, tandis que `set_color` modifie ce même attribut. L'instanciation de la classe `Dev` est possible à l'aide de la ligne de code suivante :

```
obj = Dev(3, 'Black')
```

L'objet `obj` est créé avec les arguments `3` et `'Black'`. Obtenons la valeur de l'attribut `_color` :

```
print("The color is", obj.get_color())

>>The color is Black
```

La valeur de l'attribut `_color` est bien celle spécifiée lors de la construction de l'objet.

Modifions cette valeur et vérifions si le changement a bien eu lieu :

```
obj.set_color('Green')
print("The color is", obj.get_color())

>>The color is Green
```

La fonction `print()` a affiché la valeur `Green` au lieu de la valeur `Black`. L'objet `obj` se comporte donc exactement comme prévu. Évidemment, il s'agit là d'un exemple pédagogique, mais il peut constituer une excellente base de création de classes plus en adéquation avec les besoins du programmeur.

VBS

Le paradoxe du langage VBS, c'est que ses utilisateurs ne l'exploitent généralement pas comme un véritable langage orienté objet. La possibilité de créer des classes est une perspective très peu connue de ceux qui programment en Visual Basic Scripting.

Créer une classe

Comme en Python, le mot-clé utilisé pour créer une classe en VBS est `Class` (1).

Syntaxe d'une classe en VBS

```
Class (1) Name
   (2) Properties Public/Private
   (3) Methods Public/Private
End Class (4)
```

Pour avoir une véritable utilité, une classe doit avoir au minimum des membres tels que des propriétés ou méthodes (2) (3). Ces derniers sont parfois privés (`Private`), c'est-à-dire uniquement accessibles depuis la classe elle-même. Dans certains cas, ils sont au contraire accessibles depuis d'autres classes ; ils sont alors dits publics (`Public`). Le mot-clé `End Class` (4) termine la déclaration d'une classe.

Instancier et utiliser une classe

L'exemple suivant définit une classe interrogeant le service WMI. L'objectif est de lister les services en cours d'exécution et/ou arrêtés.

```
Class SvcState
   Private p_ServiceState
   Private p_ServerName
```

```
  Private Sub Class_Initialize
    p_ServiceState = "Running"
    p_ServerName = "."
  End Sub

  Public Property Get ServiceState
    ServiceState = p_ServiceState
  End Property

  Public Property Get ServerName
    ServerName = p_ServerName
  End Property

  Public Property Let ServerName(name)
    p_ServerName = state
  End Property

  Public Property Let ServiceState(state)
    p_ServiceState = state
  End Property

  Public Sub GetServices
      Set objWMIService = GetObject("winmgmts:" _
         & "{impersonationLevel=impersonate}!\\" & p_ServerName & "\root\cimv2")

   Select Case p_ServiceState
    Case "Running"    Set colRunningServices = objWMIService.ExecQuery("Select * from
Win32_Service where State = 'Running'")
    Case "Stopped"    Set colRunningServices = objWMIService.ExecQuery("Select * from
Win32_Service where State = 'Stopped'")
   End Select

    For Each objService in colRunningServices
       Wscript.Echo objService.DisplayName  & VbTab & p_ServiceState
    Next
  End Sub
End Class
```

La classe `SvcState` est composée de deux attributss (`p_ServiceState` et `p_ServerName`) accessibles via les propriétés `ServiceState` et `ServerName`. Les mots-clés `Get` et `Let` des propriétés déclarées permettent respectivement d'obtenir ou de modifier la valeur d'un attribut. L'attribut `p_ServiceState` contient la valeur d'état (essentiellement `Running` ou `Stopped`) des services et l'attribut `p_ServerName` comporte le nom d'une machine locale ou distante sur laquelle la collecte des services va avoir lieu.

Le constructeur de classe `Class_Initialize` initialise par défaut les deux attributs susmentionnés. La méthode `GetServices` procède à une connexion au service WMI, vérifie la valeur de l'attribut `p_ServiceState` et obtient la liste des services en fonction de l'état indiqué.

Le mot-clé `New`, à l'aide de l'argument `SvcState`, crée quant à lui un nouvel objet en mémoire, comme suit :

```
Dim obj
Set obj = New SvcState
```

L'objet `obj` est d'emblée utilisable :

```
obj.GetServices

>>Générateur de points de terminaison du service Audio Windows Running
>>Audio Windows Running
>>Moteur de filtrage de base    Running
>>Infrastructure du service Broker      Running
>>Explorateur d'ordinateurs     Running
>>Search Protect by Conduit Service     Running
>>Services de chiffrement       Running
>>Lanceur de processus serveur DCOM     Running
>>Service d'association de périphérique Running
>>Client DHCP   Running
>>Client DNS    Running
>>Service de stratégie de diagnostic    Running
>>Journal d'événements Windows Running
>>Système d'événement COM+      Running
>>Hôte du fournisseur de découverte de fonctions        Running
>>Publication des ressources de découverte de fonctions Running
>>Service de cache de police Windows    Running
>>Fournisseur HomeGroup Running
>>Assistance IP Running
>>Serveur       Running
...
```

La sortie (tronquée ici) affiche bien les services en cours d'exécution, ce qui correspond à la valeur par défaut de l'attribut `p_ServiceState`. Pour que le test de l'objet `obj` soit complet, modifions la valeur de la propriété `ServiceState` :

```
obj.ServiceState = "Stopped"
```

Appelons une nouvelle fois la méthode `GetServices` :

```
obj.GetServices

>>Expérience d'application      Stopped
>>Service de la passerelle de la couche Application     Stopped
>>Agent d'installation pour tous les utilisateurs Windows       Stopped
>>Identité de l'application     Stopped
```

```
>>Informations d'application    Stopped
>>Gestion d'applications        Stopped
>>Alcohol Virtual Drive Auto-mount Service      Stopped
>>Programme d'installation ActiveX (AxInstSV)   Stopped
>>Service de chiffrement de lecteur BitLocker   Stopped
>>Service de transfert intelligent en arrière-plan      Stopped
>>Service de prise en charge Bluetooth Stopped
>>Propagation du certificat     Stopped
>>Application système COM+      Stopped
>>Fichiers hors connexion       Stopped
>>Optimiser les lecteurs        Stopped
>>Service d'installation de périphérique        Stopped
...
```

La liste de services affichée (tronquée) correspond à la nouvelle valeur (`Stopped`) de la propriété `ServiceState`. L'objet `obj` se comporte donc exactement comme prévu. D'autres fonctionnalités pourront être ajoutées à la classe `SvcState` en fonction des besoins des programmeurs. Il faut cependant être attentif à chaque étape de développement d'une classe, sous peine de s'y perdre d'un point de vue logique.

Windows PowerShell

PowerShell ne propose pas (sauf à passer par d'autres langages .NET), à partir de l'arsenal dont il dispose, de créer des classes au sens classique du terme. Cependant, étant basé sur le paradigme objet, il offre la perspective de créer et de structurer des objets dits vierges à la base.

Créer une classe

Il existe une classe permettant aux développeurs de créer des objets vierges qui ne demandent qu'à être enrichis : elle se nomme `PSCustomObject`. Cette classe constitue la base à partir de laquelle des membres comme des propriétés ou des méthodes vont, en fonction des besoins, être intégrés. La cmdlet `New-Object` (1) est une commande PowerShell qui sert à créer des objets à partir notamment de classes .NET.

Créer un objet vierge avec PowerShell

```
New-Object (1) -TypeName (2) 'PSCustomObject' (3)
```

Le paramètre `-TypeName` (2) spécifie le nom de la classe (3) .NET ou COM servant à la construction de l'objet. Et c'est tout ! Il n'y a pas besoin d'avoir des compétences en programmation orientée objet pour créer des objets en PowerShell.

Instancier et utiliser une classe

Commençons, dans cette section, par créer un objet vierge que nous nommerons `$obj` :

```
PS> $obj = New-Object -TypeName 'PSCustomObject'
```

Cet objet est véritablement vierge dans sa structure, comme le montre la ligne de code suivante :

```
PS> $obj | Get-Member

>>   TypeName : System.Management.Automation.PSCustomObject

>>Name        MemberType Definition
>>----        ---------- ----------
>>Equals      Method     bool Equals(System.Object obj)
>>GetHashCode Method     int GetHashCode()
>>GetType     Method     type GetType()
>>ToString    Method     string ToString()
```

Les membres listés n'appartiennent pas en propre à l'objet `$obj`, mais sont hérités. Pour ajouter des membres, utilisons la cmdlet `Add-Member` :

```
PS> $obj | Add-Member -MemberType NoteProperty -Name 'Fval' -Value 5
PS> $obj | Add-Member -MemberType NoteProperty -Name 'Sval' -Value 7
```

Deux propriétés ont été ajoutées : `Fval` à la valeur `5` et `Sval` à la valeur `7`. Une nouvelle consultation des membres de `$obj` indique bien que les propriétés `Fval` et `Sval` ont correctement été ajoutées :

```
PS> $obj | Get-Member

>>   TypeName : System.Management.Automation.PSCustomObject

>>Name        MemberType   Definition
>>----        ----------   ----------
>>Equals      Method       bool Equals(System.Object obj)
>>GetHashCode Method       int GetHashCode()
>>GetType     Method       type GetType()
>>ToString    Method       string ToString()
>>Fval        NoteProperty System.Int32 Fval=5
>>Sval        NoteProperty System.Int32 Sval=7
```

Dotons maintenant `$obj` d'une méthode additionnant les deux propriétés :

```
PS> Add-Member -InputObject $obj -MemberType ScriptMethod -Name Add -Value {Write-
Output ($This.Fval + $This.Sval)}
```

Un bloc de script a été lié à la méthode `Add()` et sera exécuté à chaque appel à cette dernière :

```
PS> $obj.Add()

>>12
```

La variable `$This` représente l'objet `$obj` ; c'est une forme d'abstraction ou une référence à l'objet lui-même.

Créer un objet avec PowerShell est une démarche beaucoup plus aisée qu'avec les autres langages de script étudiés dans cet ouvrage. Il s'agit donc d'une facilité qu'il est intéressant d'exploiter.

11

Les modules

*La possibilité de créer des extensions est un horizon que tout langage de script doit pouvoir ouvrir. Certaines extensions (ou modules) sont développées par des éditeurs tiers. D'autres sont créées par des développeurs pouvant appartenir à certaines communautés formées autour d'un langage particulier. Créer un module ou une extension est une voie d'accès à plus de liberté, mais aussi à plus de créativité. Toutefois, tous les langages de script n'ont pas ouvert cette voie ; c'est le cas de Visual Basic Scripting, même s'il propose l'utilisation d'objets COM (*Component Object Model*) provenant de bibliothèques tierces. Ce chapitre va comparer nos cinq langages dans les processus de création et d'utilisation de modules.*

KSH

Korn Shell n'a pas à proprement parler de système de création de modules. Cependant, à travers un autre mécanisme, il est possible de définir des bibliothèques de fonctions et de les importer pour les exploiter.

Créer un module

KSH dispose d'une variable d'environnement, FPATH (1), contenant le chemin d'accès (2) aux bibliothèques utilisables dans un script.

Définition de la variable FPATH

```
1 FPATH=/chemin/vers/modules 2
```

Les bibliothèques doivent être entreposées dans le répertoire pointé par la variable FPATH, sous peine de rencontrer des erreurs d'exécution.

> **IMPORTANT Les variables PATH et FPATH**
>
> Il ne faut pas confondre les variables PATH et FPATH, la première contenant les chemins d'accès aux exécutables.

Importer et utiliser un module

La fonction suivante analyse le fichier /etc/passwd afin d'en extraire, pour chaque nom de compte, le répertoire personnel qui lui est associé :

```
module() {
  awk -F':' '{ print "The home directory of " $1 " is " $6 }' /etc/passwd
}
```

Au sein de cette fonction, la commande awk est invoquée ; celle-ci est en effet très puissante dans l'analyse de texte. En sortie, la fonction module() retourne le répertoire personnel de chaque compte, et ce, à partir du fichier /etc/passwd dont les informations sont importantes. Copions la fonction module() dans un fichier qui portera le même nom, puis déposons ce fichier dans un répertoire que nous nommerons modules au sein du répertoire personnel du compte utilisé (par exemple, /home/kais/modules).

Définissons la variable FPATH :

```
FPATH=/home/kais/modules
```

Ajoutons-la au sein des variables d'environnement visibles dans l'ensemble des portées à l'aide de la commande export :

```
export FPATH
```

À présent, positionnons-nous par exemple dans le répertoire /tmp et créons un script que nous nommerons main.ksh. Dans ce script, nous ne ferons qu'appeler la fonction module() créée précédemment :

```
#!/usr/bin/ksh
module
```

À l'issue de l'exécution du script `main.ksh`, nous devrions avoir en sortie le résultat de l'analyse du fichier `/etc/passwd` :

```
scripting@debiandev:~$ ./main.ksh

>>The home directory of root is /root
>>The home directory of daemon is /usr/sbin
>>The home directory of bin is /bin
>>The home directory of sys is /dev
>>The home directory of sync is /bin
>>The home directory of games is /usr/games
>>The home directory of man is /var/cache/man
>>The home directory of lp is /var/spool/lpd
>>The home directory of mail is /var/mail
>>The home directory of news is /var/spool/news
>>The home directory of uucp is /var/spool/uucp
>>The home directory of proxy is /bin
>>The home directory of www-data is /var/www
>>The home directory of backup is /var/backups
>>The home directory of list is /var/list
>>The home directory of irc is /var/run/ircd
>>The home directory of gnats is /var/lib/gnats
>>The home directory of nobody is /nonexistent
>>The home directory of libuuid is /var/lib/libuuid
>>The home directory of messagebus is /var/run/dbus
>>The home directory of colord is /var/lib/colord
>>The home directory of usbmux is /home/usbmux
>>The home directory of Debian-exim is /var/spool/exim4
>>The home directory of statd is /var/lib/nfs
>>The home directory of avahi is /var/run/avahi-daemon
>>The home directory of pulse is /var/run/pulse
>>The home directory of speech-dispatcher is /var/run/speech-dispatcher
>>The home directory of hplip is /var/run/hplip
>>The home directory of sshd is /var/run/sshd
>>The home directory of rtkit is /proc
>>The home directory of saned is /home/saned
>>The home directory of Debian-gdm is /var/lib/gdm3
>>The home directory of scripting is /home/scripting
```

L'importation et l'exécution de la fonction `module()` ont donc bien eu lieu. À l'avenir, et sur la base de ce principe, toutes les fonctions fréquemment utilisées pourront être instanciées sans être perpétuellement redéfinies.

Perl

Contrairement à Korn Shell, Perl dispose d'un véritable modèle de création de modules écrits.

Créer un module

En Perl, un module (1) est un fichier portant l'extension `.pm` (2) et qui peut contenir des fonctions, des classes, ou tout morceau de code susceptible d'être utilisé plusieurs fois.

Un module doit avoir l'extension .pm pour être considéré comme tel par Perl

```
(1) /chemin/vers/modules/module.pm (2)
```

Là aussi, il est préférable de centraliser la localisation des modules écrits. La variable `@INC` contient l'ensemble des chemins d'accès aux différentes bibliothèques. Il vaut donc mieux placer les modules au sein de ces emplacements. Si ce n'est pas le cas, il faudra alors préciser dans le script d'appel des bibliothèques la localisation des modules invoqués.

Importer et utiliser un module

Voici un module fournissant des statistiques sur les partitions actuellement montées d'un système Unix/Linux :

```
use strict;
use warnings;

package Disk::Spec;

our $VERSION = "1.0";

sub new {
    my $class = shift;
    my $self = {};
    bless($self, $class);
    return $self;
}

sub Stats {
    my ($self, $part) = @_;
    my @stat = `df -h $part | awk 'NR==2 {print "Partition->" \$1 "\\nSize->" \$2
"\\nUsed->" \$3}'`;
    print(@stat);
}
```

Nous placerons ce module dans un fichier `Disk.pm`. Il contient une classe `Spec`, qui elle-même comporte une méthode `Stats()`. À partir d'un argument `$part`, cette dernière exécute une commande shell `df` pour obtenir des informations comme le nom de la partition, sa taille ainsi que l'espace utilisé en son sein. Les informations générées par la commande `df` sont traitées par la commande `awk` dans le but de les filtrer et de les disposer d'une certaine façon.

Le module `Disk.pm` doit ensuite être déposé, soit dans un des répertoires contenus dans la variable `@INC`, soit dans un autre répertoire qui sera indiqué dans le script d'appel. Nous allons

ici le déposer dans un répertoire non conventionnel (/home/scripting/Perl/Modules) et préciserons donc le chemin d'accès dans script.pl :

```
#!/usr/bin/perl

use strict;
use warnings;

use lib '/home/scripting/Perl/Modules';
use Disk;

my $Arg = shift || return;
my $obj = Disk::Spec->new();
$obj->Stats($Arg);

print("Done.\n");
```

Le répertoire des modules est indiqué à l'aide des mots-clés use lib, puis la ligne de code use Disk autorise l'importation et l'utilisation de notre module. La variable $Arg permet au script script.pl de recevoir un argument (nom de partition ou nom de point de montage) qui sera redirigé vers la méthode Stats() de l'objet $obj. Il ne nous reste plus qu'à exécuter le script script.pl et à observer le résultat :

```
scripting@debiandev:~$ ./script.pl /home

Partition->/dev/sda9
Size->7.8G
Used->754M
Done.
```

Ici, le script script.pl a été exécuté avec le paramètre /home qui représente un des points de montage. Essayons de le réexécuter, mais avec cette fois-ci un autre argument :

```
scripting@debiandev:~$ ./script.pl /var

Partition->/dev/sda6
Size->2.5G
Used->699M
Done.
```

L'argument /var a été spécifié et Perl retourne des informations claires sur la partition (/dev/sda6) liée au point de montage indiqué, sa taille et l'espace qui y est utilisé.

Le module Disk.pm fonctionne donc correctement. Il pourra, à l'avenir, être enrichi d'autres fonctionnalités ou même structuré d'une autre façon, car il existe dans Perl des dizaines de possibilités quant à la résolution d'un seul problème.

Python

Python propose un modèle de création de modules similaire à celui de Perl sur certains aspects et différent sur d'autres.

Créer un module

Python n'établit pas de distinction entre un script dit classique et un module particulier.

Un module Python pourrait parfaitement être confondu avec un script classique

```
/chemin/vers/modules/module.py (1)
```

En effet, l'extension d'un module est la même que celle d'un script traditionnel : `.py` (1). Un module peut comporter des fonctions, des classes, ou tout autre élément de langage. Définir un module Python consiste donc à placer au sein d'un script tout code susceptible d'être utilisé plusieurs fois.

Importer et utiliser un module

Reprenons, cette fois en Python, l'idée d'une fonction capable d'analyser le fichier `/etc/passwd` afin d'en extraire, pour chaque compte, le répertoire personnel associé :

```
def GetPassInfos():
   for line in file('/etc/passwd'):
      fields = line.split(':')
      print "The home directory of {} is {}".format(fields[0], fields[5])
```

Copions cette fonction dans un fichier que nous nommerons `module.py` et que nous placerons dans le même répertoire que le script d'appel `script.py` :

```
#!/usr/bin/python

import module
module.GetPassInfos()
```

Dans le script `script.py`, le module `module.py` est appelé à l'aide de la ligne `import module`, puis la fonction `GetPassInfos()` est appelée sans que sa définition n'existe dans le script d'appel. Pour les modules écrits de cette façon, il vaut mieux faire précéder le nom de la fonction par celui du module `Module.Fonction()`, même si celui-ci est clairement importé en début de script.

> ATTENTION **Référencement des modules en Python**
>
> Si les modules sont écrits et déposés dans des lieux non conventionnels, il est important de bien indiquer leur chemin d'accès (voir la documentation).

Exécutons `script.py` et observons le résultat affiché :

```
scripting@debiandev:~$ ./script.py

>>The home directory of root is /root
>>The home directory of daemon is /usr/sbin
>>The home directory of bin is /bin
>>The home directory of sys is /dev
>>The home directory of sync is /bin
>>The home directory of games is /usr/games
>>The home directory of man is /var/cache/man
>>The home directory of lp is /var/spool/lpd
>>The home directory of mail is /var/mail
>>The home directory of news is /var/spool/news
>>The home directory of uucp is /var/spool/uucp
>>The home directory of proxy is /bin
>>The home directory of www-data is /var/www
>>The home directory of backup is /var/backups
>>The home directory of list is /var/list
>>The home directory of irc is /var/run/ircd
>>The home directory of gnats is /var/lib/gnats
>>The home directory of nobody is /nonexistent
>>The home directory of libuuid is /var/lib/libuuid
>>The home directory of messagebus is /var/run/dbus
>>The home directory of colord is /var/lib/colord
>>The home directory of usbmux is /home/usbmux
>>The home directory of Debian-exim is /var/spool/exim4
>>The home directory of statd is /var/lib/nfs
>>The home directory of avahi is /var/run/avahi-daemon
>>The home directory of pulse is /var/run/pulse
>>The home directory of speech-dispatcher is /var/run/speech-dispatcher
>>The home directory of hplip is /var/run/hplip
>>The home directory of sshd is /var/run/sshd
>>The home directory of rtkit is /proc
>>The home directory of saned is /home/saned
>>The home directory of Debian-gdm is /var/lib/gdm3
>>The home directory of scripting is /home/scripting
```

Comme prévu, l'appel du module `module.py` a bien fonctionné. Ce module pourra donc être enrichi avec d'autres fonctions et être de nouveau appelé à partir d'autres scripts.

Windows PowerShell

En PowerShell, la gestion des modules est caractéristique d'une certaine richesse. Dans cette section, nous nous limiterons à la simple création et à l'utilisation de base des modules.

Créer un module

Un module Windows PowerShell est un script portant l'extension `.psm1` (1). Il peut contenir tout type de commandes ou éléments de langage.

Un module PowerShell porte l'extension .psm1

```
/chemin/vers/modules/module.psm1 (1)
```

Les modules écrits peuvent être déposés dans n'importe quel endroit du système d'exploitation, mais il est préférable de les placer dans ceux prévus à cet effet (par exemple, `$home\Documents\WindowsPowerShell\Modules`) de sorte que PowerShell les reconnaisse plus facilement.

Importer et utiliser un module

Le module suivant, nommé `FolderStats.psm1`, contient une fonction `Get-DirStats()` :

```
Function Get-DirStats {

Param(
   [String]$Directory
)

Get-ChildItem $Directory -Directory |
Select FullName,LastWriteTime,@{
   Name="Size";Expression={
   (Get-ChildItem $_.Fullname -recurse | Measure-Object Length -sum).sum
   }
}
}
```

Cette fonction, à partir d'un répertoire donné (`-Directory`), analyse chacun de ses sous-répertoires afin d'en extraire le chemin d'accès complet, l'heure de la dernière écriture ainsi que la taille (calculée sur la base d'une propriété personnalisée). Son principe est donc très simple.

Ici, le module `FolderStats.psm1` est déposé dans le répertoire `C:\scripts\PowerShell\Modules`. La cmdlet `Import-Module` nous aidera à l'importer dans une session courante :

```
PS> Import-Module C:\scripts\PowerShell\Modules\FolderStats.psm1
```

Si l'importation est réussie, alors la cmdlet `Import-Module` ne devrait normalement pas afficher de sortie. À présent, il ne nous reste plus qu'à appeler la fonction `Get-DirStats()` en passant en paramètre un nom de répertoire (`C:\Windows`).

```
PS> Get-DirStats -Directory 'C:\Windows' | Format-List

>>FullName      : C:\Windows\addins
>>LastWriteTime : 18/02/2012 09:21:56
>>Size          : 802

>>FullName      : C:\Windows\AppCompat
>>LastWriteTime : 18/02/2012 09:21:48
>>Size          :

>>FullName      : C:\Windows\apppatch
>>LastWriteTime : 18/02/2012 11:46:36
>>Size          : 12357426

>>FullName      : C:\Windows\assembly
>>LastWriteTime : 18/02/2012 09:21:48
>>Size          : 237874774

>>FullName      : C:\Windows\AUInstallAgent
>>LastWriteTime : 18/02/2012 09:21:49
>>Size          :

>>FullName      : C:\Windows\Boot
>>LastWriteTime : 18/02/2012 09:21:49
>>Size          : 34510076

>>FullName      : C:\Windows\Branding
>>LastWriteTime : 18/02/2012 09:21:49
>>Size          : 2392552

>>FullName      : C:\Windows\CbsTemp
>>LastWriteTime : 19/02/2014 08:17:56
>>Size          :

>>FullName      : C:\Windows\CSC
>>LastWriteTime : 10/01/2014 16:37:46
>>Size          :
...
```

La fonction `Get-DirStats()` est une commande extrêmement utile, car elle renseigne sur la taille des sous-répertoires d'un répertoire particulier, et ce, de manière récursive. Elle pourra donc constituer un excellent début d'une série de fonctions couvrant des objectifs similaires et formant le corpus du module `FolderStats.psm1`.

12

La gestion d'erreurs

Gérer les erreurs susceptibles de se produire lors de l'exécution d'un code quelconque est une tâche bien connue des développeurs. Le plus souvent, les administrateurs ne connaissent pas du tout cette partie du processus de développement ou alors se limitent à une compréhension de surface leur permettant d'atteindre leurs objectifs.

D'une manière générale, la gestion d'erreurs est une démarche fortement détestée des informaticiens, car elle demande du temps ainsi que de la patience. Bien souvent, on se contente de l'observation stricte des résultats, même si le code écrit pêche en qualité !

Ce chapitre présente les éléments à notre disposition dans les langages de script pour la gestion des erreurs, inévitable si on souhaite obtenir un code bien écrit et plus facile à maintenir.

KSH

En matière de gestion des erreurs, Korn Shell ne fournit pas de possibilités très sophistiquées. Cependant, la technique proposée est à la fois très simple et très efficace.

La variable $?

KSH définit de manière dynamique une variable spéciale en fonction du résultat d'une commande ou d'une instruction donnée. Cette variable se nomme `$?` ; elle prend la valeur `0` si la commande se déroule correctement et une valeur différente en cas d'échec.

Utilisation de la variable $?

```
scripting@debiandev:~$ ls
>>Desktop    Modules   PrototypeEngineProject
>>Documents Music     Public
>>Downloads Pictures Templates

scripting@debiandev:~$ echo $?
>>0

scripting@debiandev:~$ ps
>> PID TTY          TIME CMD
>> 3396 pts/0    00:00:00 bash
>> 3721 pts/0    00:00:00 ksh
>> 3753 pts/0    00:00:00 ps

scripting@debiandev:~$ echo $?
>>0

scripting@debiandev:~$ cls
>>ksh: cls: not found [No such file or directory]

scripting@debiandev:~$ echo $?
>>127
```

Manipuler la variable $?

La variable spéciale `$?` est extrêmement utile en mode shell, comme vu dans la section précédente, mais aussi en mode scripting :

```
#!/usr/bin/ksh

ETC_SHA="/etc/shadow"

cat $ETC_SHA > /dev/null 2>&1

if [[ $? != 0 ]]
then
    print "Reading $ETC_SHA is impossible. Please check the permissions."
    exit 1
else
    cat /etc/shadow
fi
```

Une variable `$ETC_SHA` contient le chemin d'accès vers le fichier `/etc/shadow`. Puis la commande `cat` doit lire ce fichier : si cela se déroule correctement, la valeur de sortie de la variable `$?` est `0` et une lecture des informations est possible. Si la lecture du fichier `/etc/shadow` n'est pas réalisable, pour des raisons qui sont souvent liées aux droits d'accès, le script indique alors la nécessité de vérifier les permissions.

L'exécution de ce script affiche donc les informations recherchées ou une erreur explicitant l'impossibilité de lire le fichier `/etc/shadow`. Commençons par utiliser un compte (`scripting`) n'ayant pas les droits de lecture de ce fichier. La sortie affiche bien l'erreur attendue :

```
scripting@debiandev:~$ ./script.ksh
>>Reading /etc/shadow is impossible. Please check the permissions.
```

Essayons à présent de reproduire l'opération avec le compte `root`. Les informations sont bien affichées :

```
root@debiandev:/home/scripting# ./script.ksh
>>root:$6$KJXLH…./:15762:0:99999:7:::
>>daemon:*:15762:0:99999:7:::
>>bin:*:15762:0:99999:7:::
>>sys:*:15762:0:99999:7:::
>>sync:*:15762:0:99999:7:::
>>games:*:15762:0:99999:7:::
>>man:*:15762:0:99999:7:::
>>lp:*:15762:0:99999:7:::
>>mail:*:15762:0:99999:7:::
>>news:*:15762:0:99999:7:::
>>uucp:*:15762:0:99999:7:::
>>proxy:*:15762:0:99999:7:::
>>www-data:*:15762:0:99999:7:::
>>backup:*:15762:0:99999:7:::
>>list:*:15762:0:99999:7:::
>>irc:*:15762:0:99999:7:::
>>gnats:*:15762:0:99999:7:::
>>nobody:*:15762:0:99999:7:::
>>libuuid:!:15762:0:99999:7:::
>>messagebus:*:15762:0:99999:7:::
>>colord:*:15762:0:99999:7:::
>>usbmux:*:15762:0:99999:7:::
>>Debian-exim:!:15762:0:99999:7:::
>>statd:*:15762:0:99999:7:::
>>avahi:*:15762:0:99999:7:::
>>pulse:*:15762:0:99999:7:::
>>speech-dispatcher:!:15762:0:99999:7:::
>>hplip:*:15762:0:99999:7:::
>>sshd:*:15762:0:99999:7:::
>>rtkit:*:15762:0:99999:7:::
>>saned:*:15762:0:99999:7:::
>>Debian-gdm:*:15762:0:99999:7:::
>>scripting:$6$mS9eVB…/:15762:0:99999:7:::
```

Au cours de l'écriture de scripts Korn Shell, le recours à la variable `$?` sera quasi systématique. Son principe est simple, son utilisation aisée et sa présence au sein de vos scripts sera donc, au fur et à mesure d'une certaine progression, naturelle et incontestée.

Perl

Le langage Perl fournit une multitude de moyens de gérer les erreurs. Dans cette section, nous en verrons deux : les fonctions `die()` et `warn()`.

La fonction die()

La fonction `die()` est très utilisée pour anticiper les erreurs susceptibles de survenir dans un script Perl :

```
#!/usr/bin/perl

chdir('/user/alma') || die("$!");
…
```

Dans cet exemple, la fonction `chdir()` est invoquée au sein d'un script pour notamment permettre un changement de répertoire. Si l'opération échoue, la fonction `die()` affiche une erreur qui peut être anticipée :

```
scripting@debiandev:~$ ./script.pl
No such file or directory at ./script.pl line 3.
```

Comme le répertoire `/user/alma` n'existe pas, Perl a levé une exception et a arrêté l'exécution du script `script.pl`. Il est aussi possible de personnaliser les messages d'erreur dans la fonction `die()`, ce qui la rend très efficace :

```
#!/usr/bin/perl

chdir('/user/alma') || die("Impossible to change the working directory.");

scripting@debiandev:~$ ./script.pl
>>Impossible to change the working directory. at ./script.pl line 3.
```

La fonction warn()

La fonction `warn()` affiche des messages d'avertissement qui ne provoquent pas, contrairement à la fonction `die()`, l'arrêt de l'exécution d'un script :

```
#!/usr/bin/perl

chdir('/user/alma') || warn("Impossible to change the working directory.\n");

print("Continue..\n");

scripting@debiandev:~$ ./script.pl
>>Impossible to change the working directory.
>>Continue..
```

Lorsque les erreurs sont considérées comme moins critiques, la fonction `warn()` constitue un excellent moyen d'afficher des messages d'avertissement ne nécessitant pas l'arrêt d'un script donné.

Python

Python, tout comme Perl, dispose de possibilités très vastes en matière de gestion d'erreurs. Cette section mentionnera une instruction importante de ce dispositif : `try..except`.

L'instruction try..except

Il est souvent nécessaire, dans un script donné, d'exécuter un bloc de code si une exception se produit par rapport à un autre bloc de code. L'instruction est composée de deux mots-clés : `try` (1) permet d'exécuter un bloc de code (3) particulier et, si une exception surgit, le bloc de code (4) lié à la clause `except` (2) est exécuté.

Syntaxe de l'instruction try..except

```
try: (1) {
    <bloc de code> (3)
} except: (2) {
    <bloc de code> (4)
}
```

Pour résumer, si une exception se produit suite à l'exécution d'une instruction donnée, une autre instruction sera exécutée, s'inscrivant par là même dans une logique d'anticipation.

try..except en action

La structure suivante tente d'ouvrir et de lire un fichier :

```
try:
    fh = open('fil.txt')
    for line in fh:
        print(line.strip())
except IOError as e:
    print('Could not open the file..', e)
```

Le nom du fichier en question est mal orthographié (`fil.txt` au lieu de `file.txt`). Lorsque l'instruction `try..except` tente de l'ouvrir et de le lire, elle échoue et le bloc de code lié à la clause `except` est exécuté par Python. À côté du mot-clé `except` est précisée une catégorie d'exception (`IOError`). Cette indication force Python à ne prendre en compte que des exceptions de ce type, c'est-à-dire dans notre cas liées à des mouvements d'entrées/sorties en relation avec le fichier `file.txt`.

La fonction `open()` ne pouvant ouvrir ce fichier à cause d'un nom mal précisé, une exception est automatiquement levée :

```
>>Could not open the file.. [Errno 2] No such file or directory: 'fil.txt'
```

L'erreur affichée à l'écran traduit l'inexistence du fichier `fil.txt` et l'instruction `try..except` a donc bien fonctionné. Ajoutons la lettre manquante :

```
try:
    fh = open('file.txt')
    for line in fh:
        print(line.strip())
except IOError as e:
    print('Could not open the file..', e)
```

Et observons le résultat :

```
>>This is line 1.
>>This is line 2.
>>This is line 3.
>>This is line 4.
>>This is line 5.
>>This is line 6.
>>This is line 7.
>>This is line 8.
>>This is line 9.
>>This is line 10.
```

Là, le fichier a été lu ! Son contenu affiché à l'écran le prouve. Dans cet exemple, il s'agit d'accéder à un fichier, mais l'instruction `try..except` peut gérer tous types d'erreurs.

VBS

Visual Basic Scripting propose une gestion d'erreurs très rudimentaire. L'instruction de référence en la matière se nomme `On Error`.

L'instruction On Error

Le comportement de l'instruction `On Error` est double. Elle permet à la fois d'activer (1) la gestion d'erreurs et de la désactiver (2).

Activer la gestion d'erreurs en VBS

```
On Error Resume Next (1)
```

Désactiver la gestion d'erreurs en VBS

```
On Error GoTo 0 (2)
```

Par défaut, l'interpréteur arrête l'exécution d'un script en cas d'erreur. Il n'est donc pas nécessaire de désactiver la gestion d'erreurs de manière explicite à l'aide de l'instruction `On Error GoTo 0`. Cette dernière pourra en revanche être appelée si l'instruction `On Error Resume Next` est présente dans un script.

On Error en action

Le script suivant liste des informations concernant le système d'exploitation ainsi que le *Service Pack* installés sur une machine locale ou distante :

```
On Error Resume Next (3)

strComputer = "notexist" (1)
Set objWMIService = GetObject("winmgmts:" _
 & "{impersonationLevel=impersonate}!\\" & strComputer & "\root\cimv2") (2)

If Err.Number <> 0 Then (4)
   WScript.Echo Err.Number & " Source: " & Err.Source & " Description: " &  Err.Descri
ption
   Err.Clear
End If

Set colOSes = objWMIService.ExecQuery("Select * from Win32_OperatingSystem")
For Each objOS in colOSes
  Wscript.Echo "Computer Name: " & objOS.CSName
  Wscript.Echo "Caption: " & objOS.Caption
  Wscript.Echo "Version: " & objOS.Version
  Wscript.Echo "Build Number: " & objOS.BuildNumber
  Wscript.Echo "Build Type: " & objOS.BuildType
  Wscript.Echo "OS Type: " & objOS.OSType
  Wscript.Echo "Other Type Description: " & objOS.OtherTypeDescription
  WScript.Echo "Service Pack: " & objOS.ServicePackMajorVersion & "."  _
   objOSServicePackMinorVersion
Next
```

La variable `strComputer` contient un nom de serveur, `notexist` (1). Le script doit se connecter au service WMI (2) de ce serveur. La première ligne du script est une activation explicite (3) de la gestion d'erreurs, ce qui nous permettra de traiter, le cas échéant, les erreurs susceptibles d'être rencontrées (4).

Si une erreur se produit suite à l'exécution du script, un objet nommé `Err` est automatiquement créé, utilisable pour gérer plus finement l'erreur au sein d'une *routine* (4) particulière. Dans le script, le nom du serveur (`notexist`) a été choisi afin de provoquer l'erreur.

```
>>462 Source: Erreur d'exécution Microsoft VBScript Description: Le serveur distant
n'existe pas ou n'est pas disponible
```

La sortie affiche à l'écran une erreur dont le contenu est exactement organisé selon le format indiqué dans le script, format pouvant évidemment être tout autre.

L'instruction `On Error Resume Next` autorise la prise en compte par VBS du bloc de code (4) lié à la gestion d'erreurs. Dans le cas contraire, c'est-à-dire sans l'existence de cette instruction dans un script, le bloc de code en question est tout simplement ignoré par VBS :

```
>>C:\Scripts\script.vbs(7, 1) Erreur d'exécution Microsoft VBScript: Le serveur
distant n'existe pas ou n'est pas disponible: 'GetObject'
```

L'erreur retournée correspond alors à un affichage par défaut. Pour éviter ce comportement et évoluer vers une personnalisation de la gestion des erreurs, l'instruction `On Error Resume Next` doit obligatoirement exister dans un script, même si la marge de manœuvre dans ce domaine est très réduite.

Windows PowerShell

En PowerShell, il existe plusieurs façons de gérer les erreurs. Parmi elles se trouve l'instruction `try..catch`.

L'instruction try..catch

`try..catch` ressemble fortement à l'instruction `try..except` de Python.

Syntaxe de l'instruction try..catch

```
try (1) {
    <bloc de code> (2)
} catch (3) {
    <bloc de code> (4)
}
```

PowerShell exécute d'abord le bloc de code (2) lié à la clause `try` (1). S'il n'y a pas d'erreur, l'exécution continue à l'instruction qui suit immédiatement la structure `try..catch`. Sinon, PowerShell exécute le bloc de code (4) lié à la clause `catch` (3), puis passe à l'instruction suivante.

try..catch en action

La ligne de commande suivante est une division par zéro.

```
PS> 1/0
>>Tentative de division par zéro.
>>Au caractère Ligne:1 : 1
>>+ 1/0
>>+ ~~~
>>    + CategoryInfo          : NotSpecified: (:) [], RuntimeException
>>    + FullyQualifiedErrorId : RuntimeException
```

Comme attendu, diviser un nombre par zéro est sujet à erreur ; ce qui s'affiche en console est une exception .NET. Il faut reconnaître qu'elle n'est pas très lisible pour le commun des utilisateurs. L'instruction `try..catch` est alors très utile pour « attraper » l'erreur et la personnaliser :

```
try
{
    1/0
}
catch
{
    Write-Host "Math exception occurred."
}
```

Le code principal est encapsulé dans le bloc `try` et le message d'erreur est personnalisé au sein du bloc `catch`. L'exécution de ce code donne ce qui suit :

```
>>Math exception occurred.
```

Voici une autre instruction `try..catch` se connectant au service WMI d'une machine distante pour y collecter le statut des services :

```
try
{
    Get-WmiObject -Class Win32_Service -ComputerName 'notexist'
}
catch
{
    Write-Host "Impossible to connect to WMI service."
}
```

Le nom de la machine distante en argument du paramètre `-ComputerName` est factice, ce qui, après une tentative de connexion au service WMI distant, poussera PowerShell à rechercher l'existence d'un bloc `catch` et donc à afficher une erreur qui n'est pas celle par défaut :

```
>>Impossible to connect to WMI service.
```

À l'inverse, si le nom de machine distante est correct et si en plus la machine visée est accessible, alors PowerShell ignorera tout simplement le bloc `catch` :

```
>>ExitCode  : 0
>>Name      : AeLookupSvc
>>ProcessId : 872
>>StartMode : Manual
>>State     : Running
>>Status    : OK

>>ExitCode  : 1077
>>Name      : ALG
>>ProcessId : 0
>>StartMode : Manual
>>State     : Stopped
>>Status    : OK

>>ExitCode  : 1077
>>Name      : AllUserInstallAgent
>>ProcessId : 0
>>StartMode : Manual
>>State     : Stopped
>>Status    : OK
...
```

L'instruction venant tout de suite après la structure `try..catch`, s'il y en a une, est ensuite évaluée.

Les exemples que nous venons de voir ne constituent qu'une simple introduction à l'instruction `try..catch`. La maîtriser demande un peu plus de pratique.

PARTIE 2

Le scripting en pratique

13

Les expressions régulières

Le monde des expressions régulières est extrêmement vaste. Ces dernières, qui ne sont rien d'autre que des motifs servant à trouver des occurrences de mots à partir de formats très spécifiques, sont bien connues des développeurs, mais plutôt méconnues des administrateurs système et réseau. Le fait est que les maîtriser n'est pas chose aisée parce qu'elles peuvent être considérées, en quelque sorte, comme un langage de programmation à part entière. Il faut donc être très patient si on veut progresser dans ce domaine.

Dans ce chapitre, nous verrons de manière très synthétique comment utiliser des expressions régulières dans les cinq langages étudiés. Toutefois, il faudrait un livre complet pour maîtriser les expressions régulières ; nous nous familiariserons simplement avec leur utilisation afin de les intégrer à juste titre dans l'arithmétique de nos réflexes.

KSH

Pour utiliser les expressions régulières, Korn Shell propose un certain nombre de commandes dites externes. Parmi elles figurent les commandes `sed` et `awk`.

La commande sed

`sed` est ce que l'on appelle un éditeur de flux dans le traitement et la transformation de texte. En général, il s'agit du contenu d'un fichier texte, mais ce peut aussi évidemment être une entrée en provenance d'un pipeline.

Nous ne rentrerons pas dans les détails de cette commande, mais nous nous intéresserons à son fonctionnement de base :

```
sed [options] (1) <instructions> (2) [fichier] (3)
```

Comme toute commande Unix/Linux, `sed` prend des options (1) qui orientent son comportement. Les instructions (2) apportent plus de précision. Un fichier (3) passé en argument contient le flux que devra traiter la commande `sed`.

Voici le contenu d'un simple fichier texte nommé `file.txt` :

```
This is the line 1
This is the line 2
This is the line 3
This is the line 4
This is the line 5
This is the line 6
This is the line 7
This is the line 8
This is the line 9
This is the line 10
```

L'option `-n` n'affiche que les lignes spécifiées. La commande `sed` suivante autorise l'extraction de la première ligne (`-n '1p'`) :

```
sed -n '1p' file.txt

>>This is the line 1
```

Si nous voulons afficher les cinq premières lignes, il faut préciser `-n '1,5p'` :

```
sed -n '1,5p' file.txt

>>This is the line 1
>>This is the line 2
>>This is the line 3
>>This is the line 4
>>This is the line 5
```

À l'inverse, s'il faut afficher un certain nombre de lignes sur la base d'une exclusion, par exemple toutes les lignes du fichier `file.txt` sauf les deux premières, il faut utiliser la négation (caractère `!`) :

```
sed -n '1,3!p' file.txt

>>This is the line 3
>>This is the line 4
>>This is the line 5
```

```
>>This is the line 6
>>This is the line 7
>>This is the line 8
>>This is the line 9
>>This is the line 10
```

Le remplacement d'une occurrence particulière s'effectue de la manière suivante :

```
sed -ne 's/This/Here/p' file.txt

>>Here is the line 1
>>Here is the line 2
>>Here is the line 3
>>Here is the line 4
>>Here is the line 5
>>Here is the line 6
>>Here is the line 7
>>Here is the line 8
>>Here is the line 9
>>Here is the line 10
```

La forme `s/<mot_à_remplacer>/<mot_remplaçant>/p` est une des formes d'expression de la commande `sed` ; `s/` signifie substitution et `/p` l'affichage de la ligne en sortie standard.

> **NOTE La commande sed**
>
> Par défaut, la commande `sed` traite un flux entrant sans en modifier l'origine. Le flux modifié ne l'est donc qu'en sortie standard.

La commande awk

`awk` s'appuie sur les expressions régulières et est un excellent complément à la commande `sed`. En effet, elle est redoutablement efficace pour ce qui est du traitement de fichiers avec colonnes ou champs. Comme pour `sed`, la syntaxe de l'outil `awk` est relativement complexe, mais concentrons-nous sur son fonctionnement de base :

```
awk '[options] (1) {action}' (2) fichier (3)
```

Des actions (2) sont réalisées sur un fichier (3). En fonction des options (1), il sera possible, par exemple, d'extraire le contenu d'un champ particulier ou même d'effectuer des actions similaires au comportement de la commande `sed` ou `grep`.

La commande `awk` suivante extrait, à partir du fichier `/etc/passwd`, le premier et le cinquième champ :

```
awk -F: '{ print $1,"->",$5 }' /etc/passwd
```

```
>>root -> root
>>daemon -> daemon
>>bin -> bin
>>sys -> sys
>>sync -> sync
>>games -> games
>>man -> man
>>lp -> lp
>>mail -> mail
>>news -> news
...
```

Dans le fichier `/etc/passwd`, chaque champ est séparé d'un autre par le signe `:`, d'où la nécessité de spécifier l'option `-F`. Les champs sont quant à eux spécifiés via les variables `$n`.

Toujours dans ce même fichier, effectuons une extraction d'une ligne, si elle existe, contenant `scripting` comme valeur de premier champ :

```
awk -F: '{ if ($1 ~ /scripting/) print }' /etc/passwd

>>scripting:x:1000:1000:scripting,,,:/home/scripting:/bin/bash
```

Une instruction conditionnelle `if` a été insérée en tant qu'action principale : si la valeur du premier champ (`$1`) contient (~) la valeur indiquée (`/scripting/`), la ligne entière doit alors être affichée (`print`) en sortie standard.

Perl

Perl est le langage des expressions régulières par excellence ! Ceux qui le connaissent bien savent combien Perl est capable de flexibilité et de souplesse. Cette section mentionnera la syntaxe ainsi que quelques exemples aidant à mieux la comprendre.

La syntaxe

Un motif particulier est structuré à partir d'une syntaxe spécifique. Les tableaux ci-après évoquent la syntaxe la plus utilisée en Perl.

Tableau 13–1 Syntaxe de base des expressions régulières en Perl

Format	Description
^	Début de la chaîne de caractères.
*	Zéro, une ou plusieurs correspondance(s).
$	Fin de la chaîne de caractères.
+	Une ou plusieurs correspondance(s).
?	Zéro ou une correspondance.

Tableau 13–1 Syntaxe de base des expressions régulières en Perl *(suite)*

Format	Description
`|`	Une alternative possible.
`[]`	Un ensemble de caractères.
`\`	Permet d'interpréter littéralement un caractère.

Tableau 13–2 Classes de caractères utilisées en Perl

Format	Description
`\w`	Un caractère (lettre ou chiffre).
`\W`	Un caractère qui n'est ni une lettre, ni un chiffre.
`\s`	Un caractère espace.
`\S`	Un caractère qui n'est pas une espace.
`\d`	Un chiffre.
`\D`	Un caractère qui n'est pas un chiffre.

Tableau 13–3 Quantificateurs reconnus en Perl

Format	Description
`*`	Zéro, une ou plusieurs correspondance(s).
`+`	Une ou plusieurs correspondance(s).
`?`	Zéro ou une correspondance.
`{n}`	Exactement `n` fois l'élément précédent.
`{n,}`	Au moins `n` fois l'élément précédent.
`{n, m}`	Entre `n` et `m` fois l'élément précédent.

Tableau 13–4 Principaux caractères d'échappement en Perl

Format	Description
`\n`	Saut de ligne.
`\t`	Tabulation.
`\r`	Retour chariot.

Quelques exemples

Voici quelques motifs très utilisés :

- `^abc` : la chaîne de caractères doit commencer par les lettres « abc ».
- `abc$` : la chaîne de caractères doit se terminer par les lettres « abc ».
- `a|b` : soit « a », soit « b ».
- `ab+c` : la lettre « a » doit être suivie d'au moins une lettre « b », elle(s)-même(s) suivie(s) de « c ».
- `ab*c` : la lettre « a » doit être suivie de zéro, une ou plusieurs lettre(s) « b », elle(s)-même(s) suivie(s) de « c ».
- `ab?c` : la lettre « a » doit être suivie de zéro ou une lettre « b », elle-même suivie de « c ».

- [abc] : correspondance sur au moins un caractère parmi « a », « b » et « c ».
- [^abc] : n'importe quel caractère sauf « a », « b » et « c ».
- \w+ : un mot pouvant contenir une lettre et/ou un chiffre et/ou un signe tiret bas « _ », mais sans espace(s).
- \d+ : un ou plusieurs chiffre(s).
- \s+ : un ou plusieurs espace(s).

Python

Python, dans la lignée de Perl, fournit des possibilités très vastes en matière d'expressions régulières. Toutefois, en Python, les mécanismes sont différents.

La syntaxe

Pour ce qui est de la syntaxe, Python se rapproche de Perl, comme le mettent en évidence les tableaux suivants.

Tableau 13–5 Syntaxe de base des expressions régulières en Python

Format	Description
^	Début de la chaîne de caractères.
*	Zéro, une ou plusieurs correspondance(s).
$	Fin de la chaîne de caractères.
+	Une ou plusieurs correspondance(s).
?	Zéro ou une correspondance.
|	Une alternative possible.
[]	Un ensemble de caractères.
\	Permet d'interpréter littéralement un caractère.

Tableau 13–6 Classes de caractères utilisées en Python

Format	Description
\w	Un caractère (lettre ou chiffre).
\W	Un caractère qui n'est ni une lettre, ni un chiffre.
\s	Un caractère espace.
\S	Un caractère qui n'est pas une espace.
\d	Un chiffre.
\D	Un caractère qui n'est pas un chiffre.

Tableau 13–7 Quantificateurs reconnus en Python

Format	Description
*	Zéro, une ou plusieurs correspondance(s).
+	Une ou plusieurs correspondance(s).

Tableau 13–7 Quantificateurs reconnus en Python *(suite)*

Format	Description
`?`	Zéro ou une correspondance.
`{n}`	Exactement `n` fois l'élément précédent.
`{n,}`	Au moins `n` fois l'élément précédent.
`{n, m}`	Entre `n` et `m` fois l'élément précédent.

Tableau 13–8 Principaux caractères d'échappement en Python

Format	Description
`\n`	Saut de ligne.
`\t`	Tabulation.
`\r`	Retour chariot.

Le module re

L'utilisation des expressions régulières en Python passe par l'importation d'un module nommé `re`. Il contient certaines fonctions qu'il est intéressant de connaître, car elles sont les plus utilisées.

La fonction search()

Tout d'abord, le module `re` doit être importé :

```
import re
```

Cette étape est une condition sine qua non pour accéder aux différentes fonctions existantes comme `search()`, qui permet essentiellement de renvoyer la position des chaînes recherchées :

```
string = "Msdcl Python Perl KSH PowerShell VBS."
```

Dans cet exemple, une variable contient une chaîne constituée de certains mots. Parmi ces derniers, essayons de trouver la position de départ du mot « Python » :

```
print(re.search('Python', string).start())
```

La fonction `search()` prend en arguments un motif ainsi qu'une chaîne sur laquelle ce dernier va s'appliquer. Ici, nous recherchons la position de départ du mot « Python » à l'aide de la méthode `start()`, ce qui donne comme résultat :

```
>>6
```

Si nous voulions obtenir la position de fin de ce même mot au sein de la chaîne `string`, alors nous utiliserions la méthode `end()`.

```
print(re.search('Python', string).end())
>>12
```

La fonction sub()

La fonction `sub()` est un excellent moyen pour remplacer des occurrences de mots. Reprenons la chaîne précédente :

```
string = "Msdcl Python Perl KSH PowerShell VBS."
```

Remplaçons « Python » par le mot « Nothing » :

```
print(re.sub('Python', 'Nothing', string))
```

Trois arguments ont été passés à la fonction `sub()` : le premier est le mot à remplacer, le deuxième est le mot remplaçant et le troisième est la chaîne sur laquelle va s'appliquer l'opération de remplacement. Voici le résultat :

```
>>Msdcl Nothing Perl KSH PowerShell VBS.
```

Ceci n'est évidemment qu'une simple introduction au module `re` ; pour plus d'informations, nous vous invitons à consulter la documentation officielle.

VBS

Comme de nombreux autres langages de script, Visual Basic Scripting s'inscrit lui aussi dans un héritage vis-à-vis de Perl en ce qui concerne les expressions régulières.

La syntaxe

Tableau 13–9 Syntaxe de base des expressions régulières en VBS

Format	Description
^	Début de la chaîne de caractères.
*	Zéro, une ou plusieurs correspondance(s).
$	Fin de la chaîne de caractères.
+	Une ou plusieurs correspondance(s).
?	Zéro ou une correspondance.
|	Une alternative possible.
[]	Un ensemble de caractères.
\	Permet d'interpréter littéralement un caractère.

Tableau 13–10 Classes de caractères utilisées en VBS

Format	Description
`\w`	Un caractère (lettre ou chiffre).
`\W`	Un caractère qui n'est ni une lettre, ni un chiffre.
`\s`	Un caractère espace.
`\S`	Un caractère qui n'est pas une espace.
`\d`	Un chiffre.
`\D`	Un caractère qui n'est pas un chiffre.

Tableau 13–11 Quantificateurs reconnus en VBS

Format	Description
`*`	Zéro, une ou plusieurs correspondance(s).
`+`	Une ou plusieurs correspondance(s).
`?`	Zéro ou une correspondance.
`{n}`	Exactement `n` fois l'élément précédent.
`{n,}`	Au moins `n` fois l'élément précédent.
`{n, m}`	Entre `n` et `m` fois l'élément précédent.

Tableau 13–12 Principaux caractères d'échappement en VBS

Format	Description
`\n`	Saut de ligne.
`\t`	Tabulation.
`\r`	Retour chariot.

La réalité est que cette syntaxe est celle de beaucoup de langages de script ; bien la connaître rend donc plus efficace et facilite le passage d'un langage à l'autre.

L'objet RegExp

Visual Basic Scripting fournit un objet COM nommé `RegExp`, par lequel les programmeurs doivent passer pour utiliser les expressions régulières. Comme tout objet, `RegExp` est doté d'un certain nombre de méthodes et propriétés facilitant l'utilisation des expressions régulières.

La ligne suivante figure le contenu d'un fichier nommé `File.txt` :

```
Perl VBS Python VBS Lisp PowerShell VBS
```

Il s'agit tout simplement d'une liste contenant plusieurs occurrences du mot « VBS ». Le script ci-après lit (1) le fichier `File.txt` et obtient la position d'index de départ de chaque occurrence du mot « VBS » :

```
strFileName = "C:\Scripts\File.txt" (1)
Set objFS = CreateObject("Scripting.FileSystemObject")
```

```
Set objTS = objFS.OpenTextFile(strFileName)
strFileContents = objTS.ReadAll
WScript.Echo "Searching Within : "
WScript.Echo strFileContents (2)
objTS.Close

Set objRE = New RegExp (3)
objRE.Global     = True (4)
objRE.IgnoreCase = False (5)
objRE.Pattern    = "VBS" (6)

Set colMatches = objRE.Execute(strFileContents) (7)
WScript.Echo vbNewLine & "Resulting Matches :"
For Each objMatch In colMatches
   WScript.Echo "At position " & objMatch.FirstIndex & " matched " & objMatch.Value (8)
Next
```

Au cours de l'exécution du script, le contenu du fichier `File.txt` est affiché en sortie à l'écran (2), puis un objet `RegExp` (3) est instancié et placé dans la variable `objRE`. Celle-ci a plusieurs propriétés.

- `Global` cherche toutes les occurrences possibles liées à un motif. Ici, la valeur est `True` (4).
- `IgnoreCase` indique si la recherche est sensible à la casse ou non. Ici, la valeur est `False` (5).
- `Pattern` spécifie le motif à partir duquel la recherche va être effectuée. Ici, la valeur est `"VBS"` (6).

La méthode `Execute()` (7) de l'objet `objRE` lance la recherche sur le contenu du fichier `File.txt`. Enfin, l'instruction `For Each..In..Next` parcourt chaque occurrence trouvée et en déduit sa position d'index de départ (8) :

```
Searching Within :
Perl VBS Python VBS Lisp PowerShell VBS

Resulting Matches :
At position 5 matched VBS
At position 16 matched VBS
At position 36 matched VBS
```

Le résultat met en évidence trois occurrences du mot « VBS », aux index de départ `5`, `16` et `36`.

À travers cet exemple, nous pouvons constater la place centrale qu'occupe l'objet `RegExp`. Toute démarche voulant être à la fois efficace et liée aux expressions régulières devra nécessairement passer par l'utilisation de cet objet.

Windows PowerShell

PowerShell fournit plusieurs méthodes pour utiliser les expressions régulières. Parmi elles, nous présenterons les opérateurs `-match` et `-notmatch`, ainsi que la cmdlet `Select-String`.

Les opérateurs -match et -notmatch

Les recours aux expressions régulières sont à l'évidence différents selon les objectifs recherchés. Si parfois le motif défini est extrêmement simple, il peut également être très complexe. PowerShell propose deux opérateurs pour définir des motifs relativement peu complexes. Le premier est `-match`, qui cherche une correspondance à partir d'un motif particulier : une entrée spécifiée à gauche de l'opérateur est analysée à l'aide d'un motif placé, quant à lui, à sa droite. `-match` retourne `True` si une correspondance est trouvée, `False` sinon :

```
PS> 'Windows PowerShell' -match 'Win\w+'
>>True

PS> 'Windows PowerShell' -match 'VBS'
>>False
```

Une variable nommée `$Matches` contient de manière précise la correspondance trouvée lorsque la valeur `True` est retournée :

```
PS> $Matches

>>Name                           Value
>>----                           -----
>>0                              Windows
```

La variable `$Matches` est un dictionnaire.

```
PS> $Matches | Get-Member

>>   TypeName : System.Collections.Hashtable

>>Name              MemberType            Definition
>>----              ----------            ----------
>>Add               Method                void Add(System.Object key, Sy
>>Clear             Method                void Clear(), void IDictionary
>>Clone             Method                System.Object Clone(), System.
>>Contains          Method                bool Contains(System.Object ke
>>ContainsKey       Method                bool ContainsKey(System.Object
>>ContainsValue     Method                bool ContainsValue(System.Obje
>>CopyTo            Method                void CopyTo(array array, int a
>>Equals            Method                bool Equals(System.Object obj)
>>GetEnumerator     Method                System.Collections.IDictionary
>>GetHashCode       Method                int GetHashCode()
>>GetObjectData     Method                void GetObjectData(System.Runt
>>GetType           Method                type GetType()
>>OnDeserialization Method                void OnDeserialization(System.
>>Remove            Method                void Remove(System.Object key)
>>ToString          Method                string ToString()
```

```
>>Item              ParameterizedProperty System.Object Item(System.Obje
>>Count             Property              int Count {get;}
>>IsFixedSize       Property              bool IsFixedSize {get;}
>>IsReadOnly        Property              bool IsReadOnly {get;}
>>IsSynchronized    Property              bool IsSynchronized {get;}
>>Keys              Property              System.Collections.ICollection
>>SyncRoot          Property              System.Object SyncRoot {get;}
>>Values            Property              System.Collections.ICollection
```

L'opérateur `-notmatch` fonctionne de manière exactement opposée, c'est-à-dire que la valeur `True` est retournée si aucune correspondance n'est trouvée, `False` sinon :

```
PS> 'Windows PowerShell' -notmatch 'Lisp'
>>True

PS> 'Windows PowerShell' -notmatch 'Windows'
>>False
```

La cmdlet Select-String

L'angle d'utilisation des expressions régulières que nous venons d'évoquer dans la section précédente est très rudimentaire. Il existe une cmdlet, `Select-String`, ouvrant la perspective à une utilisation plus riche et plus complexe.

Elle est en quelque sorte l'équivalent de l'outil `grep` sous Unix/Linux :

```
PS> 'Lisp','VBS','vbs' | Select-String -Pattern 'VBS' -CaseSensitive
>>VBS
```

Cet exemple consiste en une analyse de trois chaînes. Ces dernières sont envoyées à la cmdlet `Select-String` afin qu'elle puisse extraire, si elle existe, une occurrence (`'VBS'`) précise à l'aide du paramètre `-Pattern`, mais tout en respectant la casse (`-CaseSensitive`). Le résultat montre effectivement que seul le mot « VBS » a été retenu.

Pour réaliser une recherche non sensible à la casse, le paramètre `-CaseSensitive` doit être omis :

```
PS> 'Lisp','VBS','vbs' | Select-String -Pattern 'VBS'

>>VBS
>>vbs
```

Ici, la sortie affiche bien deux occurrences et non pas une seule. La cmdlet `Select-String` est également utilisée dans d'autres cas de figure, comme l'articulation avec d'autres cmdlets produisant, par exemple, des sorties plus ou moins bavardes.

```
PS> Get-Process | Out-String -Stream | Select-String -Pattern 'svchost'

>>    954      36    26484      24628   139             852 svchost
>>    500      19     8272      11368    62             892 svchost
>>    647      18     9652      10400    54             932 svchost
>>   3065      66   306844     192880   669            1064 svchost
>>    859      45    19096      19504   121            1104 svchost
>>    880      43   119592     106592   175            1232 svchost
>>    670      37    16592      17876  1189            1348 svchost
>>    526      39    29968      27860   151            1640 svchost
>>    212      13     3048       3172    47            2256 svchost
>>    536      30     9324      10304    69            2528 svchost
>>    402      24     7612      10068   632            3376 svchost
```

Dans cet exemple, la cmdlet `Get-Process` produit une liste de processus en cours d'exécution. Cette liste est envoyée à la cmdlet `Out-String` afin de convertir chaque objet en chaîne. La cmdlet `Out-String` envoie par la suite chaque objet traité à la cmdlet `Select-String` pour que l'analyse ait lieu.

En PowerShell, filtrer une sortie d'objets à l'aide des expressions régulières est une opération réalisable de plusieurs façons ; le mécanisme en lien avec la cmdlet `Select-String` en est une des principales illustrations.

14

La gestion de fichiers

La gestion de fichiers est un ensemble d'opérations largement répandu dans le domaine du scripting. En effet, créer un fichier, le lire, le modifier et le supprimer sont des démarches que les administrateurs rencontrent tout au long de leur travail. Ces fichiers peuvent contenir des noms de serveurs, des listes d'adresses IP ou même toutes sortes d'informations sensibles issues de sources hétérogènes.
Savoir manipuler un fichier au sein d'un script est donc une étape préalable à tout traitement ultérieur des données obtenues. Ce chapitre nous familiarisera avec les opérations de base concernant le traitement de fichiers, ce qui nous aidera par la suite à mieux cerner le rôle essentiel de ces derniers dans l'écosystème du scripting.

KSH

Gérer des fichiers en KSH s'effectue à l'aide de commandes shell tierces. Ce point est particulièrement intéressant, car le recours aux commandes shell entraîne sans conteste un gain de temps non négligeable.

Créer un fichier

Il existe plusieurs façons de créer un fichier en Korn Shell. La ligne de commande suivante utilise la commande `echo` avec une redirection de flux vers le fichier créé :

```
scripting@debiandev:~$ echo 'This is the line 1' > File.txt
```

Celle-ci utilise plutôt la commande `printf` :

```
scripting@debiandev:~$ printf 'This is the line 1\n' > File.txt
```

Lire le contenu d'un fichier

La commande `cat` sert à lire et afficher le contenu d'un fichier :

```
scripting@debiandev:~$ cat File.txt
>>This is the line 1
```

Modifier le contenu d'un fichier

L'opérateur de redirection `>>` modifie, le cas échéant, le contenu d'un fichier :

```
scripting@debiandev:~$ echo 'This is the line 2' >> File.txt
scripting@debiandev:~$ cat File.txt
>>This is the line 1
>>This is the line 2
```

Supprimer un fichier

La suppression d'un fichier est possible via la commande shell `rm` :

```
scripting@debiandev:~$ rm File.txt
```

Comme nous pouvons le constater, une simple ligne de commande suffit pour gérer de manière basique un fichier, ce qui représente un gain de temps non négligeable par rapport à d'autres langages de script.

Perl

En Perl, contrairement à Korn Shell, la gestion de fichiers se fait essentiellement par l'appel de fonctions natives.

Créer un fichier

Pour créer un fichier en Perl, la fonction utilisée est `open()` :

```
$File = "File.txt";

unless(open FILE, '>'.$File) {
```

```
        die "Unable to create $File\n";
}

print FILE "This is the line 1\n";

close FILE;
```

Il ne faudra pas oublier de préfixer le nom avec le caractère `>`, nécessaire à Perl pour créer le fichier spécifié.

Lire le contenu d'un fichier

La fonction `open()` est également invoquée lorsqu'il s'agit de lire un fichier :

```
$File = "File.txt";

unless(open FILE, $File) {
        die "Unable to open $File\n";
}

while(my $line = <FILE>) {
        print $line;
}

close FILE;
```

Cette fois-ci, il n'est nul besoin de préfixer le nom du fichier par le caractère `>`. L'instruction `while` parcourt le fichier ouvert et affiche les lignes les unes après les autres.

Modifier le contenu d'un fichier

La fonction `print()` est très utile pour modifier le contenu d'un fichier :

```
$File = "File.txt";

unless(open FILE, '>>'.$File) {
        die "Unable to read $File";
}

print FILE "This is the line 2\n";

close FILE;
```

La fonction `open()` appelée avec le préfixe `>>` sert à incrémenter le contenu du fichier `File.txt`, puis la fonction `print()` agit ici en tant que cause produisant l'ajout du contenu passé en argument.

Supprimer un fichier

La suppression d'un fichier se réalise à l'aide de la fonction `unlink()` :

```
$File = "File.txt";
unlink $File;
```

Python

Python s'inscrit dans une logique similaire à celle de Perl, c'est-à-dire que des fonctions natives sont les outils principaux quant à la gestion de fichiers.

Créer un fichier

Comme en Perl, il existe une fonction nommée `open()` servant entre autres à la création de fichiers :

```
file = open("File.txt", "w")

file.write("This is the line 1\n")

file.close()
```

Ici, la fonction `open()` a pris deux arguments ; le nom du fichier à créer et un mode d'accès à ce fichier. Le symbole `w` signifie *writing* ou accès en écriture. Cela crée le fichier tant qu'il n'existe pas.

L'objet renvoyé par la fonction `open()` a une méthode nommée `write()`. Celle-ci écrit dans le fichier vers lequel elle pointe un contenu passé en argument.

Lire le contenu d'un fichier

Pour lire le contenu d'un fichier, il faut y accéder en mode lecture, toujours à l'aide de la fonction `open()` :

```
file = open("File.txt", "r")

print(file.read())

file.close()
```

Le caractère symbolisant l'accès en lecture est `r` et la lecture à proprement parler s'effectue via la méthode `read()`.

Modifier le contenu d'un fichier

La modification est réalisée à l'aide de la méthode `write()` de l'objet renvoyé par la fonction `open()`, à condition bien sûr d'avoir spécifié le bon mode d'accès (`a` pour *append*) au fichier.

```
file = open("File.txt", "a")

print(file.write("This is the line 2\n"))

file.close()
```

Supprimer un fichier

Le module `os` contient une fonction nommée `remove()`, qui supprime précisément un fichier par le biais de son nom passé en argument :

```
import os

os.remove("File.txt")
```

VBS

Visual Basic Scripting dispose d'objets COM destinés à manipuler des fichiers. Ces objets sont natifs et disponibles par défaut.

Créer un fichier

L'objet essentiel concernant la manipulation de fichiers est `Scripting.FileSystemObject` :

```
Set objFSO = CreateObject("Scripting.FileSystemObject")
Set objFile = objFSO.CreateTextFile("C:\File.txt")
ObjFile.WriteLine("This is the line 1")
```

La fonction `CreateObject()` autorise la création d'un objet `Scripting.FileSystemObject` et la méthode `CreateTextFile()` de ce dernier permet la création d'un fichier dit vierge.

Lire le contenu d'un fichier

Le même objet `Scripting.FileSystemObject` servant d'appui à toute manipulation de fichiers, la lecture n'échappe donc pas à la règle :

```
Set objFSO = CreateObject("Scripting.FileSystemObject")
```

```
Set objFile = objFSO.OpenTextFile("C:\File.txt", 1)
Do Until objFile.AtEndOfStream
    strLine= objFile.ReadLine
    Wscript.Echo strLine
Loop
```

L'accès en lecture au fichier `File.txt` via la méthode `OpenTextFile()` est possible grâce au deuxième argument, qui est une constante : la valeur `1` signifie l'accès en mode lecture et l'instruction `Do Until..Loop` effectue une itération sur chaque ligne du fichier `File.txt`, même s'il n'y en a qu'une.

Modifier le contenu d'un fichier

La méthode `WriteLine()` incrémente le contenu d'un fichier. La différence avec `Write()` est que la première spécifie un saut de ligne :

```
Set objFSO = CreateObject("Scripting.FileSystemObject")

Set objFile = objFSO.OpenTextFile("C:\File.txt", 8)
objFile.WriteLine("This is the line 2")
```

La constante `8` signifie un accès au fichier `File.txt` en mode incrémentation.

Supprimer un fichier

On supprime un fichier en VBS avec la méthode `DeleteFile()` :

```
Set objFSO = CreateObject("Scripting.FileSystemObject")
objFSO.DeleteFile("C:\File.txt")
```

Cette méthode ne produit pas de sortie si la suppression du fichier est réussie.

Windows PowerShell

PowerShell propose des cmdlets facilitant la gestion de fichiers. La logique implicite est similaire à celle de Korn Shell.

Créer un fichier

La cmdlet `New-Item` a un comportement différent en fonction du contexte de son invocation. Nous n'entrerons pas dans les détails de sa structure, mais nous nous intéresserons plutôt à son utilisation en lien avec la création de fichiers.

```
PS> New-Item -Path . -Name File.txt -ItemType "file" -Value "This is the line 1"

>>    Répertoire : C:\Users\Kais\Desktop

>>Mode                 LastWriteTime     Length Name
>>----                 -------------     ------ ----
>>-a---        26/04/2014     16:37         18 File.txt
```

Le paramètre `-Path` indique le chemin d'accès au fichier, `-Name` le nom du fichier, `-ItemType` le type de l'élément créé et `-Value` fournit son contenu de départ. Enfin, la sortie met en évidence que le fichier `File.txt` a bien été créé.

Lire le contenu d'un fichier

La lecture de fichiers passe, entre autres moyens, par le biais de la cmdlet `Get-Content` :

```
PS> Get-Content -Path .\File.txt
>>This is the line 1
```

Cette commande obtient le contenu de l'élément spécifié, ici un fichier texte.

Modifier le contenu d'un fichier

Il y a plusieurs façons de modifier le contenu d'un fichier texte, mais la commande `Add-Content` est ici centrale :

```
PS> Add-Content -Path .\File.txt -Value "`nThis is the line 2"
```

La cmdlet `Add-Content` a permis l'ajout d'une nouvelle ligne et donc la modification du contenu du fichier `File.txt`.

Supprimer un fichier

La cmdlet `Remove-Item` supprime l'élément spécifié en fonction du contexte :

```
PS> Remove-Item -Path .\File.txt
```

L'élément spécifié est dans notre cas de figure un fichier, mais il peut s'agir d'autres types d'éléments comme des clés de registre.

15

Windows Management Instrumentation

Dans le monde des systèmes Microsoft, il existe une surface d'administration extrêmement structurée et à la fois éminemment centrale. Cette infrastructure de gestion des composants matériels et logiciels s'appelle WMI (Windows Management Instrumentation) *et constitue une base de données centralisée. Par base de données, j'entends un ensemble de données provenant de sources hétérogènes, mais dont le stockage est structuré en espaces de noms organisés par technologie.*
WMI est une construction logicielle particulièrement utile pour tous types de métiers. Qu'il s'agisse de développeurs ou d'administrateurs système et réseau, cette base de données centralisée fournit un certain nombre d'informations plus ou moins importantes et fait gagner beaucoup de temps. En outre, WMI peut être consulté à partir de différents langages de programmation, ce qui témoigne de sa grande utilité.

KSH

Dans une infrastructure de type hétérogène, il est très fréquent que certaines opérations se basent sur des systèmes différents. En ce qui concerne l'interrogation du service WMI à partir d'un client Unix/Linux, il existe un outil à la fois simple d'utilisation et capable de produire une certaine interaction.

L'outil wmic

`wmic` *(Windows Management Instrumentation Command-Line)* est une interface en ligne de commande capable de communiquer avec le service WMI. Son utilisation, à l'aide d'un client Windows ou Unix/Linux, revêt des orientations multiples, que ce soit en mode console ou scripting.

Les options possibles dans le cadre de l'utilisation de l'outil `wmic` sont nombreuses :

```
[-?|--help] [--usage] [-d|--debuglevel DEBUGLEVEL] [--debug-stderr]
[-s|--configfile CONFIGFILE] [--option=name=value] [-l|--log-basename
LOGFILEBASE] [--leak-report] [--leak-report-full] [-R|--name-resolve
NAME-RESOLVE-ORDER] [-O|--socket-options      SOCKETOPTIONS]
[-n|--netbiosâname      NETBIOSNAME]     [-W|--workgroup     WORKGROUP]
[--realm=REALM] [-i|--scope        SCOPE]         [-m|--maxprotocol
MAXPROTOCOL] [-U|--user [DOMAIN]USERNAME[%PASSWORD]][-N|--no-
pass] [--password=STRING]   [-A|--authentication-file     FILE]
[-S|--signing       on|off|required] [-P|--machine-pass]
[--simple-bind-dn=STRING]     [-k|--kerberos       STRING]
[--use-security-mechanisms=STRING] [-V|--version] [--namespace=STRING]
//host query
```

Nous ne les aborderons pas toutes, mais les exemples qui vont suivre nous donneront un aperçu significatif des possibilités offertes par cet outil.

WMI en pratique

L'exemple suivant est une requête WMI obtenant comme information la mémoire physique disponible d'une machine distante. C'est la propriété `FreePhysicalMemory` qui contient la valeur recherchée :

```
scripting@SuseLinux$ wmic -U admdevel%156/*gh //192.168.10.19 "SELECT
FreePhysicalMemory FROM Win32_OperatingSystem"

>>CLASS: Win32_OperatingSystem
>>FreePhysicalMemory
>>1104656
```

Interrogeons à présent une autre classe, `Win32_ComputerSystem`, contenant des informations importantes, comme le nom du domaine auquel est rattachée une machine donnée :

```
scripting@SuseLinux$ wmic -U admdevel%156/*gh //192.168.10.19 "SELECT Domain FROM
Win32_ComputerSystem"

>>CLASS: Win32_ComputerSystem
>>Domain|Name
>>contoso.com|MLXDEVEL
```

La requête est de type WQL *(WMI Query Language)*. Essayons d'obtenir, toujours en suivant cette logique, le nombre de processeurs disponibles au sein de cette machine :

```
scripting@SuseLinux$ wmic -U admdevel%156/*gh //192.168.10.19 "SELECT
NumberOfProcessors FROM Win32_ComputerSystem"

>>CLASS: Win32_ComputerSystem
>>Name|NumberOfProcessors
>>MLXDEVEL|2
```

Comme nous pouvons le constater, `wmic` est un outil extrêmement utile dans un contexte d'interaction avec le service WMI. En outre, intégré à un script Korn Shell, cet outil transformera ce langage de script en un client WMI très efficace.

Perl

Perl peut accéder au service Windows Management Instrumentation par le biais de divers modules. Dans cette section, nous en étudierons un en particulier : `Win32::OLE`.

Le module Win32::OLE

`Win32::OLE` est un module très utilisé par les programmeurs Perl désireux d'accéder à l'infrastructure WMI. Il permet en effet de manipuler de nombreux composants différents, par exemple :

- une feuille Excel ;
- un document Word ;
- une base de données ;
- un composant ActiveX ;
- un client de messagerie ;
- etc.

Le module `Win32::OLE` propose donc des fonctionnalités pour gérer l'infrastructure WMI. En voici quelques méthodes très intéressantes.

- `new()` crée un nouvel objet autorisant la manipulation du composant recherché.
- `GetObject()` retourne une référence vers l'objet spécifié.
- `ExecQuery()` exécute une requête à l'aide d'une syntaxe dépendante du composant interrogé.

Évidemment, les membres du module `Win32::OLE` sont abondants, mais ceux susmentionnés sont suffisants pour accéder au service WMI.

WMI en pratique

WMI est organisé en espaces de noms, eux-mêmes organisés en classes. Voici un script interrogeant l'une de ces classes, `Win32_ComputerSystem`, qui fournit des informations en lien avec le système d'exploitation ainsi que l'architecture de la machine concernée par la requête :

```
use Win32::OLE('in'); (1)

use constant wbemFlagReturnImmediately => 0x10; (2)
use constant wbemFlagForwardOnly => 0x20;

@computers = (".");
foreach $computer (@computers) {
   print "\n";

   $objWMIService = (3) Win32::OLE->GetObject("winmgmts:\\\\$computer\\root\\CIMV2")
or die "WMI connection failed.\n";
   $colItems = (4) $objWMIService->ExecQuery("SELECT * FROM Win32_ComputerSystem",
"WQL", wbemFlagReturnImmediately | wbemFlagForwardOnly);

   foreach (5) $objItem (in $colItems) {
      print "Caption: $objItem->{Caption}\n";
      print "CurrentTimeZone: $objItem->{CurrentTimeZone}\n";
      print "Description: $objItem->{Description}\n";
      print "DNSHostName: $objItem->{DNSHostName}\n";
      print "Domain: $objItem->{Domain}\n";
      print "DomainRole: $objItem->{DomainRole}\n";
      print "Manufacturer: $objItem->{Manufacturer}\n";
      print "Model: $objItem->{Model}\n";
      print "Name: $objItem->{Name}\n";
      print "NetworkServerModeEnabled: $objItem->{NetworkServerModeEnabled}\n";
      print "NumberOfProcessors: $objItem->{NumberOfProcessors}\n";
      print "Roles: " . join(",", (in $objItem->{Roles})) . "\n";
      print "Status: $objItem->{Status}\n";
      print "SystemType: $objItem->{SystemType}\n";
      print "ThermalState: $objItem->{ThermalState}\n";
      print "TotalPhysicalMemory: $objItem->{TotalPhysicalMemory}\n";
      print "UserName: $objItem->{UserName}\n";
      print "Workgroup: $objItem->{Workgroup}\n";
      print "\n";
   }
}
```

Au sein de ce script, le module `Win32::OLE` (1) est d'abord importé. Pour l'obtenir, l'endroit privilégié est le CPAN, ou *Comprehensive Perl Archive Network*. Il s'agit d'un dépôt officiel contenant essentiellement des modules et de la documentation autour du langage de script Perl.

Ensuite, deux constantes (2) définies spécifient les modalités d'accès au service WMI. Ici, il s'agit d'un appel semi-synchrone avec une gestion de la mémoire plus fine. La méthode `GetObject()` (3) se connecte au service WMI. Si la connexion, pour des raisons diverses, ne peut aboutir, Perl arrête l'exécution du script.

Au contraire, si la connexion a lieu, alors est appelée ❹ la méthode `ExecQuery()` afin d'obtenir un objet `Win32_ComputerSystem`. Enfin, accéder aux membres de cet objet par l'intermédiaire de l'instruction `foreach` ❺ nous aide à retourner les informations recherchées :

```
>>Caption: DEVEL
>>CurrentTimeZone: 120
>>Description: AT/AT COMPATIBLE
>>DNSHostName: DEVEL
>>Domain: WORKGROUP
>>DomainRole: 0
>>Manufacturer: LENOVO
>>Model: 2564
>>Name: DEVEL
>>NetworkServerModeEnabled: 1
>>NumberOfProcessors: 1
>>Roles: LM_Workstation,LM_Server,NT,Potential_Browser,Master_Browser
>>Status: OK
>>SystemType: x64-based PC
>>ThermalState: 3
>>TotalPhysicalMemory: 8543223808
>>UserName: DEVEL\Kais
>>Workgroup: WORKGROUP
```

Ces informations sont extrêmement utiles, surtout dans le cadre de la création de rapports.

Python

Python se situe dans une logique similaire à celle de Perl en ce qui concerne l'accès au service WMI, à savoir une interaction à partir d'une approche modulaire.

Le module wmi

Tout comme Perl, Python recourt à un module, qui se nomme `wmi`. Ce dernier comporte un certain nombre de mécanismes, dont voici les plus importants.

- Connexion au service : la méthode `WMI()` constitue une première étape en construisant un objet WMI.
- Interrogation d'une classe particulière : collecter des informations WMI s'effectue par l'intermédiaire de classes, qui doivent donc figurer en tant qu'attributs de l'objet créé, `Objet.Classe_WMI()`.
- Obtention et/ou modification des informations : chaque objet WMI créé contient des informations plus ou moins importantes, qui sont accessibles via des propriétés et dont les modifications d'état sont établies à l'aide de méthodes. Bien sûr, chaque objet propose des membres différents ; il n'y a donc pas de propriétés ou de méthodes types.

> **NOTE Module pywin32 et dépendance**
>
> Le module `pywin32` doit être installé conjointement avec le module `wmi`, sous peine de rencontrer des erreurs d'exécution.

WMI en pratique

Utiliser WMI nécessite d'abord de connaître exactement le niveau d'information recherché. C'est un élément fondamental car cette immense base de données est constituée de milliers de classes n'ayant pas forcément de rapport les unes avec les autres.

Si nous voulons par exemple lister l'ensemble des processus en cours d'exécution, il nous faut d'abord connaître le nom de la classe concernée. Le site MSDN *(Microsoft Developer Network)*, msdn.microsoft.com, est une véritable mine d'informations, notamment à propos de l'infrastructure WMI. Une simple exploration sur ce site renseigne sur la classe recherchée.

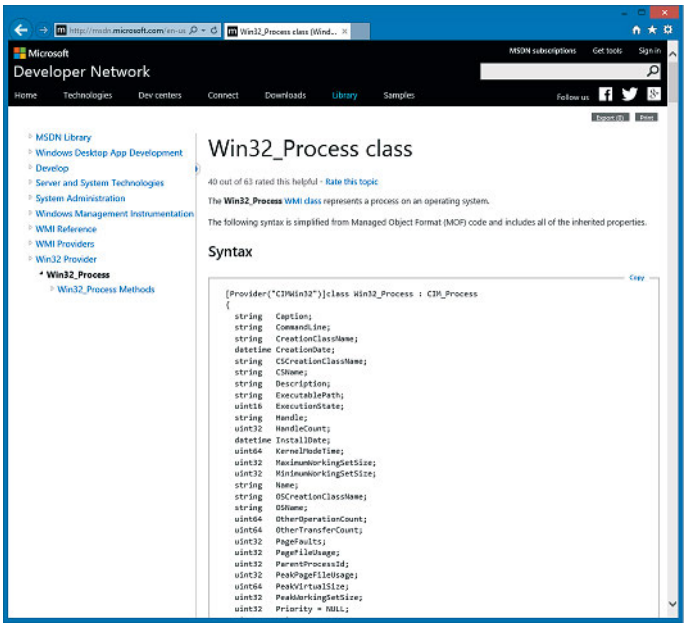


Figure 15–1
MSDN est un formidable vivier d'informations.

Pour collecter les informations, nous aurons besoin de la classe `Win32_Process`. La propriété qui nous intéresse se nomme `Caption`. Sur cette base, il n'y a plus qu'à écrire notre script. Commençons par importer le module `wmi` (1). Construisons ensuite un objet WMI vierge (2), auquel il faut donner une orientation précise correspondant à l'interrogation de la classe `Win32_Process` (3). La variable `processes` est une collection de processus en cours d'exécution ; pour obtenir le nom de chaque processus, l'instruction `for` va itérer à travers son contenu (4).

```
import wmi (1)
obj = wmi.WMI() (2)
processes = obj.Win32_Process() (3)

print("running processes:")
print("------------------\n")

for p in processes: (4)
    print(p.Caption)
```

Observons la sortie :

```
>>running processes:
>>------------------

>>System Idle Process
>>System
>>smss.exe
>>csrss.exe
>>wininit.exe
>>services.exe
>>lsass.exe
>>svchost.exe
>>svchost.exe
>>nvvsvc.exe
>>nvSCPAPISvr.exe
>>svchost.exe
>>svchost.exe
>>svchost.exe
>>svchost.exe
>>svchost.exe
>>spoolsv.exe
>>svchost.exe
>>armsvc.exe
>>AVerRemote.exe
>>AVerScheduleService.exe
...
```

La sortie affiche bien une liste de processus en cours d'exécution. Elle pourra constituer une base vers la création de statistiques.

VBS

Le service WMI fait partie des interfaces actives et accessibles nativement au langage Visual Basic Scripting. Il n'est donc pas nécessaire d'importer tel ou tel composant pour interroger cette base de données.

Accéder à WMI

En VBS, on accède au service WMI à l'aide de la fonction `GetObject()` (2). Cette dernière, dans ce contexte, connecte à une machine locale ou distante (locale dans l'exemple suivant (1)) :

```
strComputer = "." (1)
Set ObjWMICon=GetObject("winmgmts:{impersonationLevel=impersonate}!\\"
& strComputer & "\root\cimv2")(2)
```

Cette opération est nécessaire avant toute action sur une base de données WMI. Elle constitue en effet le processus de connexion sans lequel aucune manipulation ou obtention d'informations n'est possible.

WMI en pratique

WMI étant une immense base de données organisée en espaces de noms, il est parfois très difficile de trouver la ou les bonne(s) classe(s) susceptible(s) de nous aider quant à une démarche précise. Connaître les classes disponibles au sein d'un espace de noms particulier est donc toujours une démarche salutaire.

Un des espaces de noms les plus utilisés est `root\cimv2`. En effet, il contient un certain nombre de classes utilisées par de nombreuses applications, à tous les niveaux.

Le script suivant en dresse la liste :

```
strComputer = "."
Set objWMICon=GetObject("winmgmts:{impersonationLevel=impersonate}!\\" & strComputer
& "\root\cimv2") (1)

For Each objclass in objWMICon.SubclassesOf() (2)
   Wscript.Echo objClass.Path_.Class (3)
Next
```

Tout d'abord, une connexion est établie (1) au niveau d'une machine locale. Puis la méthode `SubClassesOf()` (2) de l'objet WMI créé est appelée dans le but d'obtenir toutes les classes existantes dans l'espace de noms `root\cimv2`. L'instruction `For Each..in..Next` itère à travers ces dernières pour en afficher le nom (3) :

```
>>MSFT_SCMEvent
>>MSFT_SCMEventLogEvent
>>MSFT_NetSevereServiceFailed
>>MSFT_NetTransactInvalid
>>MSFT_NetServiceNotInteractive
>>MSFT_NetTakeOwnership
>>MSFT_NetServiceConfigBackoutFailed
>>MSFT_NetServiceShutdownFailed
```

```
>>MSFT_NetServiceStartHung
>>MSFT_NetServiceStopControlSuccess
>>MSFT_NetServiceSlowStartup
>>MSFT_NetCallToFunctionFailed
>>MSFT_NetBadAccount
>>MSFT_NetBadServiceState
...
```

La sortie a été tronquée, car extrêmement longue. Certaines de ces classes sont évidemment plus utiles que d'autres selon la démarche, mais il appartient aux développeurs de connaître avec exactitude celles qui répondent précisément à leurs besoins.

Windows PowerShell

PowerShell est le langage de script qui a le plus innové vis-à-vis de l'infrastructure WMI. Il n'y a effectivement plus besoin de passer par de la « programmation système » pour accéder à ce service. Au contraire, l'existence de certaines cmdlets, dont une en particulier, facilite la programmation WMI.

La cmdlet Get-WmiObject

Il existe une cmdlet dont le fonctionnement est dédié à l'interaction avec le service WMI ; elle se nomme `Get-WmiObject` :

```
PS> Get-Help Get-WmiObject

NOM
    Get-WmiObject

RÉSUMÉ
   Gets instances of Windows Management Instrumentation (WMI) classes or information
about the available classes.

SYNTAXE
    Get-WmiObject [-Class] <String> [[-Property] <String[]>] [-Amended] [-AsJob]
    [-Authentication <AuthenticationLevel>] [-Authority <String>] [-ComputerName
    <String[]>] [-Credential <PSCredential>] [-DirectRead] [-EnableAllPrivileges]
    [-Filter <String>] [-Impersonation <ImpersonationLevel>] [-Locale <String>]
    [-Namespace <String>] [-ThrottleLimit <Int32>] [<CommonParameters>]

    Get-WmiObject [[-Class] <String>] [-Amended] [-AsJob] [-Authentication
    <AuthenticationLevel>] [-Authority <String>] [-ComputerName <String[]>]
    [-Credential <PSCredential>] [-EnableAllPrivileges] [-Impersonation
    <ImpersonationLevel>] [-List] [-Locale <String>] [-Namespace <String>] [-Recurse]
    [-ThrottleLimit <Int32>] [<CommonParameters>]
```

```
    Get-WmiObject [-Amended] [-AsJob] [-Authentication <AuthenticationLevel>]
    [-Authority <String>] [-ComputerName <String[]>] [-Credential <PSCredential>]
    [-EnableAllPrivileges] [-Impersonation <ImpersonationLevel>] [-Locale <String>]
    [-Namespace <String>] [-ThrottleLimit <Int32>] [<CommonParameters>]

    Get-WmiObject [-Amended] [-AsJob] [-Authentication <AuthenticationLevel>]
    [-Authority <String>] [-ComputerName <String[]>] [-Credential <PSCredential>]
    [-EnableAllPrivileges] [-Impersonation <ImpersonationLevel>] [-Locale <String>]
    [-Namespace <String>] [-ThrottleLimit <Int32>] [<CommonParameters>]

    Get-WmiObject [-Amended] [-AsJob] [-Authentication <AuthenticationLevel>]
    [-Authority <String>] [-ComputerName <String[]>] [-Credential <PSCredential>]
    [-DirectRead] [-EnableAllPrivileges] [-Impersonation <ImpersonationLevel>]
    [-Locale <String>] [-Namespace <String>] [-ThrottleLimit <Int32>] -Query <String>
    [<CommonParameters>]

DESCRIPTION
    The Get-WmiObject cmdlet gets instances of WMI classes or information about the
available WMI classes.
    To specify a remote computer, use the ComputerName parameter. If the List
parameter is specified, the cmdlet gets information about the WMI classes that are
available in a specified namespace. If the Query
parameter is specified, the cmdlet runs a WMI query language (WQL) statement.

    The Get-WmiObject cmdlet does not use Windows PowerShell remoting to perform
remote operations. You can use the ComputerName parameter of the Get-WmiObject cmdlet
even if your computer does not meet the    requirements for Windows PowerShell
remoting or is not configured for remoting in Windows PowerShell.

   Beginning in Windows PowerShell 3.0, the __Server property of the object that Get-
WmiObject returns has a PSComputerName alias. This makes it easier to include the
source computer name in output and reports.

LIENS CONNEXES
    Online Version: http://go.microsoft.com/fwlink/?LinkID=113337
    Invoke-WmiMethod
    Remove-WmiObject
    Set-WmiInstance
    Get-WSManInstance
    Invoke-WSManAction
    New-WSManInstance
    Remove-WSManInstance

REMARQUES
    Pour consulter les exemples, tapez : "get-help Get-WmiObject -examples".
    Pour plus d'informations, tapez : "get-help Get-WmiObject -detailed".
    Pour obtenir des informations techniques, tapez : "get-help Get-WmiObject -full".
    Pour l'aide en ligne, tapez : "get-help Get-WmiObject -online"
```

La syntaxe de cette commande montre à quel point l'utilisation de celle-ci peut être compliquée et très simple à la fois. En réalité, cela dépend du niveau et du type d'information recherchés. La section suivante illustrera quelques exemples d'utilisation de cette cmdlet.

WMI en pratique

Le paramètre `-Class` de la cmdlet `Get-WmiObject` spécifie la classe interrogée. Il est donc un élément essentiel de l'utilisation de cette commande :

```
PS> Get-WmiObject -Class 'Win32_Service' -ComputerName 'Localhost'

>>ExitCode  : 0
>>Name      : AdobeARMservice
>>ProcessId : 1828
>>StartMode : Auto
>>State     : Running
>>Status    : OK

>>ExitCode  : 0
>>Name      : AdobeFlashPlayer
>>ProcessId : 0
>>StartMode : Manual
>>State     : Stopped
>>Status    : OK

>>ExitCode  : 0
>>Name      : AeLookupSvc
>>ProcessId : 0
>>StartMode : Manual
>>State     : Stopped
>>Status    : OK

>>ExitCode  : 1077
>>Name      : ALG
>>ProcessId : 0
>>StartMode : Manual
>>State     : Stopped
>>Status    : OK

>>ExitCode  : 1077
>>Name      : AppIDSvc
>>ProcessId : 0
>>StartMode : Manual
>>State     : Stopped
>>Status    : OK

>>ExitCode  : 0
>>Name      : Appinfo
>>ProcessId : 1064
>>StartMode : Manual
>>State     : Running
>>Status    : OK
…
```

Dans cet exemple, la classe `Win32_Service` représente les services sur une machine locale dont le nom est indiqué à l'aide du paramètre `-ComputerName`. Par défaut, si ce paramètre n'est pas mentionné, la commande est exécutée au niveau de la machine locale.

La commande suivante, par le biais de la classe `Win32_Bios`, liste des informations en lien avec le BIOS d'une machine locale :

```
PS> Get-WmiObject -Class 'Win32_Bios'

>>SMBIOSBIOSVersion : ERKT16AUS
>>Manufacturer      : LENOVO
>>Name              : ERKT16AUS
>>SerialNumber      : QS01107360
>>Version           : LENOVO - 1160
```

Dans la section dédiée à VBS, nous avons écrit un script listant les classes contenues dans l'espace de noms `root\cimv2`. Voici son équivalent en PowerShell :

```
PS> (Get-WmiObject -Namespace "root\cimv2" -List).Name

…
>>Win32_DeviceChangeEvent
>>Win32_SystemConfigurationChangeEvent
>>Win32_VolumeChangeEvent
>>MSFT_WMI_GenericNonCOMEvent
>>MSFT_NCProvEvent
>>MSFT_NCProvCancelQuery
>>MSFT_NCProvClientConnected
>>MSFT_NCProvNewQuery
>>MSFT_NCProvAccessCheck
>>Win32_SystemTrace
>>Win32_ProcessTrace
>>Win32_ProcessStartTrace
>>Win32_ProcessStopTrace
>>Win32_ThreadTrace
>>Win32_ThreadStartTrace
>>Win32_ThreadStopTrace
>>Win32_ModuleTrace
>>Win32_ModuleLoadTrace
>>Win32_PowerManagementEvent
>>Win32_ComputerSystemEvent
>>Win32_ComputerShutdownEvent
>>MSFT_SCMEvent
>>MSFT_SCMEventLogEvent
>>MSFT_NetSevereServiceFailed
>>MSFT_NetTransactInvalid
>>MSFT_NetServiceNotInteractive
>>MSFT_NetTakeOwnership
>>MSFT_NetServiceConfigBackoutFailed
>>MSFT_NetServiceShutdownFailed
```

```
>>MSFT_NetServiceStartHung
>>MSFT_NetServiceStopControlSuccess
>>MSFT_NetServiceSlowStartup
>>MSFT_NetCallToFunctionFailed
>>MSFT_NetBadAccount
>>MSFT_NetBadServiceState
...
```

Ici, le paramètre `-Namespace` spécifie l'espace de noms (`root\cimv2`) à partir duquel une requête WMI va être opérée et le paramètre `-List`, comme son nom l'indique, liste les classes qu'il contient.

Il est important de constater à travers ces exemples le gain de temps offert par cette cmdlet. PowerShell propose plus d'efficacité et de productivité que les autres langages de script dans l'utilisation et la manipulation du service WMI.

16

La gestion des services

D'une manière générale, une application s'articule autour d'un certain nombre de composants participant à son bon fonctionnement. Parmi eux, les services occupent une place majeure ; un service est un composant logiciel opérant en tâche de fond et accomplissant une partie du travail de l'application qui, elle, est le composant essentiel. Une gestion correcte des services préside, toutes proportions gardées, au bon déroulement d'une application.
Selon le langage de script utilisé, la manière dont les services sont gérés est différente. En effet, certains langages proposent un arsenal de gestion des services plus complet que d'autres. Dans ce chapitre, nous montrerons comment manipuler les services d'un système d'exploitation Linux ou Windows afin de mieux cerner les modes d'interaction offerts.

KSH

Selon le système d'exploitation utilisé, il y a plusieurs façons de gérer les services existants. En effet, chaque système (Red Hat, Debian, OpenBSD…) dispose d'un arsenal dont certaines composantes sont communes à plusieurs systèmes Unix/Linux et d'autres sont tout à fait spécifiques. Dans cette section, nous nous intéresserons à la gestion des services sous un système Red Hat.

Démarrer un service

L'outil principal sous une distribution Red Hat est la commande `service`, à laquelle on fournit en arguments le nom du service et l'action à exécuter, ici `start`. Essayons par exemple de démarrer Samba (`smb`).

```
$ service smb start
>>Starting SMB services:                                    [OK]
>>Starting NMB services:                                    [OK]
```

En sortie, les valeurs `[OK]` signifient que l'opération s'est bien passée. Au contraire, les valeurs `[NOK]` auraient signifié que le service Samba n'a pas pu être démarré.

Arrêter un service

L'arrêt d'un service particulier nécessite de donner l'argument `stop` à la commande `service` :

```
$ service smb stop
>>Shutting down SMB services:                               [OK]
>>Shutting down NMB services:                               [OK]
```

Dans cet exemple, Samba a été arrêté sans problèmes (`[OK]`).

Redémarrer un service

Grâce à l'argument `restart`, la commande `service` redémarre un service, c'est-à-dire l'arrête puis le relance :

```
$ service smb restart
>>Shutting down SMB services:                               [OK]
>>Shutting down NMB services:                               [OK]
>>Starting SMB services:                                    [OK]
>>Starting NMB services:                                    [OK]
```

D'autres arguments existent mais sont aussi fonction du service concerné. En effet, la commande `service` ne fait qu'exécuter des scripts agissant comme des services. Par conséquent, cela dépend aussi des modalités des actions autorisées sur ceux-ci.

Perl

Perl est disponible sur plusieurs plates-formes. Cette section sera dédiée à la gestion des services en lien avec un système d'exploitation Windows 8.

Lister les services

Pour lister les services dans un système d'exploitation Windows, WMI constitue une excellente option.

```
use Win32::OLE('in');

use constant wbemFlagReturnImmediately => 0x10;
use constant wbemFlagForwardOnly => 0x20;

@computers = (".");
foreach $computer (@computers) {
   print "\n";

   $objWMIService = Win32::OLE->GetObject("winmgmts:\\\\$computer\\root\\CIMV2") or
die "WMI connection failed.\n";
   $colItems = $objWMIService->ExecQuery("SELECT * FROM (1) Win32_Service", "WQL",
wbemFlagReturnImmediately | wbemFlagForwardOnly);

   print "List of services\n";
   print "----------------\n";
   (2) foreach $objItem (in $colItems) {
      print "$objItem->{Caption}\n";
   }
}
```

Ce script autorise une connexion au service WMI d'une machine locale. Une requête est effectuée sur la classe `Win32_Service` (1) pour obtenir la liste de tous ses services. L'instruction `foreach` (2) parcourt tous les éléments de cette collection pour en afficher le nom :

```
>>List of services
>>----------------
>>Adobe Acrobat Update Service
>>Adobe Flash Player Update Service
>>Expérience d'application
>>Service de la passerelle de la couche Application
>>Identité de l'application
>>Informations d'application
>>Préparation des applications
>>Service de déploiement AppX (AppXSVC)
>>Générateur de points de terminaison du service Audio Windows
>>Audio Windows
>>AVerRemote
>>AVerScheduleService
>>Alcohol Virtual Drive Auto-mount Service
>>Programme d'installation ActiveX (AxInstSV)
>>Bluetooth Driver Management Service
>>Service de chiffrement de lecteur BitLocker
>>Moteur de filtrage de base
>>Service de transfert intelligent en arrière-plan
>>Service d'infrastructure des tâches en arrière-plan
>>Explorateur d'ordinateurs
>>Service de prise en charge Bluetooth
>>Bluetooth Service
>>Propagation du certificat
...
```

Démarrer un service

Démarrer un service passe par l'appel d'une méthode de la classe `Win32_Service` nommée `StartService()`. En conséquence, une connexion au service WMI est là aussi nécessaire :

```
use Win32::OLE('in');
use Time::HiRes qw(usleep);

use constant wbemFlagReturnImmediately => 0x10;
use constant wbemFlagForwardOnly => 0x20;

@computers = (".");
foreach $computer (@computers) {
   print "\n";

   $objWMIService = Win32::OLE->GetObject("winmgmts:\\\\$computer\\root\\CIMV2") or
die "WMI connection failed.\n";
   $VDS = $objWMIService->ExecQuery("SELECT * FROM Win32_Service WHERE
Caption=\'Disque virtuel\'" (1), "WQL",
wbemFlagReturnImmediately | wbemFlagForwardOnly);

   foreach $objItem (in $VDS) {
      $objItem->StartService() (2);
      usleep(2000000);
      print "The state of VDS service is -> $objItem->{State}.";
   }
}
```

Démarrer un service particulier (ici, `Disque virtuel`) nécessite de préciser au niveau de la requête (1) les critères permettant d'agir sur l'objet recherché. Dans cet exemple, la requête est réalisée à l'aide du nom complet (`'Caption'`) du service. Une fois l'objet obtenu, il ne reste plus qu'à appeler la méthode `StartService()` (2). Enfin, la propriété `'State'` sert ici d'élément de contrôle afin de vérifier l'état du service :

```
>>The state of VDS service is -> Running.
```

Arrêter un service

La classe `Win32_Service` dispose d'une autre méthode, `StopService()` (1) :

```
use Win32::OLE('in');
use Time::HiRes qw(usleep);

use constant wbemFlagReturnImmediately => 0x10;
use constant wbemFlagForwardOnly => 0x20;

@computers = (".");
foreach $computer (@computers) {
   print "\n";
```

```
   $objWMIService = Win32::OLE->GetObject("winmgmts:\\\\$computer\\root\\CIMV2") or
die "WMI connection failed.\n";
   $VDS = $objWMIService->ExecQuery("SELECT * FROM Win32_Service WHERE
Caption=\'Disque virtuel\'", "WQL",
wbemFlagReturnImmediately | wbemFlagForwardOnly);

   foreach $objItem (in $VDS) {
      $objItem->StartService() (1);
      usleep(2000000);
      print "The state of VDS service is -> $objItem->{State}.";
   }
}
```

L'état du service passe au statut « arrêté » :

```
>>The state of VDS service is -> Stopped.
```

Python

Comme en Perl, la gestion des services sous Windows passe par l'utilisation d'un module. En outre, WMI constituera aussi dans cette section la voie d'accès à la manipulation des services.

Lister les services

Dans le chapitre précédent, nous avons brièvement évoqué le module `wmi` et la classe `Win32_Service`. Le script suivant sollicite le module `wmi` (1) pour obtenir une collection (2) des services d'un système d'exploitation Windows, dont on liste le nom et l'état (3) (il est bien sûr possible d'ajouter d'autres propriétés) :

```
import wmi (1)
obj = wmi.WMI()
services = obj.Win32_Service() (2)

print("List of services:")
print("-----------------\n")

for s in services:
    print(s.Caption, '->', s.State) (3)

>>List of services:
>>-----------------

>>Adobe Acrobat Update Service -> Running
>>Adobe Flash Player Update Service -> Stopped
>>Expérience d'application -> Stopped
>>Service de la passerelle de la couche Application -> Stopped
```

```
>>Identité de l'application -> Stopped
>>Informations d'application -> Running
>>Préparation des applications -> Stopped
>>Service de déploiement AppX (AppXSVC) -> Stopped
>>Audio Windows -> Running
>>AVerRemote -> Running
>>AVerScheduleService -> Running
>>Alcohol Virtual Drive Auto-mount Service -> Stopped
>>Programme d'installation ActiveX (AxInstSV) -> Stopped
>>Bluetooth Driver Management Service -> Stopped
>>Service de chiffrement de lecteur BitLocker -> Stopped
>>Moteur de filtrage de base -> Running
>>Service de transfert intelligent en arrière-plan -> Running
>>Service d'infrastructure des tâches en arrière-plan -> Running
>>Explorateur d'ordinateurs -> Running
>>Service de prise en charge Bluetooth -> Running
>>Bluetooth Service -> Running
>>Propagation du certificat -> Stopped
>>CyberLink Product - 2013/03/30 14:12:45 -> Stopped
>>Application système COM+ -> Stopped
>>Services de chiffrement -> Running
>>Dashboard Service -> Running
>>Lanceur de processus serveur DCOM -> Running
>>Optimiser les lecteurs -> Stopped
>>Service d'association de périphérique -> Running
>>Service d'installation de périphérique -> Stopped
>>Client DHCP -> Running
>>Client DNS -> Running
…
```

Démarrer un service

Comme en Perl, c'est la méthode `StartService()` (1) qui nous servira pour démarrer un service Windows :

```
import wmi
import time

obj = wmi.WMI()
vds = obj.Win32_Service(Caption='Disque Virtuel') (2)

for i in vds:
    i.StartService() (1)
    time.sleep(3)
    print(i.Caption, '->', i.State)
```

Il faut aussi noter la présence d'un filtre WMI (2) ciblant le service qui nous intéresse au lieu d'une collection toute entière. L'appel de la méthode `StartService()` (1) change de manière automatique l'état du service :

```
>>Disque virtuel -> Running
```

Arrêter un service

Le script précédent convient parfaitement si l'on veut arrêter un service Windows particulier, sauf qu'on utilise la méthode `StopService()` (1) :

```
import wmi
import time
obj = wmi.WMI()
vds = obj.Win32_Service(Caption='Disque Virtuel')

for i in vds:
    i.StopService() (1)
    time.sleep(3)
    print(i.Caption, '->', i.State)

>>Disque virtuel -> Stopped
```

VBS

Visual Basic Scripting est, nous l'avons vu dans le chapitre précédent, un des langages les plus adaptés à la manipulation de classes WMI. D'autres moyens sont possibles pour gérer les services d'un système d'exploitation Windows, mais il est évident que WMI est le plus efficace.

Lister les services

Dans la même logique que les autres langages, l'interrogation de la classe `Win32_Service` (1) nécessite une connexion au service (2) :

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:" _
   & "{impersonationLevel=impersonate}!\\" & strComputer & "\root\cimv2") (2)

Set colListOfServices = objWMIService.ExecQuery _
       ("Select * from Win32_Service") (1)

For Each objService in colListOfServices
   Wscript.Echo objService.Name (3)
Next
```

Il n'est pas nécessaire de faire appel à un module tiers. Ici, une simple requête WQL *(WMI Query Language)* (1) retourne une collection de services Windows à partir de laquelle une itération est requise (3) :

```
>>AdobeARMservice
>>AdobeFlashPlayerUpdateSvc
>>AeLookupSvc
>>ALG
>>AppIDSvc
>>Appinfo
>>AppReadiness
>>AppXSvc
>>AudioEndpointBuilder
>>Audiosrv
>>AVerRemote
>>AVerScheduleService
>>AxAutoMntSrv
>>AxInstSV
>>BcmBtRSupport
>>BDESVC
>>BFE
>>BITS
>>BrokerInfrastructure
>>Browser
>>bthserv
>>btwdins
>>CertPropSvc
>>CLKMSVC10_3A60B698
>>COMSysApp
>>CryptSvc
>>Dashboard Service
>>DcomLaunch
…
```

Démarrer un service

Le code suivant est la traduction en VBS des scripts précédents correspondant au démarrage d'un service Windows :

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:" _
    & "{impersonationLevel=impersonate}!\\" & strComputer & "\root\cimv2")

Set VDS = objWMIService.ExecQuery _
       ("Select * from Win32_Service where Caption='Disque Virtuel'") (1)

For Each objService in VDS
    objService.StartService()
    WScript.Sleep 2000
    WScript.Echo objService.Caption & " -> " & objService.State
Next
```

D'un langage de script à un autre, la requête WQL (1) est exactement la même ; le résultat obtenu est donc logiquement identique :

```
>>Disque virtuel -> Running
```

Arrêter un service

Pour arrêter un service Windows en VBS, reprenons le code précédent avec les modifications (1) qui s'imposent :

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:" _
   & "{impersonationLevel=impersonate}!\\" & strComputer & "\root\cimv2")

Set VDS = objWMIService.ExecQuery _
       ("Select * from Win32_Service where Caption='Disque Virtuel'")

For Each objService in VDS
   objService.StopService() (1)
   WScript.Sleep 2000
   WScript.Echo objService.Caption & " -> " & objService.State
Next

>>Disque virtuel -> Stopped
```

Windows PowerShell

En PowerShell, il existe une multitude de possibilités de gérer les services Windows. Cette section sera consacrée à la gestion des services Windows via l'utilisation de cmdlets.

Lister les services

La cmdlet `Get-Service` fournit une liste de services Windows, quel que soit leur état :

```
PS> Get-Service

>>Status   Name               DisplayName
>>------   ----               -----------
>>Running  AdobeARMservice    Adobe Acrobat Update Service
>>Stopped  AdobeFlashPlaye... Adobe Flash Player Update Service
>>Stopped  AeLookupSvc        Expérience d'application
>>Stopped  ALG                Service de la passerelle de la couc...
>>Stopped  AppIDSvc           Identité de l'application
>>Running  Appinfo            Informations d'application
>>Stopped  AppReadiness       Préparation des applications
```

```
>>Stopped  AppXSvc            Service de déploiement AppX (AppXSVC)
>>Running  AudioEndpointBu... Générateur de points de terminaison...
>>Running  Audiosrv           Audio Windows
>>Running  AVerRemote         AVerRemote
>>Running  AVerScheduleSer... AVerScheduleService
>>Stopped  AxAutoMntSrv       Alcohol Virtual Drive Auto-mount Se...
>>Stopped  AxInstSV           Programme d'installation ActiveX (A...
>>Stopped  BcmBtRSupport      Bluetooth Driver Management Service
>>Stopped  BDESVC             Service de chiffrement de lecteur B...
>>Running  BFE                Moteur de filtrage de base
>>Running  BITS               Service de transfert intelligent en...
>>Running  BrokerInfrastru... Service d'infrastructure des tâches...
>>Running  Browser            Explorateur d'ordinateurs
>>Running  bthserv            Service de prise en charge Bluetooth
>>Running  btwdins            Bluetooth Service
>>Stopped  CertPropSvc        Propagation du certificat
>>Stopped  CLKMSVC10_3A60B698 CyberLink Product - 2013/03/30 14:1...
>>Stopped  COMSysApp          Application système COM+
>>Running  CryptSvc           Services de chiffrement
…
```

Toute seule, la cmdlet `Get-Service` retourne une liste de services, mais l'utilisation du paramètre `-Name` cible un ou plusieurs services au lieu de tous :

```
PS> Get-Service -Name 'Disque Virtuel'

>>Status   Name               DisplayName
>>------   ----               -----------
>>Running  vds                Disque Virtuel
```

Comme pour une majorité de cmdlets PowerShell, le paramètre `-ComputerName` autorise des requêtes sur des machines distantes :

```
PS> Get-Service -Name 'Disque Virtuel' -ComputerName 'MLDEVX5'

>>Status   Name               DisplayName
>>------   ----               -----------
>>Stopped  vds                Disque Virtuel
```

Démarrer un service

La cmdlet `Start-Service` démarre un service Windows qui est arrêté :

```
PS> Start-Service -Name 'Disque Virtuel' -PassThru

>>Status   Name               DisplayName
>>------   ----               -----------
>>Running  vds                Disque Virtuel
```

Le paramètre `-PassThru` indique à la cmdlet de retourner l'objet représentant le service.

Arrêter un service

Dans une logique similaire, la cmdlet `Stop-Service` arrête un service Windows en cours d'exécution :

```
PS> Stop-Service -Name 'Disque Virtuel' -PassThru

>>Status   Name               DisplayName
>>------   ----               -----------
>>Stopped  vds                Disque Virtuel
```

En ce qui concerne la gestion des services en PowerShell, d'autres cmdlets existent, mais celles que nous venons de voir suffisent amplement pour interagir avec les services Windows.

17

La gestion des journaux d'événements

Dans chaque système d'exploitation, des événements se produisent systématiquement, et ce, à tous les niveaux. Selon les couches qui composent un système d'exploitation, plusieurs types d'événements différents dans leur source se croisent au sein d'un même corpus appelé journal. *En outre, d'un système d'exploitation à un autre, la manière dont les événements sont administrés varie, ce qui implique une compréhension globale dans la façon dont ils doivent être gérés.*
Que l'on administre un système Windows ou Linux, chaque langage de script fournit un certain nombre d'outils qui leur sont propres. Ce chapitre esquissera de manière élémentaire comment s'articule une gestion des journaux d'événements saine et claire. Nous verrons donc comment agir sur ces sources d'informations d'une importance capitale dans le cadre d'une administration système efficace.

KSH

Dans un système Unix/Linux, il existe de nombreux outils liés à la gestion des journaux d'événements et il faudrait au moins deux chapitres pour évoquer ne serait-ce que les fondamentaux, car l'administration des journaux d'événements est un travail à temps plein à lui tout seul ! En conséquence, cette section mettra en évidence seulement quelques commandes shell fondamentales constituant une première étape dans la gestion des journaux d'événements.

Les commandes head et tail

Un journal d'événements est, quel que soit le système d'exploitation, un condensé d'informations extrêmement riches. Parmi elles, certaines sont plus importantes que d'autres ; de même, il est parfois nécessaire de consulter des événements plus récents que d'autres et vice versa.

Dans cette perspective, des commandes shell réalisent différents types d'actions selon les objectifs recherchés. En ce qui concerne les événements anciens, la commande `head` est un excellent recours, puisqu'elle consiste à afficher en sortie les premières lignes d'un fichier. Par exemple, la ligne de commande suivante affiche les quinze premières lignes du fichier `/var/log/messages` :

```
# head -15 /var/log/messages
>>Apr 20 00:12:38 debiandev rsyslogd: [origin software="rsyslogd" swVersion="5.8.11"
x-pid="1912" x-info="http://www.rsyslog.com"] rsyslogd was HUPed
>>Apr 20 01:21:07 debiandev shutdown[4278]: shutting down for system halt
>>Apr 20 01:21:08 debiandev kernel: [ 6088.797244] vboxvideo: disagrees about version
of symbol drm_pci_init
>>Apr 20 01:21:08 debiandev kernel: [ 6088.797247] vboxvideo: Unknown symbol
drm_pci_init (err -22)
>>Apr 20 01:21:08 debiandev kernel: [ 6088.797250] vboxvideo: disagrees about version
of symbol drm_vblank_init
>>Apr 20 01:21:08 debiandev kernel: [ 6088.797251] vboxvideo: Unknown symbol
drm_vblank_init (err -22)
>>Apr 20 01:21:08 debiandev kernel: [ 6088.797253] vboxvideo: disagrees about version
of symbol drm_pci_exit
>>Apr 20 01:21:08 debiandev kernel: [ 6088.797254] vboxvideo: Unknown symbol
drm_pci_exit (err -22)
>>Apr 20 01:21:08 debiandev kernel: [ 6088.797256] vboxvideo: disagrees about version
of symbol drm_core_reclaim_buffers
>>Apr 20 01:21:08 debiandev kernel: [ 6088.797257] vboxvideo: Unknown symbol
drm_core_reclaim_buffers (err -22)
>>Apr 20 01:21:09 debiandev kernel: [ 6089.722579] colord-sane[3338]: segfault at 4
ip b573d472 sp b5710fb8 error 6 in libdbus-1.so.3.7.2[b5712000+49000]
>>Apr 25 05:01:33 debiandev kernel: imklog 5.8.11, log source = /proc/kmsg started.
>>Apr 25 05:01:33 debiandev rsyslogd: [origin software="rsyslogd" swVersion="5.8.11"
x-pid="1890" x-info="http://www.rsyslog.com"] start
>>Apr 25 05:01:33 debiandev kernel: [    0.000000] Initializing cgroup subsys cpuset
>>Apr 25 05:01:33 debiandev kernel: [    0.000000] Initializing cgroup subsys cpu
```

Les quinze événements (`-15`) de cet exemple sont les plus anciens. Pour afficher les quinze plus récents, optons plutôt pour la commande `tail` :

```
# tail -n 15 /var/log/messages
>>May 6 17:14:35 debiandev kernel: [   15.516821] ppdev: user-space parallel port
driver
>>May 6 17:14:35 debiandev kernel: [   15.852612] [drm] Initialized drm 1.1.0
20060810
```

```
>>May 6 17:14:35 debiandev kernel: [   15.901715] vboxvideo: disagrees about version
of symbol drm_pci_init
>>May 6 17:14:35 debiandev kernel: [   15.901719] vboxvideo: Unknown symbol
drm_pci_init (err -22)
>>May 6 17:14:35 debiandev kernel: [   15.901723] vboxvideo: disagrees about version
of symbol drm_vblank_init
>>May 6 17:14:35 debiandev kernel: [   15.901726] vboxvideo: Unknown symbol
drm_vblank_init (err -22)
>>May 6 17:14:35 debiandev kernel: [   15.901729] vboxvideo: disagrees about version
of symbol drm_pci_exit
>>May 6 17:14:35 debiandev kernel: [   15.901731] vboxvideo: Unknown symbol
drm_pci_exit (err -22)
>>May 6 17:14:35 debiandev kernel: [   15.901734] vboxvideo: disagrees about version
of symbol drm_core_reclaim_buffers
>>May 6 17:14:35 debiandev kernel: [   15.901736] vboxvideo: Unknown symbol
drm_core_reclaim_buffers (err -22)
>>May 6 17:14:45 debiandev pulseaudio[3076]: [pulseaudio] alsa-util.c: Disabling
timer-based scheduling because running inside a VM.
>>May 6 17:14:45 debiandev pulseaudio[3076]: [pulseaudio] alsa-util.c: Disabling
timer-based scheduling because running inside a VM.
>>May 6 17:14:57 debiandev pulseaudio[3224]: [pulseaudio] alsa-util.c: Disabling
timer-based scheduling because running inside a VM.
>>May 6 17:14:57 debiandev pulseaudio[3224]: [pulseaudio] alsa-util.c: Disabling
timer-based scheduling because running inside a VM.
>>May 6 17:43:35 debiandev rsyslogd: [origin software="rsyslogd" swVersion="5.8.11"
x-pid="1917" x-info="http://www.rsyslog.com"] rsyslogd was HUPed
```

Le nombre de lignes à extraire du fichier `/var/log/messages` est spécifié à l'aide du paramètre `-n`. De cette façon, la vue offerte concentre uniquement les événements les plus récents.

Les commandes `head` et `tail` sont très utilisées dans le cas d'un diagnostic système. Savoir s'en servir est donc primordial pour une meilleure efficacité dans la lecture des journaux d'événements et pour gagner un temps considérable à long terme.

La commande logger

Un autre outil très utile est la commande `logger`. Elle permet d'écrire des événements dans le journal système et est donc très importante :

```
# logger System rebooted [administration tasks] 23:15
```

Cette commande insère dans le journal `/var/log/messages` une ligne indiquant un redémarrage système suite à des tâches d'administration. La ligne figure effectivement bien au sein du journal système :

```
# tail -n 1 /var/log/messages
>>May 6 23:15:39 debiandev scripting: System rebooted [administration tasks] 23:15
```

Ces opérations forment une première étape dans la gestion des logs mais, nous le rappelons, gérer les journaux d'événements dans un système Unix/Linux est presque un métier à part entière.

Perl

Lister les événements d'un journal

En Perl sous Windows, nous utilisons la classe `Win32_NTLogEvent` (1) pour lister les événements d'un journal :

```
use Win32::OLE('in');

use constant wbemFlagReturnImmediately => 0x10;
use constant wbemFlagForwardOnly => 0x20;

@computers = (".");
foreach $computer (@computers) {
   print "\n";

   $objWMIService = Win32::OLE->GetObject("winmgmts:\\\\$computer\\root\\CIMV2") or
die "WMI connection failed.\n";
  $LoggedEvents = $objWMIService->ExecQuery("SELECT * FROM Win32_NTLogEvent (1) WHERE
Logfile = \'Application\' (2) ", "WQL",
wbemFlagReturnImmediately | wbemFlagForwardOnly);

   foreach my $objItem (in $LoggedEvents) {
      print "Category: $objItem->{Category}\n"; (3)
      print "CategoryString: $objItem->{CategoryString}\n";
      print "ComputerName: $objItem->{ComputerName}\n";
      print "EventCode: $objItem->{EventCode}\n";
      print "EventIdentifier: $objItem->{EventIdentifier}\n";
      print "EventType: $objItem->{EventType}\n";
      print "Message: $objItem->{Message}\n";
      print "SourceName: $objItem->{SourceName}\n";
      print "TimeGenerated: $objItem->{TimeGenerated}\n";
      print "TimeWritten: $objItem->{TimeWritten}\n";
      print "Type: $objItem->{Type}\n";
      print "User: $objItem->{User}\n";
      print "\n";
   }
}
```

La classe `Win32_NTLogEvent` (1) permet de collecter des événements appartenant ou non à un journal spécifique, ici le journal d'application (2). Celui-ci contient un certain nombre d'événements de sources différentes et d'un niveau d'importance variable.

Les propriétés (3) collectées sont les plus significatives, mais il en existe beaucoup d'autres.

```
>>Category: 1
>>CategoryString: Général
>>ComputerName: A720
>>EventCode: 102
>>EventIdentifier: 102
>>EventType: 3
>>Message: svchost (1608) Instance: Le moteur de la base de données (6.03.9600.0000)
a démarré une nouvelle instance (0).
>>SourceName: ESENT
>>TimeGenerated: 20140323143330.000000-000
>>TimeWritten: 20140323143330.000000-000
>>Type: Information
>>User:

>>Category: 1
>>CategoryString: Général
>>ComputerName: A720
>>EventCode: 103
>>EventIdentifier: 103
>>EventType: 3
>>Message: svchost (684) Instance: Le moteur de base de données a arrêté l'instance
(0).
>>Arrêt incorrect : 0
>>Séquence de temporisation interne : [1] 0.000, [2] 0.000, [3] 0.000, [4] 0.000, [5]
0.000, [6] 0.000, [7] 0.000, [8] 0.000, [9] 0.016, [10] 0.000, [11] 0.000, [12]
0.000, [13] 0.000, [14] 0.000, [15] 0.000.
>>SourceName: ESENT
>>TimeGenerated: 20140323142819.000000-000
>>TimeWritten: 20140323142819.000000-000
>>Type: Information
>>User:

>>Category: 1
>>CategoryString: Général
>>ComputerName: A720
>>EventCode: 327
>>EventIdentifier: 327
>>EventType: 3
>>Message: svchost (684) Instance: Le moteur de base de données a détaché une base de
données (1, C:\ProgramData\Microsoft\Windows\AppRepository\PackageRepository.edb).
(Durée = 0 secondes)
>>Séquence de temporisation interne : [1] 0.000, [2] 0.000, [3] 0.000, [4] 0.000, [5]
0.000, [6] 0.000, [7] 0.000, [8] 0.000, [9] 0.000, [10] 0.313, [11] 0.015, [12]
0.000.
>>Cache enregistré : 0 0
>>SourceName: ESENT
>>TimeGenerated: 20140323142819.000000-000
>>TimeWritten: 20140323142819.000000-000
>>Type: Information
>>User:
...
```

Si on envisage de créer des rapports basés sur des diagnostics système, ces informations sont d'une importance capitale. Bien reconnaître la nature des informations contenues dans chaque propriété est donc une étape fondamentale à toute lecture claire et logique.

Lister les événements correspondant à une période spécifique

Parfois, il est nécessaire de collecter des événements uniquement à partir de critères bien définis. Par exemple, comme les informations enregistrées dans les journaux d'événements sont souvent très nombreuses, limiter la recherche à une période précise fait gagner en temps et en efficacité :

```
use Win32::OLE('in');

use constant wbemFlagReturnImmediately => 0x10;
use constant wbemFlagForwardOnly => 0x20;

$ds = "20140423",
$de = "20140427",
$format = "000000.000000-000";
$DateStart = $ds . $format;
$DateEnd = $de . $format;

@computers = (".");
foreach $computer (@computers) {
   print "\n";

   $objWMIService = Win32::OLE->GetObject("winmgmts:\\\\$computer\\root\\CIMV2") or
die "WMI connection failed.\n";
   $LoggedEvents = $objWMIService->ExecQuery("SELECT * FROM Win32_NTLogEvent WHERE
TimeWritten >= \'$DateStart\' and TimeWritten < \'$DateEnd\' (1)", "WQL",
wbemFlagReturnImmediately | wbemFlagForwardOnly);

   foreach my $objItem (in $LoggedEvents) {
      print "Category: $objItem->{Category}\n";
      print "CategoryString: $objItem->{CategoryString}\n";
      print "ComputerName: $objItem->{ComputerName}\n";
      print "EventCode: $objItem->{EventCode}\n";
      print "EventIdentifier: $objItem->{EventIdentifier}\n";
      print "EventType: $objItem->{EventType}\n";
      print "Message: $objItem->{Message}\n";
      print "SourceName: $objItem->{SourceName}\n";
      print "TimeGenerated: $objItem->{TimeGenerated}\n";
      print "TimeWritten: $objItem->{TimeWritten}\n";
      print "Type: $objItem->{Type}\n";
      print "Log: $objItem->{LogFile}\n"; (2)
      print "\n";
   }
}
```

Ce script, après s'être connecté au service WMI d'une machine locale, interroge la classe `Win32_NTLogEvent` afin de collecter des événements appartenant à différents journaux, et ce, sur

la base d'une période donnée (1). En effet, la clause `WHERE` utilise la propriété `TimeWritten` pour définir une période commençant au `23/04/2014` et finissant au `27/04/2014`. De cette façon, seuls les événements appartenant à cette période seront affichés en sortie par Perl :

```
…
>>Category: 0
>>CategoryString:
>>ComputerName: A720
>>EventCode: 16
>>EventIdentifier: 16
>>EventType: 3
>>Message: L'historique des accès de la ruche
\??\C:\Users\Kais\AppData\Local\Packages\winstore_cw5n1h2txyewy\Settings\settings.da
t a été effacé, mettant à jour 4 clés et créant 1 pages modifiées.
>>SourceName: Microsoft-Windows-Kernel-General
>>TimeGenerated: 20140424180700.952034-000
>>TimeWritten: 20140424180700.952034-000
>>Type: Information
>>Log: System

>>Category: 0
>>CategoryString:
>>ComputerName: A720
>>EventCode: 16
>>EventIdentifier: 16
>>EventType: 3
>>Message: L'historique des accès de la ruche
\??\C:\Users\Kais\AppData\Local\Packages\windows.immersivecontrolpanel_cw5n1h2txyewy
\Settings\settings.dat a été effacé, mettant à jour 2 clés et créant 1 pages
modifiées.
>>SourceName: Microsoft-Windows-Kernel-General
>>TimeGenerated: 20140424180700.577035-000
>>TimeWritten: 20140424180700.577035-000
>>Type: Information
>>Log: System

>>Category: 0
>>CategoryString:
>>ComputerName: A720
>>EventCode: 16
>>EventIdentifier: 16
>>EventType: 3
>>Message: L'historique des accès de la ruche
\??\C:\Users\Kais\AppData\Local\Packages\rara.com.rara.com_2tghmx54nqzjm\Settings\se
ttings.dat a été effacé, mettant à jour 2 clés et créant 1 pages modifiées.
>>SourceName: Microsoft-Windows-Kernel-General
>>TimeGenerated: 20140424180659.983238-000
>>TimeWritten: 20140424180659.983238-000
>>Type: Information
>>Log: System
...
```

Cependant, comme les événements affichés sont de sources différentes et proviennent de journaux différents, il est fondamental de préciser leur origine (2) pour mieux les identifier. Si les informations sont lues de manière claire, leur traitement n'en sera que plus efficace.

Python

Nous nous sommes jusqu'à présent attachés aux contenus des journaux d'événements. Dans cette partie, essayons de nous intéresser aux journaux eux-mêmes, en tant qu'objets statistiques.

Obtenir le nombre des événements contenus dans un journal

Dans une logique d'administration des journaux d'événements bien menée, il est très important de connaître certaines informations, comme le nombre d'événements d'un journal afin de mieux contrôler le nombre d'entrées ou la taille d'un journal pour mieux maîtriser l'espace occupé (et c'est ce que nous verrons dans la section suivante).

Le script suivant sollicite la classe `Win32_NTEventLogFile` (1) pour obtenir le nombre d'événements du journal `System` :

```
import wmi
obj = wmi.WMI()
LogSystem = obj.Win32_NTEventLogFile(LogFileName='System') (1)

for Log in LogSystem:
    print('The number of entries is ->', Log.NumberOfRecords) (2)
```

La propriété nous donnant cette information se nomme `NumberOfRecords` (2) :

```
>>The number of entries is -> 48015
```

Obtenir la taille d'un journal d'événements

Pour obtenir la taille d'un journal d'événements, la propriété est `FileSize` (1) :

```
import wmi
obj = wmi.WMI()
LogSystem = obj.Win32_NTEventLogFile(LogFileName='System')

for Log in LogSystem:
    print('The size is ->', Log.FileSize, 'bytes') (1)
```

La taille est représentée en `bytes` ; sur cette base, et selon le contexte, des opérations de conversion peuvent être menées :

```
>>The size is -> 20975616 bytes
```

VBS

Les journaux d'événements étant des fichiers très volumineux, il est souvent nécessaire de mettre en place une politique de rotation des logs ou de gestion de l'espace occupé. Les opérations de sauvegarde et d'effacement de journaux sont des tâches courantes et qui, de plus, peuvent être automatisées.

Sauvegarder un journal d'événements

La méthode dont nous aurons besoin pour la sauvegarde se nomme `BackupEventLog()`. Elle appartient à la classe `Win32_NTEventLogFile` et permet de réaliser une copie d'un journal d'événements (1) :

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:" _
    & "{impersonationLevel=impersonate,(Backup)}!\\" & _
        strComputer & "\root\cimv2")

Set colLogFiles = objWMIService.ExecQuery _
    ("Select * from Win32_NTEventLogFile where LogFileName='Application'")

For Each objLogfile in colLogFiles
    errBackupLog = objLogFile.BackupEventLog("C:\Backup\application_bak.evt") (1)
    If errBackupLog <> 0 (2) Then
        Wscript.Echo "The Application event log could not be backed up."
    End If
Next
```

L'argument passé à la méthode `BackupEventLog()` doit être un chemin où résidera la sauvegarde, laquelle aura l'extension `.evt`.

L'exécution de `BackupEventLog()` retourne une valeur (2) qui, si elle est autre que `0`, indique une erreur qu'il faudra essayer d'identifier. Enfin, aucune sortie n'est affichée si l'opération de sauvegarde s'est bien déroulée.

Effacer un journal d'événements

Après avoir sauvegardé un journal d'événements, il est commun d'en effacer le contenu afin de ne pas y conserver d'éléments trop anciens. La classe `Win32_NTEventLogFile` contient pour ce faire une autre méthode, `ClearEventLog()` (1) :

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:" _
    & "{impersonationLevel=impersonate,(Backup)}!\\" & _
        strComputer & "\root\cimv2")
```

```
Set colLogFiles = objWMIService.ExecQuery _
    ("Select * from Win32_NTEventLogFile where LogFileName='Application'")

For Each objLogfile in colLogFiles
    errClearLog = objLogFile.ClearEventLog() (1)
    If errClearLog <> 0 Then
        Wscript.Echo "The Application event log could not be cleared."
    End If
Next
```

De même que pour `BackupEventLog()`, aucune sortie n'est affichée si l'opération s'est bien déroulée.

Les méthodes `BackupEventLog()` et `ClearEventLog()` assurent des opérations de base quant à la gestion des logs sous Windows. Pour plus d'informations, le mieux est de consulter la documentation MSDN concernant la classe `Win32_NTEventLogFile`.

> **IMPORTANT L'action sur les journaux d'événements**
>
> Agir sur certains journaux d'événements demande un niveau de privilèges suffisant pour mener à bien les opérations de sauvegarde et d'effacement.

Windows PowerShell

Gérer les journaux d'événements avec Windows PowerShell est plus aisé qu'avec les autres langages de script, car il prend quasiment en charge toute la logique nécessaire.

Lister les journaux d'événements

En PowerShell, il n'est pas nécessaire de recourir à de la programmation système pour manipuler les journaux d'événements. Au contraire, il existe de nombreuses cmdlets renfermant un certain niveau d'abstraction, d'où un gain de temps. Parmi ces cmdlets, `Get-EventLog` collecte les événements d'un journal particulier ou liste l'ensemble des journaux d'événements :

```
PS> Get-EventLog -List

>> Max(K) Retain OverflowAction       Entries Log
>> ------ ------ --------------       ------- ---
>> 20 480      0 OverwriteAsNeeded      18 327 Application
>> 20 480      0 OverwriteAsNeeded           0 HardwareEvents
>>    512      7 OverwriteOlder              0 Internet Explorer
>> 20 480      0 OverwriteAsNeeded           0 Key Management Service
>>    128      0 OverwriteAsNeeded          34 OAlerts
>> 20 480      0 OverwriteAsNeeded      29 524 Security
>> 20 480      0 OverwriteAsNeeded      47 978 System
>> 15 360      0 OverwriteAsNeeded      10 376 Windows PowerShell
```

Lorsque le paramètre `-List` est utilisé, la cmdlet `Get-EventLog` retourne une liste de journaux d'événements, mais en ne retenant que des informations statistiques comme le nombre d'entrées d'un journal ou son nom.

Pour obtenir les entrées elles-mêmes, il faut utiliser le paramètre `-LogName` :

```
PS> Get-EventLog -LogName 'Windows PowerShell'

>>Index Time          EntryType   Source      InstanceID Message
>>----- ----          ---------   ------      ---------- -------
>>10376 mai 11 16:56 Information PowerShell         400 L'état du moteur
>>10375 mai 11 16:56 Information PowerShell         600 Le fournisseur «
>>10374 mai 11 16:56 Information PowerShell         600 Le fournisseur «
>>10373 mai 11 16:56 Information PowerShell         600 Le fournisseur «
>>10372 mai 11 16:56 Information PowerShell         600 Le fournisseur «
>>10371 mai 11 16:56 Information PowerShell         600 Le fournisseur «
>>10370 mai 11 16:56 Information PowerShell         600 Le fournisseur «
>>10369 mai 11 16:56 Information PowerShell         403 L'état du moteur
>>10368 mai 11 16:56 Information PowerShell         400 L'état du moteur
>>10367 mai 11 16:56 Information PowerShell         600 Le fournisseur «
>>10366 mai 11 16:56 Information PowerShell         600 Le fournisseur «
>>10365 mai 11 16:56 Information PowerShell         600 Le fournisseur «
...
```

Le nombre d'informations étant élevé, l'affichage est rendu partiel par Windows PowerShell. En effet, si l'on observe l'affichage de manière attentive, on s'aperçoit que la colonne `Message` n'est pas affichée de manière complète. En conséquence, l'exploitation des informations est quasi impossible en l'état.

Il faut donc adopter un mode de lecture plus ciblé. Par exemple, dans le journal Windows PowerShell, obtenons l'événement le plus récent à l'aide du paramètre `-Newest`, puis envoyons le résultat à la cmdlet `Format-List` afin que cette dernière reformate les données reçues sous la forme d'une liste :

```
PS> Get-EventLog -LogName 'Windows PowerShell' -Newest 1 | Format-List

>>Index              : 10376
>>EntryType          : Information
>>InstanceId         : 400
>>Message            : L'état du moteur passe de None à Available.

>>                     Détails :
>>                         NewEngineState=Available
>>                         PreviousEngineState=None
>>
>>                         SequenceNumber=13
>>
>>                         HostName=ConsoleHost
>>                         HostVersion=4.0
>>                         HostId=e386f654-b196-4ecc-9e0f-10d6748ba53c
```

```
>>                          EngineVersion=4.0
>>                          RunspaceId=8ec8a3b3-4667-4117-b258-e0f74f113510
>>                          PipelineId=
>>                          CommandName=
>>                          CommandType=
>>                          ScriptName=
>>                          CommandPath=
>>                          CommandLine=
>>Category           : Cycle de vie du moteur
>>CategoryNumber     : 4
>>ReplacementStrings : {Available, None,     NewEngineState=Available
>>                          PreviousEngineState=None

>>                          SequenceNumber=13

>>                          HostName=ConsoleHost
>>                          HostVersion=4.0
>>                          HostId=e386f654-b196-4ecc-9e0f-10d6748ba53c
>>                          EngineVersion=4.0
>>                          RunspaceId=8ec8a3b3-4667-4117-b258-e0f74f113510
>>                          PipelineId=
>>                          CommandName=
>>                          CommandType=
>>                          ScriptName=
>>                          CommandPath=
>>                          CommandLine=}
>>Source             : PowerShell
>>TimeGenerated      : 11/05/2014 16:56:29
>>TimeWritten        : 11/05/2014 16:56:29
>>UserName           :
```

Les informations sont à présent clairement lisibles, donc immédiatement exploitables ! Ici, tout n'a été qu'une question de format des données.

Par défaut, la cmdlet `Get-EventLog` retourne tous les types d'événements, sans distinction. Le paramètre `-EntryType` sert à préciser un type à retourner :

```
PS> Get-EventLog -LogName 'Windows PowerShell' -EntryType 'Information'

>>Index Time          EntryType   Source     InstanceID Message
>>----- ----          ---------   ------     ---------- -------
>>10376 mai 11 16:56   Information PowerShell        400 L'état du moteur
>>10375 mai 11 16:56   Information PowerShell        600 Le fournisseur «
>>10374 mai 11 16:56   Information PowerShell        600 Le fournisseur «
>>10373 mai 11 16:56   Information PowerShell        600 Le fournisseur «
>>10372 mai 11 16:56   Information PowerShell        600 Le fournisseur «
>>10371 mai 11 16:56   Information PowerShell        600 Le fournisseur «
>>10370 mai 11 16:56   Information PowerShell        600 Le fournisseur «
>>10369 mai 11 16:56   Information PowerShell        403 L'état du moteur
```

```
>>10368 mai 11 16:56   Information PowerShell        400 L'état du moteur
>>10367 mai 11 16:56   Information PowerShell        600 Le fournisseur «
>>10366 mai 11 16:56   Information PowerShell        600 Le fournisseur «
>>10365 mai 11 16:56   Information PowerShell        600 Le fournisseur «
>>10364 mai 11 16:56   Information PowerShell        600 Le fournisseur «
>>10363 mai 11 16:56   Information PowerShell        600 Le fournisseur «
...
```

Le filtrage peut aussi se faire sur la base d'un identifiant d'événement, spécifié à l'aide du paramètre `-InstanceId` :

```
PS> Get-EventLog -LogName 'Windows PowerShell' -EntryType 'Information' -InstanceId
600

>>Index Time          EntryType   Source     InstanceID Message
>>----- ----          ---------   ------     ---------- -------
>>10375 mai 11 16:56   Information PowerShell        600 Le fournisseur «
>>10374 mai 11 16:56   Information PowerShell        600 Le fournisseur «
>>10373 mai 11 16:56   Information PowerShell        600 Le fournisseur «
>>10372 mai 11 16:56   Information PowerShell        600 Le fournisseur «
>>10371 mai 11 16:56   Information PowerShell        600 Le fournisseur «
>>10370 mai 11 16:56   Information PowerShell        600 Le fournisseur «
>>10367 mai 11 16:56   Information PowerShell        600 Le fournisseur «
>>10366 mai 11 16:56   Information PowerShell        600 Le fournisseur «
>>10365 mai 11 16:56   Information PowerShell        600 Le fournisseur «
>>10364 mai 11 16:56   Information PowerShell        600 Le fournisseur «
>>10363 mai 11 16:56   Information PowerShell        600 Le fournisseur «
>>10362 mai 11 16:56   Information PowerShell        600 Le fournisseur «
>>10359 mai 10 21:10   Information PowerShell        600 Le fournisseur «
>>10358 mai 10 21:10   Information PowerShell        600 Le fournisseur «
...
```

D'autres paramètres en lien avec la cmdlet `Get-EventLog` existent et se complètent d'ailleurs très bien, mais ceux que nous venons de découvrir suffisent à démontrer qu'une seule ligne de commande produit à elle seule une quantité d'informations qu'un autre langage de script ne produirait qu'à la seule condition d'une programmation fine et plus longue.

Effacer et supprimer les journaux d'événements

Il existe une distinction très claire entre effacer un journal d'événements (le vider) et le supprimer.

En PowerShell, la cmdlet `Clear-EventLog` efface toutes les entrées d'un journal, sans produire aucune sortie :

```
PS> Clear-EventLog -LogName 'Windows PowerShell'
```

La cmdlet `Remove-EventLog` supprime, quant à elle, le journal dans sa totalité, également sans sortie :

```
PS> Remove-EventLog -LogName 'Windows PowerShell'
```

Son comportement peut toutefois légèrement différer en ne supprimant qu'une ou plusieurs source(s).

18

La gestion d'une interface réseau

Un réseau informatique est composé d'éléments dits interconnectés, c'est-à-dire disposant d'une interface réseau et reliés à d'autres éléments dans le but de transmettre des données. Il est essentiel de connaître clairement comment s'articule la configuration d'une interface réseau, qu'elle soit filaire ou sans fil.

Dans ce chapitre, nous verrons comment obtenir et manipuler des informations de base concernant la gestion d'une interface réseau : adresse IP, masque de sous-réseau, passerelle par défaut, serveur DNS.

Ces informations constituent le socle fondamental de la structuration d'une interface réseau ; leur mauvaise définition irait même jusqu'à remettre en question le bon fonctionnement d'un réseau dans son ensemble.

KSH

En Korn Shell, la gestion d'une interface réseau s'effectue à l'aide de commandes dédiées. Parmi les plus utilisées se trouve la commande `ifconfig`.

Obtenir une configuration IP

Obtenir des informations en lien avec une ou des interface(s) réseau est possible à l'aide de la commande `ifconfig`. Elle est utilisée pour configurer des interfaces réseau existantes dans le noyau. Elle a donc un rôle très important dans un environnement Unix/Linux.

Utilisée toute seule, la commande `ifconfig` affiche l'état des configurations actives :

```
root@debiandev:~# ifconfig
>>eth0 (1)    Link encap:Ethernet HWaddr 08:00:27:1a:73:c7 (2)
>>            inet addr:10.0.2.4 (3) Bcast:10.0.2.255 Mask:255.255.255.0 (4)
>>            inet6 addr: fe80::a00:27ff:fe1a:73c7/64 Scope:Link
>>            UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
>>            RX packets:12 errors:0 dropped:0 overruns:0 frame:0
>>            TX packets:118 errors:0 dropped:0 overruns:0 carrier:0
>>            collisions:0 txqueuelen:1000
>>            RX bytes:2517 (2.4 KiB) TX bytes:16305 (15.9 KiB)

>>lo         Link encap:Local Loopback
>>            inet addr:127.0.0.1 Mask:255.0.0.0
>>            inet6 addr: ::1/128 Scope:Host
>>            UP LOOPBACK RUNNING MTU:16436 Metric:1
>>            RX packets:8 errors:0 dropped:0 overruns:0 frame:0
>>            TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
>>            collisions:0 txqueuelen:0
>>            RX bytes:480 (480.0 B) TX bytes:480 (480.0 B)
```

On observe qu'une interface réseau est identifiée par un nom (1), une adresse MAC (2), une adresse IP (3), ainsi qu'un masque de sous-réseau (4). D'autres informations sont présentes, comme le flux de paquets entrants et sortants, mais celles mises en exergue composent le socle de base.

Définir une configuration IP

Dans la section précédente, nous avons utilisé la commande `ifconfig` pour afficher des informations essentielles liées à la composition d'interfaces réseau. Toutefois, `ifconfig` propose également de modifier ces interfaces à l'aide d'arguments :

```
root@debiandev:~# ifconfig eth0 10.0.2.7 netmask 255.0.0.0
root@debiandev:~#
```

Le premier argument est le nom de l'interface (ici, `eth0`) à modifier ; puis doit suivre la nouvelle adresse affectée ainsi qu'éventuellement d'autres éléments comme le masque de sous-réseau ou l'adresse de *broadcast*. Une fois l'opération terminée, le changement de configuration peut être vérifié via la commande `ifconfig` suivie du seul nom de l'interface :

```
root@debiandev:~# ifconfig eth0
>>eth0      Link encap:Ethernet HWaddr 08:00:27:1a:73:c7
>>          inet addr:10.0.2.7 Bcast:10.255.255.255 Mask:255.0.0.0
>>          inet6 addr: fe80::a00:27ff:fe1a:73c7/64 Scope:Link
>>          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
>>          RX packets:612 errors:0 dropped:0 overruns:0 frame:0
>>          TX packets:499 errors:0 dropped:0 overruns:0 carrier:0
>>          collisions:0 txqueuelen:1000
>>          RX bytes:796742 (778.0 KiB) TX bytes:55181 (53.8 KiB)
```

La commande `ifconfig` est donc extrêmement efficace, car simple d'utilisation. Elle dispose d'un nombre conséquent de paramètres ; tous les aborder ne rentre pas dans les objectifs de cet ouvrage, mais les quelques exemples que nous venons de voir sont suffisants pour conclure qu'il est très facile dans un environnement Unix/Linux d'agir de manière concrète sur des interfaces réseau.

> **À SAVOIR Champ d'action de la commande ifconfig**
>
> La commande `ifconfig` n'agit que de manière temporaire, contrairement à une configuration permanente. Pour rendre une configuration permanente, il faut en fixer les paramètres dans un fichier dont le nom dépend du système d'exploitation. Dans ce cas, il vaut mieux consulter la documentation du système utilisé.

Perl

Korn Shell étant le mieux adapté pour manipuler des interfaces réseau en environnement Unix/Linux, les quatre autres langages seront sollicités pour manipuler des interfaces réseau en environnement Microsoft.

Lister les adresses IP d'une machine

Collecter une liste d'adresses IP sur une machine locale ou distante est une opération réalisable de plusieurs façons en Perl. Nous utiliserons un module nommé `Win32::IPConfig` (1), très simple à manipuler :

```
use Win32::IPConfig (1);

   $host = shift || "";
   if ($config = Win32::IPConfig->new($host) (2)) {
       print "Hostname ->", $config->get_hostname (3), "\n";

       foreach my $adapter ($config->get_adapters) {
          print "\nInterface '", $adapter->get_name (4), "':\n";

          foreach my $ip ($adapter->get_ipaddresses (5)) {
            print "IP -> $ip\n";
          }
      }
   }
```

Une première étape consiste à créer un objet `Win32::IPConfig` (2), grâce auquel plusieurs informations peuvent être collectées, comme le nom de la machine (3) où la requête est réalisée, le nom de l'interface réseau (4) et l'adresse IP correspondante (5).

```
…
>>Hostname ->MLSTKM

>>Interface 'Ethernet':
>>IP -> 192.168.1.82

>>Interface 'Wi-Fi':
>>IP -> 192.168.1.71
…
```

À la vue des informations affichées, nous pouvons remarquer qu'une adresse IP est liée à chaque interface, ce qui démontre qu'en très peu de lignes de code, des informations critiques sont accessibles aisément.

Le module `Win32::IPConfig` permet, entre autres, de lister les adresses IP d'une machine locale ou distante, ce qui en fait un module très prisé des administrateurs système/réseau.

Collecter des informations de configuration réseau

Il est fréquent de rechercher, en plus des adresses IP, d'autres informations qui reflètent par exemple l'état de la configuration d'une carte réseau.

La classe WMI `Win32_NetworkAdapterConfiguration` (1) est extrêmement intéressante pour obtenir ce genre d'informations :

```
use Win32::OLE('in');
use constant wbemFlagReturnImmediately => 0x10;
use constant wbemFlagForwardOnly => 0x20;

$computer = ".";
$objWMIService = Win32::OLE->GetObject
    ("winmgmts:\\\\$computer\\root\\CIMV2") or die "WMI connection failed.\n";
$colItems = $objWMIService->ExecQuery
    ("SELECT * FROM Win32_NetworkAdapterConfiguration (1) WHERE IPEnabled =
\'TRUE\'","WQL",wbemFlagReturnImmediately | wbemFlagForwardOnly);

foreach my $objItem (in $colItems)
{
      print "Caption: $objItem->{Caption}\n";
      print "Database Path: $objItem->{DatabasePath}\n";
      print "Default IP Gateway: " . join(",", (in $objItem->{DefaultIPGateway})) . "\n";
      print "Default TOS: $objItem->{DefaultTOS}\n";
      print "Default TTL: $objItem->{DefaultTTL}\n";
      print "Description: $objItem->{Description}\n";
      print "DHCP Enabled: $objItem->{DHCPEnabled}\n";
      print "DHCP Lease Expires: $objItem->{DHCPLeaseExpires}\n";
      print "DHCP Lease Obtained: $objItem->{DHCPLeaseObtained}\n";
      print "DHCP Server: $objItem->{DHCPServer}\n";
      print "DNS Domain: $objItem->{DNSDomain}\n";
```

```
        print "DNS Domain Suffix Search Order: " . join(",", (in $objItem->
                        {DNSDomainSuffixSearchOrder})) . "\n";
        print "DNS Enabled For WINS Resolution: $objItem->
                        {DNSEnabledForWINSResolution}\n";
        print "DNS Host Name: $objItem->{DNSHostName}\n";
        print "DNS Server Search Order: " . join(",", (in $objItem->
                        {DNSServerSearchOrder})) . "\n";
        print "Domain DNS Registration Enabled: $objItem->
                        {DomainDNSRegistrationEnabled}\n";
        print "Forward Buffer Memory: $objItem->{ForwardBufferMemory}\n";
        print "Full DNS Registration Enabled: $objItem->
                        {FullDNSRegistrationEnabled}\n";
        print "Gateway Cost Metric: " . join(",", (in $objItem->
                        {GatewayCostMetric})) . "\n";
        print "IGMP Level: $objItem->{IGMPLevel}\n";
        print "Index: $objItem->{Index}\n";
        print "IP Address: " . join(",", (in $objItem->{IPAddress})) . "\n";
        print "IP ConnectionMetric: $objItem->{IPConnectionMetric}\n";
        print "IP Enabled: $objItem->{IPEnabled}\n";
        print "IP Filter Security Enabled: $objItem->{IPFilterSecurityEnabled}\n";
        print "IP Port Security Enabled: $objItem->{IPPortSecurityEnabled}\n";
        print "IPSec Permit IP Protocols: " . join(",", (in $objItem->
                        {IPSecPermitIPProtocols})) . "\n";
        print "IPSec Permit TCP Ports: " . join(",", (in $objItem->
                        {IPSecPermitTCPPorts})) . "\n";
        print "IPSec Permit UDP Ports: " . join(",", (in $objItem->
                        {IPSecPermitUDPPorts})) . "\n";
        print "IP Subnet: " . join(",", (in $objItem->{IPSubnet})) . "\n";
        print "IP Use Zero Broadcast: $objItem->{IPUseZeroBroadcast}\n";
        print "Keep Alive Interval: $objItem->{KeepAliveInterval}\n";
        print "Keep Alive Time: $objItem->{KeepAliveTime}\n";
        print "MAC Address: $objItem->{MACAddress}\n";
        print "MTU: $objItem->{MTU}\n";
        print "Num Forward Packets: $objItem->{NumForwardPackets}\n";
        print "PMTUBH Detect Enabled: $objItem->{PMTUBHDetectEnabled}\n";
        print "PMTU Discovery Enabled: $objItem->{PMTUDiscoveryEnabled}\n";
        print "Service Name: $objItem->{ServiceName}\n";
        print "Setting ID: $objItem->{SettingID}\n";
        print "Tcpip Netbios Options: $objItem->{TcpipNetbiosOptions}\n";
        print "Tcp Max Connect Retransmissions: $objItem->
                        {TcpMaxConnectRetransmissions}\n";
        print "Tcp Max Data Retransmissions: $objItem->{TcpMaxDataRetransmissions}\n";
        print "Tcp Num Connections: $objItem->{TcpNumConnections}\n";
        print "Tcp Use RFC1122 Urgent Pointer: $objItem->
                        {TcpUseRFC1122UrgentPointer}\n";
        print "Tcp Window Size: $objItem->{TcpWindowSize}\n";
        print "WINS Enable LMHosts Lookup: $objItem->{WINSEnableLMHostsLookup}\n";
        print "WINS Host Lookup File: $objItem->{WINSHostLookupFile}\n";
        print "WINS Primary Server: $objItem->{WINSPrimaryServer}\n";
        print "WINS Scope ID: $objItem->{WINSScopeID}\n";
        print "WINS Secondary Server: $objItem->{WINSSecondaryServer}\n";
        print "\n";
}
```

Cette classe dispose de nombreuses informations plus ou moins importantes qu'il est utile de connaître lorsque, par exemple, un rapport sur l'état des cartes réseau de l'ensemble d'un domaine est demandé :

```
>>Caption: [00000000] Contrôleur Realtek PCIe GBE Family
>>Database Path: %SystemRoot%\System32\drivers\etc
>>Default IP Gateway: 192.168.1.254
>>Default TOS:
>>Default TTL:
>>Description: Contrôleur Realtek PCIe GBE Family
>>DHCP Enabled: 1
>>DHCP Lease Expires: 20140514204425.000000+120
>>DHCP Lease Obtained: 20140513204425.000000+120
>>DHCP Server: 192.168.1.254
>>DNS Domain: lan
>>DNS Domain Suffix Search Order: lan
>>DNS Enabled For WINS Resolution: 0
>>DNS Host Name: A720
>>DNS Server Search Order: 192.168.1.254
>>Domain DNS Registration Enabled: 0
>>Forward Buffer Memory:
>>Full DNS Registration Enabled: 1
>>Gateway Cost Metric: 0
>>IGMP Level:
>>Index: 0
>>IP Address: 192.168.1.86,fe80::90c:3007:d14:8660
>>IP ConnectionMetric: 20
>>IP Enabled: 1
>>IP Filter Security Enabled: 0
>>IP Port Security Enabled:
>>IPSec Permit IP Protocols:
>>IPSec Permit TCP Ports:
>>IPSec Permit UDP Ports:
>>IP Subnet: 255.255.255.0,64
>>IP Use Zero Broadcast:
>>Keep Alive Interval:
>>Keep Alive Time:
>>MAC Address: 08:5E:01:55:E2:8F
>>MTU:
>>Num Forward Packets:
>>PMTUBH Detect Enabled:
>>PMTU Discovery Enabled:
>>Service Name: RTL8168
>>Setting ID: {D9413EF1-168C-4F2A-A0E3-F5BC5919C4E1}
>>Tcpip Netbios Options: 0
>>Tcp Max Connect Retransmissions:
>>Tcp Max Data Retransmissions:
>>Tcp Num Connections:
>>Tcp Use RFC1122 Urgent Pointer:
>>Tcp Window Size:
>>WINS Enable LMHosts Lookup: 1
>>WINS Host Lookup File:
```

```
>>WINS Primary Server:
>>WINS Scope ID:
>>WINS Secondary Server:
```

Le nombre d'informations affichées est conséquent mais elles doivent être croisées avec d'autres éléments tout aussi critiques afin de constituer une synthèse la plus claire possible.

Python

Une configuration réseau peut avoir plusieurs statuts : elle peut être dynamique, statique ou posséder d'autres dispositions en lien avec certains services. Cette partie mettra en évidence comment définir une configuration statique et désactiver le protocole IPSec.

Définir une configuration statique

Il est parfois nécessaire de passer d'une configuration dynamique à une configuration statique. Certaines machines ont en effet besoin de ne pas modifier leur configuration réseau, car elles doivent être joignables tout le temps.

Automatiser l'attribution de configurations statiques sur certaines machines est donc une opération très récurrente de nos jours :

```
import wmi
obj = wmi.WMI()
NetCard = obj.Win32_NetworkAdapterConfiguration(Caption='Intel(R) PRO/1000 MT' (1))

ip = u'10.0.2.40' (2)
subnetmask = u'255.0.0.0' (3)
gateway = u'10.0.2.100' (4)

for interface in NetCard:
    error_static = interface.EnableStatic(IPAddress=[ip],
                                          SubnetMask=[subnetmask]) (5)
    if error_static[0] == 0:
        print "The IP address has been changed."
    else:
        print "The IP address could not be changed."
    error_gat = interface.SetGateways(DefaultIPGateway=[gateway]) (6)
    if error_gat[0] == 0:
        print "The gateway address has been changed."
    else:
        print "The gateway address could not be changed."
```

Ce code opère un changement de configuration IP vers le mode statique. Une interface réseau est sélectionnée (1), puis une nouvelle adresse IP (2), un masque de sous-réseau (3) et une passerelle par défaut (4) sont définis. Ces informations constituent la nouvelle configuration de l'interface concernée.

Pour affecter cette nouvelle configuration à l'interface réseau, les méthodes `EnableStatic()` (5) et `SetGateway()` (6) de la classe `Win32_NetworkAdapterConfiguration` sont instanciées ; dans les deux cas, si le changement effectué est réalisé sans problèmes, nous devrions le voir en sortie :

```
>>The IP address has been changed.
>>The gateway address has been changed.
```

Dans le cas contraire, un avertissement nous est également indiqué en sortie. Cette étape franchie, une simple vérification (optionnelle) à l'aide de la commande `ipconfig` nous permettra de constater ce changement :

```
PS> ipconfig

>>Configuration IP de Windows


>>Carte Ethernet … :

>>   Suffixe DNS propre à la connexion. . . :
>>   Adresse IPv6 de liaison locale. . . . .: fe80::…
>>   Adresse IPv4. . . . . . . . . . . . . .: 10.0.2.40
>>   Masque de sous-réseau. . . . . . . . . : 255.0.0.0
>>   Passerelle par défaut. . . . . . . . . : 10.0.2.100
```

Désactiver IPSec

Le protocole IPSec *(Internet Protocol Security)* est un ensemble de standards utilisés pour sécuriser le transport de données sur un réseau IP. Pour des raisons diverses, l'activation de ce protocole est parfois essentielle, parfois totalement inutile.

Le script suivant sert à désactiver précisément le protocole IPSec :

```
import wmi
obj = wmi.WMI()
NetCard = obj.Win32_NetworkAdapterConfiguration(Caption='Intel(R) PRO/1000 MT')

for interface in NetCard:
    error_ipsec = interface.DisableIPSec() (1)
    if error_ipsec[0] == 0:
        print "The IPSec protocol has been disabled."
    elif error_ipsec[0] == 1:
        print "The IPSec protocol has been disabled. Reboot required."
    else:
        print "The IPSec protocol could not be disabled."
```

La méthode `DisableIPSec()` (1) autorise la désactivation du protocole IPSec au niveau de l'interface réseau visée :

```
>>The IPSec protocol has been disabled. Reboot required.
```

Une étape importante consiste à vérifier la valeur retournée par la méthode exécutée, car de multiples cas de figure peuvent apparaître. Pour plus d'informations, consultez la documentation MSDN.

VBS

Visual Basic Scripting est très efficace en matière de gestion d'une interface réseau. C'est ce que nous allons voir ici à travers le domaine de la configuration DNS *(Domain Name System)*.

Définir un nom de domaine associé à des interfaces réseau

Il est fréquent de voir de grands réseaux subdivisés en domaines. Ceci implique que chaque nœud d'un domaine a comme caractéristique un lien d'appartenance à un domaine particulier. Cette remarque est logique, mais plus l'infrastructure est complexe, plus le nombre de liaisons entre un nœud donné et des domaines liés les uns aux autres est élevé.

La démarche qui consiste à lier une interface réseau spécifique à un ou plusieurs domaine(s) est donc une opération qui a tout intérêt à être automatisée, car il peut s'agir de centaines, voire de milliers de machines à la fois. Le code suivant est une illustration en VBS de cette démarche :

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:" _
    & "{impersonationLevel=impersonate}!\\" & strComputer & "\root\cimv2")

Set colNetCards = objWMIService.ExecQuery _
    ("Select * From Win32_NetworkAdapterConfiguration (1) Where IPEnabled = True")

For Each objNetCard in colNetCards
    objNetCard.SetDNSDomain("aurora.ml") (2)
Next
```

Toujours via la classe `Win32_NetworkAdapterConfiguration` (1), la méthode `SetDNSDomain()` (2) définit le nom de domaine associé à une interface réseau. En l'occurrence, le code précédent autorise une association entre le domaine `[aurora.ml]` et l'ensemble des interfaces réseau d'une machine locale.

Surtout, lors de ce genre d'opérations, et il en sera de même pour la section suivante, les privilèges requis doivent être suffisants, au risque de rencontrer des erreurs.

Définir l'ordre dans lequel les serveurs DNS sont interrogés

Pour une configuration DNS la plus saine possible, il est nécessaire de spécifier l'ordre dans lequel les serveurs sont interrogés. En effet, une architecture structurée de manière solide comporte des serveurs DNS primaires, mais aussi d'autres serveurs DNS secondaires capables de répondre à certaines requêtes si le contexte le nécessite.

Ci-après, la méthode `SetDNSServerSearchOrder()` (1) ordonne la façon dont une machine locale effectue ses requêtes DNS :

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:" _
    & "{impersonationLevel=impersonate}!\\" & strComputer & "\root\cimv2")

Set colNetCards = objWMIService.ExecQuery _
    ("Select * From Win32_NetworkAdapterConfiguration Where IPEnabled = True")

For Each objNetCard in colNetCards
    arrDNSServers = Array("10.0.2.98", "10.0.2.99") (2)
    objNetCard.SetDNSServerSearchOrder(arrDNSServers) (1)
Next
```

L'argument (2) passé à la méthode `SetDNSServerSearchOrder()` est une liste d'adresses de serveurs DNS vis-à-vis desquels chaque requête, et ce dans l'ordre spécifié, va se déployer. La question ici est de bien réaliser une configuration homogène à l'échelle de l'ensemble de l'infrastructure ; cette dimension est extrêmement importante parce qu'une configuration réseau bien établie préside à un évitement de cas isolés provoquant des problèmes allant jusqu'à faire baisser l'efficacité et la productivité d'une infrastructure dans sa totalité.

Windows PowerShell

Avec l'arrivée de PowerShell version 3 (et ultérieures), gérer la configuration d'une interface réseau, côté client ou côté serveur, est plus aisé. En effet, pour beaucoup de tâches d'administration, il n'y a plus besoin de pratiquer la programmation système. À la place, un certain nombre de cmdlets existent aujourd'hui et autorisent une configuration réseau plus accessible.

Obtenir des informations DNS plus détaillées côté client

Windows PowerShell propose, au fur et à mesure de l'avancement des versions, de nombreux modules ayant chacun une orientation bien précise. Pour la gestion d'une interface réseau, il existe un module nommé `DnsClient` ; comme son nom l'indique, il permet une configuration DNS côté client.

Parmi les commandes offertes, nous en évoquerons deux : `Get-DnsClient` et `Set-DnsClient`. Tout d'abord, la cmdlet `Get-DnsClient` est très utile pour obtenir d'une ou de plusieurs interface(s) réseau des informations DNS essentielles :

```
PS> Get-DnsClient | Format-List

>>InterfaceAlias                     : Eth_Net
>>InterfaceIndex                     : 12
>>ConnectionSpecificSuffix           : aurora.ml
>>ConnectionSpecificSuffixSearchList : {}
>>RegisterThisConnectionsAddress     : True
>>UseSuffixWhenRegistering           : False

>>InterfaceAlias                     : isatap.aurora.ml
>>InterfaceIndex                     : 15
>>ConnectionSpecificSuffix           : aurora.ml
>>ConnectionSpecificSuffixSearchList : {}
>>RegisterThisConnectionsAddress     : True
>>UseSuffixWhenRegistering           : True

>>InterfaceAlias                     : Loopback Pseudo-Interface 1
>>InterfaceIndex                     : 1
>>ConnectionSpecificSuffix           :
>>ConnectionSpecificSuffixSearchList : {}
>>RegisterThisConnectionsAddress     : True
>>UseSuffixWhenRegistering           : True

>>InterfaceAlias                     : Teredo Tunneling Pseudo-Interface
>>InterfaceIndex                     : 14
>>ConnectionSpecificSuffix           :
>>ConnectionSpecificSuffixSearchList : {}
>>RegisterThisConnectionsAddress     : True
>>UseSuffixWhenRegistering           : True
```

Utilisée sans paramètres, `Get-DnsClient` retourne des informations DNS concernant toutes les interfaces réseau présentes. Toutefois, le paramètre `-InterfaceAlias` sert à cibler une requête :

```
PS> Get-DnsClient -InterfaceAlias Eth_Net | Format-List

>>InterfaceAlias                     : Eth_Net
>>InterfaceIndex                     : 12
>>ConnectionSpecificSuffix           : aurora.ml
>>ConnectionSpecificSuffixSearchList : {}
>>RegisterThisConnectionsAddress     : True
>>UseSuffixWhenRegistering           : False
```

Modifier des informations DNS côté client

La modification d'informations DNS côté client s'effectue entre autres à l'aide de la commande `Set-DnsClient`. Par exemple, essayons de modifier le suffixe DNS d'une interface réseau appartenant à une machine locale :

```
PS> Set-DnsClient -InterfaceAlias Eth_Net -ConnectionSpecificSuffix
'devel.aurora.ml' -PassThru | fl

>>InterfaceAlias                    : Eth_Net
>>InterfaceIndex                    : 12
>>ConnectionSpecificSuffix          : devel.aurora.ml (1)
>>ConnectionSpecificSuffixSearchList : {}
>>RegisterThisConnectionsAddress     : True
>>UseSuffixWhenRegistering           : False
```

Le changement de cette information a été possible grâce au paramètre `-ConnectionSpecificSuffix`. D'ailleurs, l'attribut `ConnectionSpecificSuffix` (1) montre que le changement a bien eu lieu.

Une autre information susceptible d'être modifiée est l'inscription du nom DNS d'une interface réseau ainsi que l'adresse IP qui correspond :

```
PS> Set-DnsClient -InterfaceAlias Eth_Net -RegisterThisConnectionsAddress 0 -PassThru
| fl

>>InterfaceAlias                    : Eth_Net
>>InterfaceIndex                    : 12
>>ConnectionSpecificSuffix          : devel.aurora.ml
>>ConnectionSpecificSuffixSearchList : {}
>>RegisterThisConnectionsAddress     : False
>>UseSuffixWhenRegistering           : False
```

Cette ligne de commande sollicite le paramètre `-RegisterThisConnectionsAddress` afin de désactiver, à l'aide de la valeur `0`, l'inscription dynamique du nom DNS de l'interface concernée (ici, `Eth_Net`).

On observe enfin que certaines cmdlets PowerShell effectuent aujourd'hui un travail qui auparavant était réalisé par des commandes natives Windows. Ce phénomène va, avec le temps, se répandre à toutes les couches d'administration d'une plate-forme Windows.

19

Manipuler un fichier XML

XML (Extensible Markup Language) *est devenu aujourd'hui universel. En effet, le principe sur lequel ce langage repose est à l'heure actuelle parfaitement intégré à de très nombreux écosystèmes pouvant présenter tout à la fois des caractéristiques similaires ou totalement différentes.*

Les contextes d'utilisation de XML sont évidemment variés. Il peut s'agir, entre autres, de fichiers de configuration, constitution de bases de données, procédés présidant à une certaine interopérabilité ou encore développement de corpus de métadonnées.

Les domaines d'application de XML sont nombreux car il s'intègre bien aux autres langages de programmation, notamment de script, sollicitant des voies dites natives, mais aussi d'autres plus modulaires. Dans ce chapitre, nous verrons comment manipuler un fichier XML à partir des cinq langages invoqués depuis le début de cet ouvrage.

Le fichier books.xml

Pour mettre en valeur les exemples des sections suivantes, nous nous appuierons sur le fichier XML suivant (`books.xml`) :

```
<?xml version="1.0"?>
<catalog>
   <book id="bk101">
      <author>Gambardella, Matthew</author>
      <title>XML Developer's Guide</title>
      <genre>Computer</genre>
      <price>44.95</price>
```

```
    <publish_date>2000-10-01</publish_date>
    <description>An in-depth look at creating applications
    with XML.</description>
</book>
<book id="bk102">
    <author>Ralls, Kim</author>
    <title>Midnight Rain</title>
    <genre>Fantasy</genre>
    <price>5.95</price>
    <publish_date>2000-12-16</publish_date>
    <description>A former architect battles corporate zombies,
    an evil sorceress, and her own childhood to become queen
    of the world.</description>
</book>
<book id="bk103">
    <author>Corets, Eva</author>
    <title>Maeve Ascendant</title>
    <genre>Fantasy</genre>
    <price>5.95</price>
    <publish_date>2000-11-17</publish_date>
    <description>After the collapse of a nanotechnology
    society in England, the young survivors lay the
    foundation for a new society.</description>
</book>
<book id="bk104">
    <author>Corets, Eva</author>
    <title>Oberon's Legacy</title>
    <genre>Fantasy</genre>
    <price>5.95</price>
    <publish_date>2001-03-10</publish_date>
    <description>In post-apocalypse England, the mysterious
    agent known only as Oberon helps to create a new life
    for the inhabitants of London. Sequel to Maeve
    Ascendant.</description>
</book>
<book id="bk105">
    <author>Corets, Eva</author>
    <title>The Sundered Grail</title>
    <genre>Fantasy</genre>
    <price>5.95</price>
    <publish_date>2001-09-10</publish_date>
    <description>The two daughters of Maeve, half-sisters,
    battle one another for control of England. Sequel to
    Oberon's Legacy.</description>
</book>
<book id="bk106">
    <author>Randall, Cynthia</author>
    <title>Lover Birds</title>
    <genre>Romance</genre>
    <price>4.95</price>
    <publish_date>2000-09-02</publish_date>
    <description>When Carla meets Paul at an ornithology
```

```
   conference, tempers fly as feathers get ruffled.</description>
</book>
<book id="bk107">
   <author>Thurman, Paula</author>
   <title>Splish Splash</title>
   <genre>Romance</genre>
   <price>4.95</price>
   <publish_date>2000-11-02</publish_date>
   <description>A deep sea diver finds true love twenty
   thousand leagues beneath the sea.</description>
</book>
<book id="bk108">
   <author>Knorr, Stefan</author>
   <title>Creepy Crawlies</title>
   <genre>Horror</genre>
   <price>4.95</price>
   <publish_date>2000-12-06</publish_date>
   <description>An anthology of horror stories about roaches,
   centipedes, scorpions and other insects.</description>
</book>
<book id="bk109">
   <author>Kress, Peter</author>
   <title>Paradox Lost</title>
   <genre>Science Fiction</genre>
   <price>6.95</price>
   <publish_date>2000-11-02</publish_date>
   <description>After an inadvertant trip through a Heisenberg
   Uncertainty Device, James Salway discovers the problems
   of being quantum.</description>
</book>
<book id="bk110">
   <author>O'Brien, Tim</author>
   <title>Microsoft .NET: The Programming Bible</title>
   <genre>Computer</genre>
   <price>36.95</price>
   <publish_date>2000-12-09</publish_date>
   <description>Microsoft's .NET initiative is explored in
   detail in this deep programmer's reference.</description>
</book>
<book id="bk111">
   <author>O'Brien, Tim</author>
   <title>MSXML3: A Comprehensive Guide</title>
   <genre>Computer</genre>
   <price>36.95</price>
   <publish_date>2000-12-01</publish_date>
   <description>The Microsoft MSXML3 parser is covered in
   detail, with attention to XML DOM interfaces, XSLT processing,
   SAX and more.</description>
</book>
<book id="bk112">
   <author>Galos, Mike</author>
   <title>Visual Studio 7: A Comprehensive Guide</title>
```

```
      <genre>Computer</genre>
      <price>49.95</price>
      <publish_date>2001-04-16</publish_date>
      <description>Microsoft Visual Studio 7 is explored in depth,
      looking at how Visual Basic, Visual C++, C#, and ASP+ are
      integrated into a comprehensive development
      environment.</description>
   </book>
</catalog>
```

Ce fichier constitue un catalogue comprenant, pour chaque ouvrage représenté, un certain nombre d'attributs dont voici la liste :

- un identifiant (`book id`) ;
- un nom d'auteur (`author`) ;
- un titre (`title`) ;
- un genre (`genre`) ;
- un prix (`price`) ;
- une date de publication (`publish_date`) ;
- un synopsis (`description`).

Vous l'aurez compris, ce fichier peut servir différentes démarches, dont la construction d'une petite base de données.

KSH

Dans le monde de la programmation Shell, il existe plusieurs moyens de lire et de manipuler un fichier XML. Nous nous intéresserons uniquement à un outil en particulier : `xmllint`. Cette commande, parmi tant d'autres, est essentiellement un outil d'analyse en lien avec la structure d'un fichier XML.

L'outil xmllint

Il n'y rien de tel qu'un outil dédié pour effectuer des tâches bien précises. Dans le domaine de la manipulation de fichiers XML, la commande `xmllint` est un des outils principaux. Elle offre en effet une possibilité d'interaction à partir d'une navigation dédiée :

```
root@debiandev:/home/scripting/# xmllint --shell books.xml
/ > 1
```

Le paramètre `--shell` sert à passer en mode shell interactif 1 afin d'y réaliser toutes les opérations dont nous avons besoin. Pour afficher une aide montrant les commandes disponibles, il faut taper la commande `help`.

```
/ > help
      base         display XML base of the node
      setbase URI change the XML base of the node
      bye          leave shell
      cat [node]   display node or current node
      cd [path]    change directory to path or to root
      dir [path]   dumps informations about the node (namespace, attributes, content)
      du [path]    show the structure of the subtree under path or the current node
      exit         leave shell
      help         display this help
      free         display memory usage
      load [name] load a new document with name
      ls [path]    list contents of path or the current directory
      set xml_fragment replace the current node content with the fragment parsed in
context
      xpath expr   evaluate the XPath expression in that context and print the result
      setns nsreg register a namespace to a prefix in the XPath evaluation context
                   format for nsreg is: prefix=[nsuri] (i.e. prefix= unsets a prefix)
      setrootns    register all namespace found on the root element
                   the default namespace if any uses 'defaultns' prefix
      pwd          display current working directory
      whereis      display absolute path of [path] or current working directory
      quit         leave shell
      save [name] save this document to name or the original name
      write [name] write the current node to the filename
      validate     check the document for errors
      relaxng rng validate the document agaisnt the Relax-NG schemas
      grep string search for a string in the subtree
/ >
```

Comme nous pouvons le constater, les commandes disponibles sont nombreuses. Par exemple, `cat` servira à afficher tous les titres du catalogue :

```
/ > cat //title
-------
<title>XML Developer's Guide</title>
-------
<title>Midnight Rain</title>
-------
<title>Maeve Ascendant</title>
-------
<title>Oberon's Legacy</title>
-------
<title>The Sundered Grail</title>
-------
<title>Lover Birds</title>
-------
<title>Splish Splash</title>
-------
<title>Creepy Crawlies</title>
```

```
-------
<title>Paradox Lost</title>
-------
<title>Microsoft .NET: The Programming Bible</title>
-------
<title>MSXML3: A Comprehensive Guide</title>
-------
<title>Visual Studio 7: A Comprehensive Guide</title>
/ >
```

Le paramètre `//title` représente le nœud `<title>`, correspondant lui-même à un titre d'ouvrage. Lister l'ensemble des noms d'auteurs utilise la même méthode, en modifiant évidemment le nom du nœud :

```
/ > cat //author
-------
<author>Gambardella, Matthew</author>
-------
<author>Ralls, Kim</author>
-------
<author>Corets, Eva</author>
-------
<author>Corets, Eva</author>
-------
<author>Corets, Eva</author>
-------
<author>Randall, Cynthia</author>
-------
<author>Thurman, Paula</author>
-------
<author>Knorr, Stefan</author>
-------
<author>Kress, Peter</author>
-------
<author>O'Brien, Tim</author>
-------
<author>O'Brien, Tim</author>
-------
<author>Galos, Mike</author>
/ >
```

La commande `cd` autorise la navigation d'un nœud à un autre. Par exemple, la ligne suivante opère un déplacement de la racine (`/`) vers le nœud `<book id="bk103">` :

```
/ > cd ///catalog/book[@id = "bk103"]
book > ls
tan        7
---        1 author
tan        7
---        1 title
```

```
tan         7
---         1 genre
tan         7
---         1 price
tan         7
---         1 publish_date
tan         7
---         1 description
ta-         4
book > dir
ELEMENT book
  ATTRIBUTE id
    TEXT
      content=bk103
book >
```

Le retour en arrière est aussi possible :

```
book > cd /
/ >
```

L'outil `grep` est reconnu, ce qui autorise la recherche d'occurrences précises :

```
/ > grep Visual
/catalog/book[12]/title : t--   38 Visual Studio 7: A Comprehensive Guide
/catalog/book[12]/description : t-- 182 Microsoft Visual Studio 7 is explored in...
/ >
```

Il faudrait au moins un chapitre entier pour explorer l'outil `xmllint` ; les nombreuses combinaisons possibles associées à la possibilité de travailler avec plusieurs fichiers XML le rend précieux.

Perl

Perl est un langage de script extrêmement puissant dans le traitement de fichiers et XML ne déroge pas à cette règle. Le fait est que l'extraction d'informations à partir de fichiers XML est réalisable de plusieurs façons. Nous allons présenter un module très populaire, `XML::Simple`, qui est dédié à la manipulation de telles structures.

Le module XML::Simple

`XML::Simple` est très utilisé par les programmeurs Perl car il est, à la base, simple d'emploi. Le code suivant invoque ce module (1) afin de créer un fichier XML contenant des informations (identifiant et titre) liées à des ouvrages.

```
use XML::Simple; (1)

(3) $structure = {
    book => (2) [
              {
                id      => 1,
                title   => [ "KSH" ]
              },
              {
                id      => 2,
                title   => [ "Perl" ]
              },
              {
                id      => 3,
                title   => [ "Python" ]
              },
               {
                id      => 4,
                title   => [ "VBS" ]
              },
               {
                id      => 5,
                title   => [ "PowerShell" ]
              }
            ]
  };
$xml = (4) XMLout($structure, OutputFile => '.\out_file.xml');
```

La structure du code consiste en un tableau anonyme (2) contenant une liste de dictionnaires anonymes, qui constituent autant de nœuds (ici, `<book></book>`) devant apparaître dans le fichier XML. Au final, nous devrions voir cinq nœuds `<book></book>` au sein du fichier `out_file.xml` :

```
<opt>
  <book id="1">
    <title>KSH</title>
  </book>
  <book id="2">
    <title>Perl</title>
  </book>
  <book id="3">
    <title>Python</title>
  </book>
  <book id="4">
    <title>VBS</title>
  </book>
  <book id="5">
    <title>PowerShell</title>
  </book>
</opt>
```

La variable `$structure` ❸ contenant la structure XML est passée en argument à la méthode `XMLout()` ❹, ce qui permet ici la création du fichier `out_file.xml`.

La méthode `XMLin()` ❺ sert, quant à elle, à lire une structure XML. Reprenons, par exemple, le fichier `out_file.xml` et essayons d'en extraire quelques informations :

```
use XML::Simple;

$xml = ❺ XMLin('.\out_file.xml');

foreach ❻ $id ( keys %{$xml->{book}} ) {
    print $xml->{book}{$id}{title}, "\n";
}
```

Une instruction `foreach` ❻ itère à travers l'ensemble des éléments `<book></book>` pour en extraire, à chaque fois, une information issue du nœud `<title></title>`. Le résultat affiché représente une liste de titres d'ouvrages :

```
>>VBS
>>KSH
>>Python
>>Perl
>>PowerShell
```

Le module `XML::Simple` est un très bon outil dans le cadre de la manipulation de fichiers XML. Cependant, en fonction de la complexité du fichier XML, son niveau d'utilisation peut devenir très ardu.

Python

Comme en Perl, différents modules, similaires dans leur fonctionnement de base, servent au traitement de structures XML. Nous nous attarderons là aussi sur un module en particulier, `xml.etree.ElementTree`.

Le module xml.etree.ElementTree

Le module `xml.etree.ElementTree` sert à manipuler des structures de données XML. De telles structures sont composées d'éléments fortement hiérarchisés : chaque élément contient lui-même d'autres éléments, et ainsi de suite.

Cet aspect hiérarchique des données aide le programmeur à mieux gérer des informations qui sont parfois massives. La fonction du module `xml.etree.ElementTree` est justement de créer cette facilité – plus les données à gérer sont importantes, plus l'aide apportée par ce module est précieuse.

Le fichier `books.xml` contient une structure non massive facile à manipuler. Articulons-le avec le module `xml.etree.ElementTree` (1) :

```
import xml.etree.ElementTree as ET (1)
tree = ET.parse('books.xml') (2)
root = tree.getroot() (3)

for (4) title in root.iter('title'): (5)
    print(title.text)
```

D'abord, le fichier `books.xml` doit être analysé afin de construire un objet `ElementTree` (2). Puis nous devons nous placer à la racine de l'arborescence XML (3) avant de démarrer une quelconque opération, de manière à ce qu'une itération (4) à partir de la base puisse être réalisée.

La méthode `iter()` (5) parcourt les éléments en commençant par la racine de l'arborescence XML, le but étant de lister tous les titres provenant de ce catalogue :

```
>>XML Developer's Guide
>>Midnight Rain
>>Maeve Ascendant
>>Oberon's Legacy
>>The Sundered Grail
>>Lover Birds
>>Splish Splash
>>Creepy Crawlies
>>Paradox Lost
>>Microsoft .NET: The Programming Bible
>>MSXML3: A Comprehensive Guide
>>Visual Studio 7: A Comprehensive Guide
```

Le code suivant liste tous les titres du catalogue, avec leur prix associé :

```
import xml.etree.ElementTree as ET
tree = ET.parse('books.xml')
root = tree.getroot()

for book (1) in root.findall('book'):
    title = book.find('title').text (2)
    price = book.find('price').text (3)
    print(title, '->', price)
```

Pour chaque élément `<book></book>` (1), nous affichons les sous-éléments `<title></title>` (2) et `<price></price>` (3). Voici le résultat :

```
>>XML Developer's Guide -> 44.95
>>Midnight Rain -> 5.95
>>Maeve Ascendant -> 5.95
>>Oberon's Legacy -> 5.95
```

```
>>The Sundered Grail -> 5.95
>>Lover Birds -> 4.95
>>Splish Splash -> 4.95
>>Creepy Crawlies -> 4.95
>>Paradox Lost -> 6.95
>>Microsoft .NET: The Programming Bible -> 36.95
>>MSXML3: A Comprehensive Guide -> 36.95
>>Visual Studio 7: A Comprehensive Guide -> 49.95
```

Les exemples d'utilisation du module `xml.etree.ElementTree` montrent à quel point la manipulation de fichiers XML en Python peut être aisée. Comme évoqué plus haut, il en existe évidemment d'autres, mais ce module confère incontestablement plus d'efficacité dans ce domaine.

VBS

En VBS, il existe essentiellement un objet majeur pour la manipulation de fichiers XML, nommé `Microsoft.XMLDOM`.

L'objet Microsoft.XMLDOM

La hiérarchie des éléments composant un document XML doit pouvoir être exposée clairement, afin de faciliter la manipulation des composants.

L'exposition du contenu d'un document XML est réalisée par la spécification DOM *(Document Object Model)*. Cette dernière consiste en un ensemble de procédés permettant la manipulation de structures XML. Il y a toutefois plusieurs implémentations exigeant comme étape préalable la lecture d'une multitude de documentations.

L'objet `Microsoft.XMLDOM` va nous servir dans cette section à modifier le prix d'un ouvrage du fichier `books.xml` :

```
Set xmlDoc = _
  CreateObject("Microsoft.XMLDOM") (1)

xmlDoc.Async = "False"
xmlDoc.Load(".\books.xml") (2)

Set colNodes=xmlDoc.selectNodes _
  ("catalog/book[@id = 'bk101']/price") (3)

For Each objNode in colNodes
   objNode.Text = "48.95" (4)
Next

xmlDoc.Save ".\books.xml" (5)
```

L'activation de DOM nécessite la création d'un objet `Microsoft.XMLDOM` (1). Ensuite, le fichier `books.xml` doit être chargé en mémoire (2). Ici, l'opération de modification consistera à changer le prix du livre ayant la référence `bk101`.

Pour ce faire, la méthode `selectnodes()` (3) doit nous faire accéder à la sélection du nœud `<price></price>` ; une attention importante doit être portée à cette étape, au risque de ne pas modifier les bonnes informations. Par exemple, comme l'ouvrage concerné porte la référence `bk101`, l'attribut `id` doit être correctement référencé (`[@id = '101']`).

Le prix de l'ouvrage est ensuite modifié (4) à l'aide de la propriété `Text` de l'objet représentant le nœud sélectionné. Enfin, il faut enregistrer (5) le ou les changement(s). La nouvelle information apparaît dans le fichier `books.xml` :

```
<book id="bk101">
      <author>Gambardella, Matthew</author>
      <title>XML Developer's Guide</title>
      <genre>Computer</genre>
      <price>48.95</price>
      <publish_date>2000-10-01</publish_date>
      <description>An in-depth look at creating applications
      with XML.</description>
</book>
```

Windows PowerShell

Une structure XML est, en PowerShell, représentée sous la forme d'un objet utilisable comme n'importe quel autre. Cette approche facilite effectivement toute manipulation dans la mesure où les informations exposées respectent une hiérarchie fidèle à celle composant la structure d'un document XML.

La classe System.Xml.XmlDocument

Pour manipuler un fichier XML en PowerShell, sa structure doit être exposée en tant qu'objet :

```
PS> [xml]$books = Get-Content -Path .\books.xml
```

Cette ligne de commande n'est rien d'autre qu'un transtypage du contenu du fichier `books.xml` en un objet de type `System.Xml.XmlDocument` (1), comme le montre le résultat de la commande `Get-Member` :

```
PS> $books | Get-Member

>>   TypeName : System.Xml.XmlDocument (1)
```

```
>>Name                          MemberType          Definition
>>----                          ----------          ----------
>>ToString                      CodeMethod          static string XmlNode
>>AppendChild                   Method              System.Xml.XmlNode Ap
>>Clone                         Method              System.Xml.XmlNode Cl
>>CloneNode                     Method              System.Xml.XmlNode Cl
>>CreateAttribute               Method              System.Xml.XmlAttribu
>>CreateCDataSection            Method              System.Xml.XmlCDataSe
>>CreateComment                 Method              System.Xml.XmlComment
>>CreateDocumentFragment        Method              System.Xml.XmlDocumen
>>CreateDocumentType            Method              System.Xml.XmlDocumen
>>CreateElement                 Method              System.Xml.XmlElement
>>CreateEntityReference         Method              System.Xml.XmlEntityR
>>CreateNavigator               Method              System.Xml.XPath.XPat
>>CreateNode                    Method              System.Xml.XmlNode Cr
>>CreateProcessingInstruction Method               System.Xml.XmlProcess
>>CreateSignificantWhitespace Method               System.Xml.XmlSignifi
>>CreateTextNode                Method              System.Xml.XmlText Cr
>>CreateWhitespace              Method              System.Xml.XmlWhitesp
>>CreateXmlDeclaration          Method              System.Xml.XmlDeclara
…
```

Nous pouvons effectuer à présent des opérations sur le fichier `books.xml` à travers les membres exposés par la classe `System.Xml.XmlDocument`. Affichons d'abord l'ensemble des éléments à partir de la racine :

```
PS> $books.catalog.book

>>id           : bk101
>>author       : Gambardella, Matthew
>>title        : XML Developer's Guide
>>genre        : Computer
>>price        : 44.95
>>publish_date : 2000-10-01
>>description  : An in-depth look at creating applications
                 with XML.

>>id           : bk102
>>author       : Ralls, Kim
>>title        : Midnight Rain
>>genre        : Fantasy
>>price        : 5.95
>>publish_date : 2000-12-16
>>description  : A former architect battles corporate zombies,
                 an evil sorceress, and her own childhood to become queen
                 of the world.

>>id           : bk103
>>author       : Corets, Eva
>>title        : Maeve Ascendant
```

```
>>genre        : Fantasy
>>price        : 5.95
>>publish_date : 2000-11-17
>>description  : After the collapse of a nanotechnology
                  society in England, the young survivors lay the
                  foundation for a new society.

>>id           : bk104
>>author       : Corets, Eva
>>title        : Oberon's Legacy
>>genre        : Fantasy
>>price        : 5.95
>>publish_date : 2001-03-10
>>description  : In post-apocalypse England, the mysterious
                  agent known only as Oberon helps to create a new life
                  for the inhabitants of London. Sequel to Maeve
                  Ascendant.

>>id           : bk105
>>author       : Corets, Eva
>>title        : The Sundered Grail
>>genre        : Fantasy
>>price        : 5.95
>>publish_date : 2001-09-10
>>description  : The two daughters of Maeve, half-sisters,
                  battle one another for control of England. Sequel to
                  Oberon's Legacy.
…
```

Lister tous les titres du catalogue est très simple :

```
PS> $books.catalog.book.title
>>XML Developer's Guide
>>Midnight Rain
>>Maeve Ascendant
>>Oberon's Legacy
>>The Sundered Grail
>>Lover Birds
>>Splish Splash
>>Creepy Crawlies
>>Paradox Lost
>>Microsoft .NET: The Programming Bible
>>MSXML3: A Comprehensive Guide
>>Visual Studio 7: A Comprehensive Guide
```

Chaque élément est accessible comme une propriété d'objet et ceci est valable pour toute la hiérarchie d'une structure XML à partir de la racine. La méthode `SelectNodes()` est très utile pour sélectionner un nœud ainsi que les sous-nœuds correspondants.

```
PS> $books.selectNodes("catalog/book[@id='bk102']")


>>id           : bk102
>>author       : Ralls, Kim
>>title        : Midnight Rain
>>genre        : Fantasy
>>price        : 5.95
>>publish_date : 2000-12-16
>>description  : A former architect battles corporate zombies,
                 an evil sorceress, and her own childhood to become queen
                 of the world.
```

Modifions à l'aide de la méthode `SelectSingleNode()` le prix de l'ouvrage « Midnight Rain » :

```
PS> $books.SelectSingleNode("//catalog/book[@id='bk102']/price").
InnerText = 7.95
```

Le moindre changement nécessite de sauvegarder le fichier :

```
PS> $books.Save('C:\xml_docs\books.xml')
```

Nous pouvons vérifier que la modification a bien eu lieu :

```
…
>>id           : bk102
>>author       : Ralls, Kim
>>title        : Midnight Rain
>>genre        : Fantasy
>>price        : 7.95
>>publish_date : 2000-12-16
>>description  : A former architect battles corporate zombies,
                 an evil sorceress, and her own childhood to become queen
                 of the world.
…
```

Windows PowerShell a parfaitement réussi à intégrer le XML à son écosystème ; il n'y a pas de différence entre la manipulation d'une structure XML et celle de tout autre objet.

20

La gestion de l'Active Directory

Une infrastructure est constituée de tout un ensemble d'éléments matériels, logiciels ou même humains. Si, à l'échelle d'une infrastructure donnée, il faut considérer l'articulation de l'ensemble de ces éléments, la nécessité de les organiser constitue une étape fondamentale.

Active Directory, technologie créée par Microsoft, répond de manière très précise à cette exigence. Ce service d'annuaire, capable de recenser de très nombreuses ressources (utilisateurs, groupes, unités d'organisation, etc.), est une plate-forme dédiée à la centralisation de leur gestion.

Ce chapitre ne décrira pas Active Directory de manière détaillée, mais esquissera plutôt quelques éléments de manipulation à travers les langages de script étudiés dans cet ouvrage. Une connaissance du mode de fonctionnement d'Active Directory est toutefois un prérequis afin de mieux saisir la nature des opérations réalisées.

Active Directory est une implémentation du standard LDAP *(Lightweight Directory Access Protocol)*. Il s'agit d'une immense base de données et il est très fréquent de l'interroger pour en extraire différentes informations. Les outils disponibles pour cela varient d'une plate-forme à une autre.

KSH

En ce qui concerne la programmation Shell dans son ensemble, c'est l'outil `ldapsearch` qui sert à manipuler Active Directory.

Avec l'outil ldapsearch

`ldapsearch` est un excellent outil pour effectuer des recherches d'informations au sein d'une infrastructure LDAP. Essayons, par exemple, de lister les machines existant dans un domaine Active Directory donné :

```
kais@redhatlinux$ ldapsearch -LLL -H ldap://contoso.com -x -D 'contoso\develslk' -w
'1/*569@#' -b 'dc=contoso,dc=com' '(objectClass=Computer)'

…
>>dn: CN=suselinux,CN=Computers,DC=contoso,DC=com
>>objectClass: top
>>objectClass: person
>>objectClass: organizationalPerson
>>objectClass: user
>>objectClass: computer
>>cn: suselinux
>>distinguishedName: CN=suselinux,CN=Computers,DC=contoso,DC=com
>>instanceType: 4
>>whenCreated: 20140526144604.0Z
>>whenChanged: 20140526144604.0Z
>>uSNCreated: 201989
>>uSNChanged: 201997
>>name: suselinux
>>objectGUID:: 3x5IsaJsEUab5h9pjpnKlg==
>>userAccountControl: 69632
>>badPwdCount: 0
>>codePage: 0
>>countryCode: 0
>>badPasswordTime: 0
>>lastLogoff: 0
>>lastLogon: 130455930171717827
>>localPolicyFlags: 0
>>pwdLastSet: 130455891645159940
>>primaryGroupID: 515
>>objectSid:: AQUAAAAAAAUVAAAAg/5igBPoPlW6jymlzgQAAA==
>>accountExpires: 9223372036854775807
>>logonCount: 5
>>sAMAccountName: suselinux$
>>sAMAccountType: 805306369
>>dNSHostName: suselinux.contoso.com
>>servicePrincipalName: HOST/suselinux.contoso.com
>>servicePrincipalName: HOST/SUSELINUX
>>objectCategory: CN=Computer,CN=Schema,CN=Configuration,DC=contoso,DC=com
>>isCriticalSystemObject: FALSE
>>dSCorePropagationData: 16010101000000.0Z
>>lastLogonTimestamp: 130455891648109159

>># refldap://ForestDnsZones.contoso.com/DC=ForestDnsZones,
DC=contoso,DC=com
```

```
>># refldap://DomainDnsZones.contoso.com/DC=DomainDnsZones,
DC=contoso,DC=com

>># refldap://contoso.com/CN=Configuration,DC=contoso,DC=com
…
```

Pour réaliser une requête bien précise, un nombre conséquent d'options doit être utilisé.

- `-LLL` précise un format de données.
- `-H` indique l'URI *(Uniform Resource Identifier)* d'un serveur LDAP.
- `-x` utilise une authentification simple.
- `-D` indique le nom d'utilisateur (plusieurs formats possibles).
- `-w` indique le mot de passe.
- `-b` spécifie un point de départ de la recherche.

Le filtre `'(objectClass=Computer)'` cible la recherche et ne retourne que les objets LDAP représentant les ordinateurs du domaine `contoso.com`.

Voici une autre requête LDAP :

```
kais@redhatlinux$ ldapsearch -LLL -H ldap://contoso.com -x -D '
contoso\develslk' -w '1/*569@#' -b 'dc=contoso,dc=com' '(sAMAccountName=KimA)'
```

Cette requête cible un utilisateur précis et retourne des informations en lien avec son compte Active Directory :

```
>>dn: CN=Kim Akers,OU=Accounts,DC=contoso,DC=com
>>objectClass: top
>>objectClass: person
>>objectClass: organizationalPerson
>>objectClass: user
>>cn: Kim Akers
>>sn: Akers
>>givenName: Kim
>>distinguishedName: CN=Kim Akers,OU=Accounts,DC=contoso,DC=com
>>instanceType: 4
>>whenCreated: 20100726174059.0Z
>>whenChanged: 20100806173148.0Z
>>displayName: Kim Akers
>>uSNCreated: 61693
>>memberOf: CN=Voicemail Users,CN=Users,DC=contoso,DC=com
>>memberOf: CN=All Users,CN=Users,DC=contoso,DC=com
>>uSNChanged: 115610
>>proxyAddresses: sip:KimA@contoso.com
>>proxyAddresses: SMTP:KimA@contoso.com
>>name: Kim Akers
>>objectGUID:: 2wKSLAzrhUiTQAVJBuOIYg==
>>userAccountControl: 66048
...
```

Cette fois-ci, le filtre utilisé est `'(sAMAccountName=KimA)'` : la requête est donc circonscrite à l'aide de l'attribut `sAMAccountName`.

Les deux exemples que nous venons de voir illustrent le grand intérêt de l'outil `ldapsearch`. L'intégration de ce dernier au sein de scripts Korn Shell est un atout majeur pour ce langage de script.

> **IMPORTANT** **Accéder aux données contenues dans Active Directory**
>
> Afin d'interroger un service d'annuaire Active Directory, il est fondamental de satisfaire un certain nombre de prérequis (authentification, droits d'accès, bonne intégration de clients LDAP au domaine concerné, etc.), sous peine de ne pas pouvoir accéder aux données recherchées.

Perl

Perl est également un excellent moyen d'accéder à Active Directory. Dans cette partie, nous solliciterons une nouvelle fois le module `Win32::OLE`, décidément très utile dans l'interaction avec un environnement Windows.

Avec le module Win32::OLE

Le script suivant interroge un annuaire Active Directory pour lister l'ensemble des utilisateurs du domaine `contoso.com` :

```
use Win32::OLE;

$Domain= "dc=contoso,dc=com"; (1)
$Base   = "<LDAP://" . $Domain . ">;"; (2)
$Filter = "(&(objectclass=user)(objectcategory=person));"; (3)
$Attrs = "name;"; (4)
$Scope = "subtree"; (5)

$Conn = Win32::OLE->CreateObject("ADODB.Connection"); (6)
$Conn->{Provider} = "ADsDSOObject";
$Conn->Open;
$ADORS = $Conn->Execute($Base . $Filter . $Attrs . $Scope); (7)
$ADORS->MoveFirst;
while (not $ADORS->EOF) {
    print $ADORS->Fields(0)->Value,"\n"; (8)
    $ADORS->MoveNext;
}
```

La première étape consiste à définir les éléments de configuration de la requête : le nom du domaine ciblé (1) servant de point de départ (2) pour la requête, le filtre (3) associé à cette dernière, ainsi que l'attribut (4) qui nous intéresse. Il est également possible de spécifier l'étendue de la requête (5) ; dans notre cas, la requête Active Directory va couvrir toute l'arborescence de domaine.

Il nous faut maintenant créer un objet `ADODB.Connection` (6) autorisant une connexion au service d'annuaire Active Directory. Les éléments de configuration de la requête sont ensuite transmis (7) à l'objet de connexion pour que la requête soit exécutée et que la liste des noms d'utilisateurs du domaine `contoso.com` nous soit retournée (8) :

```
>>Administrator
>>Guest
>>krbtgt
>>ADRMS
>>RTCService
>>RTCComponentService
>>Aaron Con
>>Adam Barr
>>Alan Brewer
>>Allison Brown
>>Ann Beebe
>>Arlene Huff
>>Arno Bost
>>Beth Gilchrist
>>Bob Kelly
>>Brian Johnson
>>Carlos Grilo
>>Claire O'Donnell
>>Dan Hough
>>Dieter Zilch
>>Ed Banti
>>Eli Bowen
>>Elisabetta Scotti
>>Florian Voss
>>Frank Miller
>>Giorgio Veronesi
>>Hao Chen
>>Holly Holt
>>Hugo Garcia
>>Isabel Martins
…
```

Ce script constitue un modèle de base dans la collecte d'informations en lien avec Active Directory, car un changement de filtre de requête sera suffisant pour obtenir d'autres informations.

Python

En Python, tout comme en Perl, il existe un nombre non négligeable de voies d'accès au service d'annuaire. Nous nous focaliserons sur un module, `active_directory`, proposant un mode de manipulation des données tout à fait aisé.

Avec le module active_directory

Pour mieux illustrer le mode de fonctionnement du module `active_directory`, nous allons, à partir du domaine `contoso.com`, lister les groupes auxquels appartient l'utilisateur `Kim Akers`. Le module `active_directory` doit d'abord être importé :

```
import active_directory
```

Pour collecter ensuite les informations relatives au compte utilisateur `Kim Akers`, nous devons utiliser la fonction `find_user()` pour le trouver au sein du domaine `contoso.com` :

```
kim = active_directory.find_user("Kim Akers")
```

La propriété listant les groupes auxquels est lié l'utilisateur `Kim Akers` se nomme `memberOf`. Il ne nous reste donc plus qu'à itérer afin d'extraire chaque groupe et en lister les membres :

```
for group in kim.memberOf:
      print "\nKim Akers is member of group", "(" + group.cn + ")"
      print "Members of group", group.cn
      for member in group.member:
          print member
```

Voici le résultat de la requête :

```
>>Kim Akers is member of group (Voicemail Users)
>>Members of group Voicemail Users
>>LDAP://CN=Ben Miller,OU=Mailboxes,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Erik Ryan,OU=IT,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Zheng Mu,OU=Mailboxes,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Yan Li,OU=Mailboxes,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Ursula Fliegl,OU=Mailboxes,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Tomas Navarro,OU=IT,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Todd Meadows,OU=Legal,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Terry Adams,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Susana Oliveira,OU=Mailboxes,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Spencer Low,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Shy Cohen,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Shane Kim,OU=Executives,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Russell King,OU=Mailboxes,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Roman Lembeck,OU=Mailboxes,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Ray Chow,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Peter Waxman,OU=Mailboxes,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Pedro Gutierrez,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Paul Duffy,OU=IT,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Pablo Rovira Diez,OU=Mailboxes,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Oliver Lee,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Nuno Bento,OU=Mailboxes,OU=Accounts,DC=contoso,DC=com
...
```

```
>>Kim Akers is member of group (All Users)
>>Members of group All Users
>>LDAP://CN=Brian Johnson,CN=Users,DC=contoso,DC=com
>>LDAP://CN=Ben Miller,OU=Mailboxes,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Erik Ryan,OU=IT,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Zheng Mu,OU=Mailboxes,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Yan Li,OU=Mailboxes,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Ursula Fliegl,OU=Mailboxes,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Tomas Navarro,OU=IT,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Todd Meadows,OU=Legal,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Terry Adams,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Susana Oliveira,OU=Mailboxes,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Spencer Low,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Shy Cohen,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Shane Kim,OU=Executives,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Russell King,OU=Mailboxes,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Roman Lembeck,OU=Mailboxes,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Ray Chow,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Peter Waxman,OU=Mailboxes,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Pedro Gutierrez,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Paul Duffy,OU=IT,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Pablo Rovira Diez,OU=Mailboxes,OU=Accounts,DC=contoso,DC=com
>>LDAP://CN=Oliver Lee,OU=Accounts,DC=contoso,DC=com
...
```

La fonction `find_user()` n'est qu'un exemple parmi d'autres illustrant de manière adéquate les possibilités offertes par le module `active_directory`. Ce dernier constitue un très bon moyen d'interagir avec une source de données Active Directory.

VBS

Un ensemble d'interfaces nommé ADSI *(Active Directory Service Interfaces)* constitue la voie d'accès par excellence au service d'annuaire Active Directory. Le principe est simple : toutes les ressources exposées au sein d'une source de données Active Directory doivent être accessibles, ce qui fait d'ADSI un interlocuteur essentiel invoqué à partir du matériau Visual Basic Scripting.

Avec ADSI

En VBS, nous allons solliciter ADSI dans le but de créer une unité d'organisation que nous nommerons `Research and Development` dans le domaine `contoso.com`. Commençons par établir une connexion au domaine `contoso.com` (1). Nous invoquons ensuite la méthode `Create()` pour précisément créer l'unité d'organisation `Research and Development`.

Deux arguments sont passés à `Create()` (2) : l'objet à créer (une unité d'organisation, `OrganizationalUnit`) et le nom relatif qui lui est associé (`ou=Research and Development`). Enfin,

la méthode `SetInfo()` fixe cet ajout (3). En très peu de lignes de code, ADSI nous a donc permis de créer une unité d'organisation :

```
Set objDomain = GetObject("LDAP://dc=contoso,dc=com") (1)

Set objOU = objDomain.Create("OrganizationalUnit", "ou=Research and Development") (2)
objOU.SetInfo (3)
```

L'exemple que nous avons étudié illustre à quel point Visual Basic Scripting est un très bon matériau pour réaliser des opérations en lien avec Active Directory.

Windows PowerShell

PowerShell dispose d'un module entièrement dédié à la gestion de l'Active Directory. Nous l'utiliserons donc dans cette partie pour juger sur pièce de ses performances.

Avec le module ActiveDirectory

Il existe, depuis la version 2008 R2 de Windows Server (par ailleurs la plus répandue à l'heure actuelle), un module dont l'orientation est spécialement réservée à l'interaction avec le service d'annuaire Active Directory. Commençons par importer ce module :

```
PS> Import-Module ActiveDirectory
```

Listons les commandes à notre disposition :

```
PS> Get-Command -Module ActiveDirectory | select Name

>>Name
>>----
>>Add-ADComputerServiceAccount
>>Add-ADDomainControllerPasswordReplicationPolicy
>>Add-ADFineGrainedPasswordPolicySubject
>>Add-ADGroupMember
>>Add-ADPrincipalGroupMembership
>>Clear-ADAccountExpiration
>>Disable-ADAccount
>>Disable-ADOptionalFeature
>>Enable-ADAccount
>>Enable-ADOptionalFeature
>>Get-ADAccountAuthorizationGroup
>>Get-ADAccountResultantPasswordReplicationPolicy
>>Get-ADComputer
>>Get-ADComputerServiceAccount
>>Get-ADDefaultDomainPasswordPolicy
>>Get-ADDomain
```

```
>>Get-ADDomainController
>>Get-ADDomainControllerPasswordReplicationPolicy
>>Get-ADDomainControllerPasswordReplicationPolicyUsage
>>Get-ADFineGrainedPasswordPolicy
>>Get-ADFineGrainedPasswordPolicySubject
>>Get-ADForest
>>Get-ADGroup
>>Get-ADGroupMember
>>Get-ADObject
>>Get-ADOptionalFeature
>>Get-ADOrganizationalUnit
>>Get-ADPrincipalGroupMembership
>>Get-ADRootDSE
>>Get-ADServiceAccount
>>Get-ADUser
...
```

La cmdlet `New-ADuser` sert à créer un ou plusieurs compte(s) utilisateur(s) Active Directory, comme le montre l'aide qui lui est associée :

```
PS> help New-ADUser

NAME
    New-ADUser

SYNOPSIS
    Creates a new Active Directory user.

SYNTAX
    New-ADUser [-Name] <string> [-AccountExpirationDate
<System.Nullable[System.DateTime]>] [-AccountNotDelegated <System.Nullable[bool]>]
[-AccountPassword <SecureString>]
[-AllowReversiblePasswordEncryption <System.Nullable[bool]>]
[-AuthType {Negotiate | Basic}] [-CannotChangePassword <System.Nullable[bool]>] [-
Certificates <X509Certificate[]>]
[-ChangePasswordAtLogon <System.Nullable[bool]>] [-City <string>]
[-Company <string>] [-Country <string>] [-Credential <PSCredential>]
[-Department <string>] [-Description <string>] [-DisplayName <string>]
[-Division <string>] [-EmailAddress <string>] [-EmployeeID <string>]
[-EmployeeNumber <string>] [-Enabled <System.Nullable[bool]>]
[-Fax <string>] [-GivenName <string>] [-HomeDirectory <string>]
[-HomeDrive <string>] [-HomePage <string>] [-HomePhone <string>]
[-Initials <string>] [-Instance <ADUser>] [-LogonWorkstations <string>] [-Manager
<ADUser>] [-MobilePhone <string>] [-Office <string>]
[-OfficePhone <string>] [-Organization <string>] [-OtherAttributes <hashtable>] [-
OtherName <string>] [-PassThru <switch>]
[-PasswordNeverExpires <System.Nullable[bool]>] [-PasswordNotRequired
<System.Nullable[bool]>] [-Path <string>] [-POBox <string>] [-PostalCode <string>] [-
ProfilePath <string>] [-SamAccountName <string>] [-ScriptPath <string>] [-Server
<string>] [-ServicePrincipalNames <string[]>]
```

```
[-SmartcardLogonRequired <System.Nullable[bool]>] [-State <string>]
[-StreetAddress <string>] [-Surname <string>] [-Title <string>]
[-TrustedForDelegation <System.Nullable[bool]>] [-Type <string>]
[-UserPrincipalName <string>] [-Confirm] [-WhatIf] [<CommonParameters>]

DESCRIPTION
    The New-ADUser cmdlet creates a new Active Directory user. You can set commonly
used user property values by using the cmdlet parameters.

   Property values that are not associated with cmdlet parameters can be set by using
the OtherAttributes parameter.
   When using this parameter be sure to place single quotes around the attribute name
as in the following example.

    New-ADUser -SamAccountName "glenjohn" -GivenName "Glen" -Surname "John" -
DisplayName "Glen John" -Path 'CN=Users,DC=fabrikam,DC=local' -OtherAttributes
@{'msDS-PhoneticDisplayName'="GlenJohn"}

    You must specify the SAMAccountName parameter to create a user.

    You can use the New-ADUser cmdlet to create different types of user accounts such
as iNetOrgPerson accounts. To do this in AD DS, set the Type parameter to the LDAP
display name for the type of account you want to create. This type can be any class
in the Active Directory schema that is a subclass of user and that has an object
category of person.

   The Path parameter specifies the container or organizational unit (OU) for the new
user. When you do not specify the Path parameter, the cmdlet creates a user object in
the default container for user objects in the domain.

    The following methods explain different ways to create an object by using this
cmdlet.

    Method 1: Use the New-ADUser cmdlet, specify the required parameters, and set any
additional property values by using the cmdlet parameters.

    Method 2: Use a template to create the new object. To do this, create a new user
object or retrieve a copy of an existing user object and set the Instance parameter
to this object. The object provided to the Instance parameter is used as a template
for the new object. You can override property values from the template by setting
cmdlet paramet
ers. For examples and more information, see the Instance parameter description for
this cmdlet.

    Method 3: Use the Import-CSV cmdlet with the New-ADUser cmdlet to create multiple
Active Directory user objects. To do this, use the Import-CSV cmdlet to create the
custom objects from a comma-separated value (CSV) file that contains a list of object
properties. Then pass these objects through the pipeline to the New-ADUser cmdlet to
create the user objects.
```

```
RELATED LINKS
    Online version: http://go.microsoft.com/fwlink/?LinkID=144970
    Get-ADUser
    Set-ADUser
    Remove-ADUser

REMARKS
    To see the examples, type: "get-help New-ADUser -examples".
    For more information, type: "get-help New-ADUser -detailed".
    For technical information, type: "get-help New-ADUser -full".
```

Dans la section dédiée à VBS, nous avons élaboré une unité d'organisation nommée `Research and Development`. Nous allons maintenant créer un compte utilisateur nommé `Rebecca Miles`, que nous affecterons à cette nouvelle unité d'organisation :

```
PS> New-ADUser -SamAccountName "RebeccaM" -GivenName "Rebecca" -Surname "Miles" -Name
"Rebecca Miles" -Path 'OU=Research and Development,DC=contoso,DC=com' -
AccountPassword (Read-Host -AsSecureString "ADPassword") -ChangePasswordAtLogon
$True -Passthru | Enable-ADAccount
```

Les paramètres `-SamAccountName`, `-GivenName`, `-SurName` et `-Name` spécifient l'identité de l'utilisateur. La localisation du compte au sein de l'unité d'organisation `Research and Development` est établie à l'aide du paramètre `-Path`. Enfin, `-AccountPassword` et `-ChangePasswordAtLogon` indiquent à eux deux un mot de passe temporaire devant être changé lors de la connexion du compte utilisateur au domaine `contoso.com`, évidemment après activation (`Enable-Account`).

Si le compte a bien été créé, nous devrions le voir au sein de l'annuaire :

```
PS> Get-ADUser -Filter 'Name -like "Rebecca Miles"'

>>DistinguishedName : CN=Rebecca Miles,OU=Research and
>>Development,DC=contoso,DC=com
>>Enabled           : True
>>GivenName         : Rebecca
>>Name              : Rebecca Miles
>>ObjectClass       : user
>>ObjectGUID        : 6d171335-401d-488f-bc2b-9c60a081d2a5
>>SamAccountName    : RebeccaM
>>SID               : S-1-5-21-2153971331-1430186003-2770964410-1232
>>Surname           : Miles
>>UserPrincipalName :
```

Voilà ! Utiliser PowerShell pour interroger l'Active Directory n'est en réalité pas très compliqué. Le module `ActiveDirectory` prend en charge une grande partie du travail, ce qui évite de recourir à de la programmation pure.

Index

$? 147–149
$Matches 169
$This 135
'(objectClass=Computer)' 251
'(sAMAccountName=KimA)' 252
--shell 236
-AccountPassword 259
-CaseSensitive 170
-ChangePasswordAtLogon 259
-Class 191
-ComputerName 192, 204
-ConnectionSpecificSuffix 232
-EntryType 218
-GivenName 259
-InstanceId 219
-InterfaceAlias 231
-ItemType 179
-List 217
-LogName 217
-match 169
-n 209
-Name 179, 204, 259
-Namespace 193
-Newest 217
-notmatch 170
-PassThru 205
-Path 179, 259
-RegisterThisConnectionsAddress 232
-SamAccountName 259
-SurName 259
-Value 179
.evt 215
.NET 63, 93, 133, 155
.pm 140
.psm1 144
.py 142
/etc/passwd 138–139, 142, 161
/etc/shadow 148
/home 138, 141
/tmp 138
> 175
>> 174
@INC 140
[Char] 66
[Object] 66

A

accolades 115
Active Directory 249
Active Directory Service Interfaces (ADSI) 255
active_directory 253–255
ActiveDirectory 256
Add-Content 179
Add-Member 134
AddDays() 63
ADODB.Connection 253
adresse IP 223–224, 227, 232
ADSI 255
ADSI (Active Directory Service Interfaces) 255
alias 47
arborescence XML 242
argument 129
Asc() 60
attribut 56, 129, 131–132
authentification 252
awk 138, 140, 161

B

BackupEventLog() 215–216
bloc
  d'instructions 100, 102, 104, 107–111, 115
  de code 151, 154
  Param 120

boucle
do until..loop 108
do while..loop 107
for 99, 102, 105, 109, 117
For Each 118
for each..next 107
for..next 106
foreach 86, 103, 110
until 104
while 100, 104–105, 111
Bourne Again Shell 7

C

Caption 186
case..in..esac 70
cat 148, 174, 237
cd 48, 238
chdir() 150
chomp() 51
chop() 51
chr() 51, 55
Cint() 60
Class 130
class 129
classe 123, 125–126, 128–129, 131, 133–134, 184
créer 123, 127, 130
instancier 124, 127, 130
clause
default 74, 76
PROCESS 120
clé 27, 87, 97
Clear-EventLog 219
ClearEventLog() 215–216
Clng() 60
cmdlet 133–134, 144, 170–171, 178–179, 189, 191, 203–205, 216–219, 230–232
Get-Member 94
Get-WmiObject 120
collection 99
de valeurs 103, 107, 110
COM 9, 90, 133, 137, 167, 177
commande
alias 47
cd 48
echo 48
exec 49
exit 49
export 49
jobs 49
kill 49
let 50
native 47
print 50
pwd 50
service 195
test 68, 114
typeset 50, 83
umask 50
unset 50
commentaire 13–15, 17, 19
commenter
bloc de lignes 14, 16, 18, 20, 22
ligne 13, 15, 18–19, 21
Component Object Model 9, 137
Comprehensive Perl Archive Network 184
compteur 106, 109, 111
condition 100, 102, 104, 107, 111
configuration
DNS 229–230
IP 221–222
réseau 227
réseau statique 227
constructeur 127–129, 131
CPAN 184
Create() 255
CreateObject() 62, 177
CreateTextFile() 177

D

Date() 59
DateAdd() 59
Day() 60
def 129
default 74, 76, 81
defined() 52
DeleteFile() 178
dictionnaire
anonyme 127, 240

créer 84, 86, 89, 91, 95
manipuler 85, 87, 89, 91, 96
die() 150
DisableIPSec() 229
DNS 229
DnsClient 230
Document Object Model (DOM) 243
Domain Name System 229
droits
d'accès 252
de lecture 149

E

echo 48, 173
ElementTree 242
Enable-Account 259
EnableStatic() 228
End Class 130
end() 165
espace de noms 184, 188
eval() 56
événement
lister 210, 212
nombre 214
exception 151, 155
exec 48–49
ExecQuery() 183, 185
Execute() 168
exit 49
export 49, 138
expression régulière 159, 162, 164–168, 171
Extensible Markup Language 233
extension 140, 142, 144, 215

F

FileSize 214
find_user() 254–255
fonction 113, 127–128, 138–139, 142, 144
array() 90
Asc() 60
chomp() 51
chop() 51
chr() 51, 55
Cint() 60
Clng() 60
CreateObjet() 62
Date() 59
DateAdd() 59
Day() 60
defined() 52
définir 113, 115–117, 119
dict() 89
eval() 56
exemple 114–117, 120
FormatPercent() 61
func_fexists() 114
func_loop 116
Get-Svc 120
Get_Logs() 118
getattr() 56
hasattr() 56
Hex() 61
hex() 52
id() 30
isinstance() 56
issubclass() 57
join() 52, 87
keys() 87
lc() 52
Lcase() 61
len() 57, 88
length() 53
max() 57
min() 57
Now() 60
ord() 53, 58
print() 29
q() 53
qq() 53
qw() 54
range() 58, 117
Replace() 62
Rnd() 61
round() 58
scalar() 86
sort() 54, 87
sorted() 58, 89
split() 55

sum() 59
Trim() 62
type() 30
uc() 55
Ucase() 62
values() 87
fonctions 47
For Each..In..Next 168
For Each..in..Next 188
foreach 86, 197, 241
Format-List 217
FormatPercent() 61
FPATH 137–138
FreePhysicalMemory 182

G

gestion
d'erreurs 147, 151, 153
de fichiers 173
des services 195
gestion d'erreurs 153
Get-Content 179
Get-DnsClient 231
Get-EventLog 216–219
Get-Member 244
Get-Process 171
Get-Service 203
Get-WmiObject 189, 191
getattr() 56
GetObject() 183–184, 188
given 73–74
Global 168
GoTo 0 153
grep 161, 170, 239

H

hasattr() 56
head 208
help 236
Hex() 61
hex() 52

I

if..elif..else 75
if..else 71, 74, 79
if..elsif..else 72
if..then..elif..else..fi 69
if..then..else..end if 77
if..then..else..fi 68
if..then..elseif..then..else..end if 77
ifconfig 221–223
IgnoreCase 168
import 142
Import-Module 144
indentation 116
index 84, 86, 88, 94
information
de configuration réseau 224
DNS 230–232
instance 123
instanciation 124–125, 128–129, 134
instruction 99, 103, 106–107, 151
Dim 31
given 74
interface réseau 221, 229–232
Internet Protocol Security 228
interpréteur 25, 31, 34, 153
IOError 151
IPSec 227–228
isinstance() 56
IsNullOrEmpty() 65
issubclass() 57
iter() 242
itération 99–100

J

jobs 49
join() 52
journaux d'événements 207
effacer 215, 219
lister 216
sauvegarder 215
supprimer 219
taille 214

K

kill 48–49
Korn Shell 7, 124–125, 137, 147, 159, 173, 183, 221, 252

KSH 7, 13, 19, 23, 37, 39, 41, 47, 67, 83, 99, 113, 123, 137, 147, 159, 173, 181, 195, 207, 221, 236, 249
ksh93 123

L

lc() 52
Lcase() 61
LDAP 249
ldapsearch 250, 252
len() 57
length() 53
let 50
Lightweight Directory Access Protocol 249
logger 209

M

max() 57
memberOf 254
méthode 123, 125, 128–131, 134
  Add() 97
  add() 91
  append() 89
  exists() 92
  items() 92
  keys() 89, 92
  remove() 89, 92
  SetArgs() 124
  SetValue() 95
Microsoft 7, 9–10
Microsoft Developer Network 186
Microsoft.XMLDOM 243–244
min() 57
module 137–138, 140, 142, 144, 165, 177, 185–186, 199, 223–224, 230, 239, 241, 252–256
mot-clé
  def 116
  End Function 117
  Function 117
  function 119
  return 115
MSDN 186, 216, 229

N

new() 183
New-ADuser 257
New-Item 178
New-Object 133
niveau de privilèges 216
nom de domaine 229
nom DNS 232
Now() 60

O

objet 123–125, 127–128, 132–134
  Scripting.Dictionary 91
Objet.Classe_WMI() 185
On Error 152–154
open() 152, 174–176
OpenTextFile() 178
opérateur
  arithmétique 38, 40–43
  d'affectation 23–24, 29, 31, 38, 40, 42, 44
  d'appartenance 42
  de comparaison 38, 40–41, 43–44
  de concaténation 43
  de déréférencement 128
  de fractionnement et de jointure 45
  de redirection 39, 45
  de test sur des fichiers 39
  de type 42, 45
  logique 38, 40–41, 43–44
ord() 53, 58
ordre des serveurs DNS 230
os 177
Out-String 171

P

package 127
paquet 127
paramètres positionnels 24
PATH 138
Pattern 168
Perl 7–8, 13–15, 19, 25, 37, 39, 41, 51–53, 55, 59, 71, 85, 102, 115, 127–128, 139, 150, 162, 174, 183, 196, 210, 223, 239, 252
Plain Old Documentation 16
POD 16
PowerShell 13, 19, 21–22

pragmas 15
print 50–56, 59
print() 175
printf 174
Private 130
propriété 124, 129–131, 133–134
  Keys 96
  Values 96
PSCustomObject 133
Public 130
pwd 50
Python 7–8, 13, 17, 19, 28, 37, 41, 55, 59, 74, 88, 105, 116, 128, 142, 151, 164, 176, 185, 199, 214, 227, 241, 253
pywin32 186

Q

q() 53
qq() 53
qw() 54

R

range() 58
re 165
read() 176
RegExp 167
remove() 177
Remove-EventLog 220
Remove-Item 179
répertoire personnel 138, 142
Replace() 62, 64
restart 196
résultat
  false 68, 71
  true 68, 71
Resume Next 152–154
rm 174
Rnd() 61
rootccimv2 188, 192–193
rotation des logs 215
Round() 63
round() 58

S

sAMAccountName 252
Scripting.Dictionary 91
Scripting.FileSystemObject 177
search() 165
sed 159–161
Select-String 170–171
select..case..end select 78
SelectNodes() 246
selectnodes() 244
SelectSingleNode() 247
self 129
séquence de valeurs 105
serveur DNS 230
serveurs DNS 230
service 196
  arrêter 196, 198, 201, 203, 205
  commande 195
  démarrer 195, 198, 200, 202, 204
  lister 196, 199, 201, 203
  redémarrer 196
Set-DnsClient 231–232
SetDNSDomain() 229
SetDNSServerSearchOrder() 230
SetGateway() 228
SetInfo() 256
shell 173, 207–208
sort() 54
sorted() 58
Split() 65
split() 55
start 195
start() 165
Start-Service 204
StartService() 198, 200
StartsWith() 65
stop 196
Stop-Service 205
StopService() 198, 201
structure
  case..in..esac 70
  conditionnelle 67
  given 73
  if..elif..else 75
  if..else 71, 74, 79
  if..elseif..else 80

if..elsif..else 72
if..then..elif..else..fi 69
if..then..else..end if 77
if..then..else..fi 68
if..then..elseif..then..else..end if 77
select..case..end select 78
switch 76, 81
unless 73
sub() 166
SubClassesOf() 188
SubString() 65
sum() 59
switch 76, 81
System.Xml.XmlDocument 244–245

T

tableau 100
anonyme 127, 240
créer 83, 85, 88, 90, 93
manipuler 84, 86, 88, 90, 93
tail 208
test 68
Text 244
TimeWritten 213
ToLower() 63
ToUpper() 64
Trim() 62, 64
TrimEnd() 64
TrimStart() 64
try..catch 154
try..except 151
typeset 48, 50, 83, 123

U

uc() 55
Ucase() 62
umask 50
unless 73
unlink() 176
unset 50

V

variable 23, 114–115
$# 24
$$ 24, 34
$* 24
$? 24, 34
$@ 24
$^ 34
$_ 28, 34
$0 24
$1 24
$ConfirmPreference 36
$DebugPreference 36
$Error 34
$ErrorActionPreference 36
$ErrorView 36
$false 34
$Host 34
$Matches 34
$MaximumAliasCount 36
$MaximumErrorCount 36
$OFS 36
$Profile 34
$PSEmailServer 36
$PSHome 34
$true 34
$VerbosePreference 36
$WarningPreference 36
%ENV 28
@_ 28
@ARGV 28
@INC 28
automatique 34
colLoggedEvents 118
créer 29, 33
d'environnement 24, 34, 137
de préférence 35
déclarer 23, 31
dictionnaire 25, 27
EDITOR 25
HISTFILE 25
HISTSIZE 25
HOME 25
LANG 25
LD_LIBRARY_PATH 25
LogType 118
MANPATH 25
num 117

OSTYPE 25
PATH 25
PS1 25
PWD 25
scalaire 25
SHELL 25
spéciale 24–25, 28, 147–148, 169
tableau 25–26, 32
TEMP 25
USER 25
VBS 7, 9, 13, 19, 31, 37, 42, 59, 76, 90, 106, 117, 130, 152, 154, 166, 177, 187, 201, 215, 229, 243, 255
VBScript 7, 10
Visual Basic Scripting 9, 137, 152, 166, 177, 187, 201, 229, 256

W

warn() 150
Win32::IPConfig 223–224
Win32::OLE 183–184, 252
Win32_Bios 192
Win32_ComputerSystem 182, 184–185
Win32_NetworkAdapterConfiguration 224, 228–229
Win32_NTEventLogFile 214–216
Win32_NTLogEvent 210, 212
Win32_Process 186
Win32_Service 192, 197–199, 201
Windows Management Instrumentation 130–131, 181
Windows PowerShell 7–8, 10, 33, 37, 43, 63, 66, 79, 93, 109, 119, 133, 143, 154, 168, 178, 189, 203, 216, 230, 244, 256
WMI 117, 130–131, 153, 155, 181, 183, 186, 188, 191, 193, 196, 199, 201, 212, 224
requête 120
wmi 185, 199
WMI Query Language 202
WMI() 185
wmic 182–183
WQL 183, 202–203
Write() 178
write() 176–177
WriteLine() 178

X

XML 233
xml.etree.ElementTree 241, 243
XML::Simple 239, 241
XMLin() 241
xmllint 236, 239
XMLout() 241