UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

PowerShell

Sommaire

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Introduction à PowerShell

Qu'est-ce que PowerShell ?

- PowerShell est constitué de deux parties
  1. un shell de **ligne de commande**
  2. un langage de **script**
- Au départ, c'était une infrastructure pour **automatiser des tâches** d'administration dans Windows
- PowerShell est devenu un outil **multiplateforme** utilisé pour de nombreux types de tâches

Intérêt de PowerShell

- Interaction plus rapide qu'avec une interface graphique
- Exécution de lot de commandes : automatisation et intégration continue
- Interaction avec le cloud (Azure, AWS)
- Stockage de script dans des fichiers (extension `.ps1`)

Fonctionnalités de PowerShell

- **Système d'aide intégré :** manuel pour chaque commande, à l'instar de `man` sur Linux
- **Pipeline (`|`) :** permet l'exécution d'une séquence de commandes
- **Alias :** prise en charge de noms alternatifs pour des commandes. Ex: `cls`, `ls`

Particularité de PowerShell

- **objets par-dessus le texte :** PowerShell utilise des objets comme entrée et comme sortie ce qui simplifie la mise en forme et l'extraction
- **applets de commande (ou cmdlets) :** commandes basées sur un runtime commun au lieu d'être des exécutables distincts. Elles sont basées sur .NET Core
- **nombreux types de commandes :** des exécutables natifs, des applets de commande, des fonctions, des scripts ou des alias

Windows PowerShell et PowerShell Core

Windows PowerShell (5.1)

- Basé sur .NET Framework 4.5
- Windows Uniquement
- Installé de base avec Windows
- powershell.exe

PowerShell Core (6.x >)

- Utilisation de .NET Core
- Multiplateforme
- OpenSource
- pwsh.exe

Installation

- Windows PowerShell est installé par défaut sur Windows 8 ou ultérieur
- Pour les autres systèmes d'exploitation : github/powershell

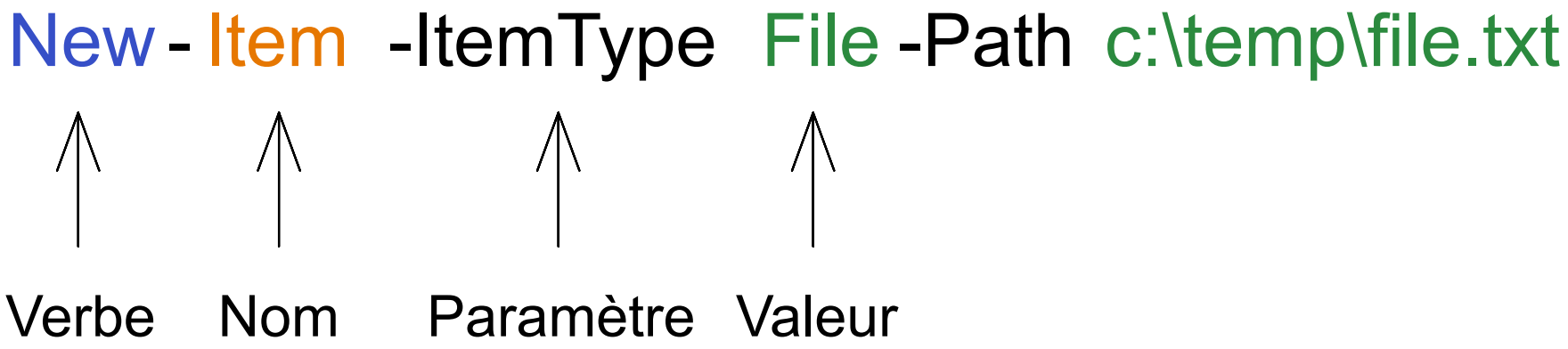
UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Les bases de PowerShell

Applets de commande PowerShell

- PowerShell est fourni avec des centaines de commandes préinstallées
- Les commandes PowerShell sont appelées **cmdlets**
- Le nom de chaque cmdlet est constitué d'une paire `verbe-nom`. Par exemple : **Get**-*Process*
- Il est possible de filtrer des cmdlets sur le verbe ou le nom

Structure d'une commande



Les 3 principales cmdlets de PowerShell

- **Get-Command** : permet de lister les commandes sur le système
- **Get-Help** : indique comment utiliser les commandes et les localiser
- **Get-Member** : obtient les membres, les propriétés et les méthodes des objets

Get-Help

- Exécuter `Get-Help -Name Get-Help` (autre syntaxe: `help Get-Help`)
- La structure de l'aide est la suivante :
  - NOM
  - RÉSUMÉ
  - SYNTAXE : documentation
  - DESCRIPTION
  - LIENS CONNEXES
  - REMARQUES

Get-Command

- `Get-Command` sans aucun paramètre retourne la liste de toutes les commandes du système
- Filtrer sur un nom complet `Get-Command -Name <name>`
- Filtrer sur un verbe `Get-Command -Verb <verb>`
- Filtrer sur un nom `Get-Command -Noun <name>`

Get-Member

- La cmdlet `Get-Member` permet d'obtenir des informations sur les objets qui sont passés en pipeline (`|`)
- Elle affiche les propriétés et les méthodes des objets, ainsi que leur type et leur source

```
$toto = 'toto'
$toto | Get-Member

Name                 MemberType           Definition
----                 ----------           ----------
Clone                Method               System.Object Clone()
...                  ...                  ...
```

Exercice

1. Afficher toutes les commandes contenant le nom "Process"
2. Écrire une commande qui affiche des exemples d'utilisation de Get-Service
3. Écrire une commande qui affiche les méthodes de Get-Process

Historique des commandes

Applet de commande	Alias	Description
Get-History	h	Obtient l'historique des commandes.
Invoke-History	r	Exécute une commande de l'historique.
Add-History		Ajoute une commande à l'historique des commandes.
Clear-History	clhy	Supprime les commandes de l'historique des commandes.

Pipeline

- Les pipelines sont un mécanisme qui permet de passer le résultat d'une commande à une autre commande, sans avoir à utiliser des variables temporaires
- Un pipeline est constitué d'une ou plusieurs commandes séparées par le symbole `|`

```
Get-Command | Out-File C:\temp\file.txt
Get-Content | Sort-Object | Out-File C:\temp\fichierbis.txt
```

Manipuler les résultats

- **Where-Object** : Filtrer les résultats

```
Get-Service | Where-Object -Property Status -eq "Stopped"
```

- **Foreach-Object** : Effectue une action pour chaque résultat reçu par le pipeline

```
Get-Service | ForEach-Object {$_.Name.ToUpper()}
```

- **Group-Object** : Groupe le résultat en fonction d'une propriété

```
Get-Service | Group-Object -Property Status
```

Mise en forme de l'affichage

- PowerShell dispose d'un ensemble de cmdlets qui permettent de contrôler la façon dont les propriétés s'affichent pour certains objets
- Le nom de toutes les applets de commande commence par **Format**
  - **Format-Wide** : par défaut affiche uniquement la propriété par défaut d'un objet
  - **Format-List** : affiche un objet sous forme de liste
  - **Format-Table** : affiche un objet sous forme tabulaire

Les Alias

- Un Alias est un nom alternatif à un Cmdlet ou à un exécutable
- `New-Alias -Name gcd -Value Get-Command`
- Les commandes liés aux Alias :
  - **Get-Alias**: Liste les Alias
  - **New-Alias** : Créé un nouvel Alias
  - **Set-Alias** : Modifie un alias
  - **Remove-Alias** : Supprime un alias
  - **Export-Alias** : Exporte un alias

Exercice

1. Créez un alias nommé **"pss"** pour la commande Get-Process
2. Utiliser le pipeline pour transmettre la liste des processus en les triant par ordre décroissant de mémoire utilisée avec Sort-Object
3. Utiliser Where-Object pour filtrer la sortie de la commande précédente et ne garder que les processus dont le nom commence par "p"
4. Utiliser la mise en forme pour afficher les processus filtrés sous forme de tableau, avec les colonnes suivantes : Name, Id, WorkingSet64

Les providers

- Les providers permettent d'accès au données du système (Fichiers, Registre, Variables d'environnement)
- Pour accéder au registre il faut utiliser le provider **Registry**
- Pour accéder au lecteur C: il faut utiliser le provider **FileSystem**
- Pour lister les différents provider : `Get-PSProvider`
- La colonne Drives indique le chemin d'accès

Utilisation des providers

- Les Cmdlets à utiliser seront toujours les mêmes qu'importe le provider utilisé (créer, modifier, renommer, supprimer un fichier)
- Exemple : New-Item peut créer un fichier, variable d'environnement, ...
- `Get-PSProvider` : Liste les providers disponibles
- `Get-PSDrive` : Liste les lecteurs d'accès aux données

Variables automatique

- Les variables automatiques stockent des informations d'état pour PowerShell
- Ces variables sont créées et gérées par PowerShell

Variable	Description
$HOME	Répertoire de base de l'utilisateur
$PROFILE	Chemin d'accès du profil pwsh
$PSVersionTable	Détail version pwsh

Variables de préférences

- Les variables de préférence affectent l'environnement d'exploitation PowerShell et toutes les commandes s'exécutent dans l'environnement
- Certains Cmdlets ont des paramètres qui permettent de remplacer le comportement de préférence pour une commande spécifique
- Exemple : `$ErrorActionPreference`
- Pour lister l'ensemble des variables : `Get-Variable`

Redirection des résultats

- Par défaut PowerShell renvoie la sortie des cmdlets dans la console

Opérateur	Description	Syntaxe
>	Envoie le flux spécifié à un fichier	n>
>>	Ajoute le flux spécifié à un fichier	n>>
>&1	Redirige le flux spécifié vers le flux de réussite	n>&1

- Flux principaux : 1: succès, 2: erreurs, 3: avertissements ...

Exercice

1. Ecrire un script powershell qui redirige la sortie de la commande Get-Process vers un fichier nommé process.txt dans le dossier C:\Temp
2. Le script doit également rediriger les erreurs éventuelles vers un fichier nommé errors.txt dans le même dossier
3. Le script doit utiliser les opérateurs de redirection PowerShell et ne pas écraser les fichiers existants

Paramètres communs

- Les paramètres communs sont implémentés par PowerShell et sont automatiquement disponibles pour tous les Cmdlets
- **ErrorAction** (ea) : Définir le comportement en cas d'erreur
- **ErrorVariable** (ev) : Variable dans laquelle l'erreur est stockée
- **OutVariable** (ov) : Stocke la sortie d'une cmdlet
- **WhatIf** : Simule le comportement d'une cmdlet
- **Confirm** : Demande une confirmation avant exécution

Paramétrer les erreurs

- Le comportement en cas d'erreur est défini avec le paramètre commune **ErrorAction**
- En fonction des préférences, les erreurs sont stockées dans la variable **$Error**
- **ErrorAction** peut avoir plusieurs valeurs : Break, Suspend, Ignore, Inquire, Continue, Stop, SilentlyContinue

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Scripts PowerShell

Stratégie d'exécution

- La stratégie d’exécution affecte uniquement les commandes qui s’exécutent dans un script
- `Get-ExecutionPolicy` permet de déterminer la stratégie d’exécution des scripts
- `Set-ExecutionPolicy` est utilisée pour modifier la stratégie d’exécution
- **RemoteSigned** exige que les scripts téléchargés soient signés par un éditeur approuvé pour pouvoir être exécutés. Plus d'infos

Exécuter un script

- Un script est un fichier de texte brut qui contient une ou plusieurs commandes PowerShell
- Les scripts PowerShell ont une extension de fichier `.ps1`
- Pour exécuter un script, taper le nom complet et le chemin d'accès complet au fichier de script

```
.\MonPremierScript.ps1
```

Les commentaires

- Il existe deux types de commentaires pour les scripts PowerShell
  - Commentaire en ligne `#`
  - Commentaire multi-lignes `<# #>`

```
# Commentaire inline

<#

Commentaire multi-lignes

#>
```

Les régions

- Les régions permettent de structurer le code d'un script
- Une région débute avec `#region` et se termine avec `#endregion`

```
#region Variable

$age = 12
$age++
Write-Host $age

#endregion
```

Directive #Requires

- La directive `#Requires` empêche l'exécution d'un script sans les éléments requis
- Un script peut inclure plusieurs `#Requires` instructions

```
#Requires -Version 6.0
#Requires -RunAsAdmistrator
#Requires -Module PSReadLine
```

Les variables

- Une variable est uen donnée stockée en mémoire vive
- Elle peut être modifiée à tous moment lors de l'exécution d'un programme
- Dans PowerShell, les variables commencent par le caractère `$`
- Pour affecter une valeur à une variable, on utilise `=`

```
$MaVariable = 12

$MonPrenom = "Arthur"
```

Typage fort

- Il est possible de stocker n'importe quel type d'objet dans une variable
- Une notation permet de typer une variable pour qu'elle ne puisse accepter que ce type : **[type]**$variable

```
[int]$number = 8
[string]$words = "Hello"
$number = "world" # Lève une erreur
```

Les constantes

- Les constantes sont des variables qui ne peuvent pas être modifiée pendant l'exécution d'un script
- Pour déclarer une constante on utilise la cmdlet `New-Variable`

```
New-Variable -Name "MotDePasse" -Value "Secret" -Option Constant
```

La concaténation de chaînes

- La concaténation permet d'assembler plusieurs chaînes ensemble
- Il existe plusieurs techniques pour concaténer des chaînes en Pwsh

1. Opérateur `+`

```
$name = 'Kevin Marquette'
$message = 'Hello, ' + $name # utilisation de l'opérateur '+'
```

2. Substitution de variables

```
$message = "Hello, $first $last."
```

Substitution de commandes

- PowerShell vous permet de demander l'exécution de commandes à l'intérieur de la chaîne avec une syntaxe spéciale `$()`

```
$message = "Time: $($directory.CreationTime)"
```

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Les opérateurs

Opérateurs arithmétiques

Opérateur	Description	Utilisation
+	Addition	6 + 2
-	Soustraction	6 - 2
*	Multiplication	6 * 2
/	Divise des nombres	6 / 2
%	Retourne le reste d'une opération de division	7 % 2

Opérateur d'assignation

Opérateur	Description	Utilisation
=	Opérateur d'affectation	$var = 1
+=	Incrémente la valeur d'une variable	$var += 1
-=	Décrémente la valeur d'une variable	$var -= 1
*=	Multiplie la valeur d'une variable	$var *= 2
/=	Divise la valeur d'une variable	$var /= 3
%=	Reste de la division de la variable	$var %= 2

Opérateurs de comparaison (1/2)

Opérateur	Description	Utilisation
-eq	Égal à	$a -eq $b
-ne	Différent de	$a -ne $b
-gt	Supérieur à	$a -gt $b
-ge	Supérieur ou égal à	$a -ge $b
-lt	Inférieur à	$a -lt $b
-le	Inférieur ou égal à	$a -le $b

Opérateurs de comparaison (2/2)

Opérateur	Description
-Like	Correspondance à l'aide du caractère générique *
-NotLike	Absence de correspondance à l'aide du caractère générique *
-Match	Correspond à l'expression régulière spécifiée
-NotMatch	Ne correspond pas à l'expression régulière spécifiée
-Contains	Détermine si la collection contient une valeur spécifiée
-NotContains	Détermine si une collection ne contient pas de valeur spécifique
-In	Détermine si une valeur spécifiée se trouve dans une collection
-NotIn	Détermine si une valeur spécifiée ne se trouve pas dans une collection
-Replace	Remplace la valeur spécifiée

Opérateurs logiques

Opérateur	Description	Exemple
-and	Retourne TRUE lorsque les deux expressions sont TRUE	($a -gt $b) -and ($a -lt 20)
-or	Retourne TRUE lorsque l'une ou l'autre expression est TRUE	($a -eq $b) -or ($a -ne $c)
-xor	Retourne TRUE lorsque seulement une expression est TRUE	($a -eq 1) -xor ($b -eq 2)
-not ou !	Inverse la valeur de l'expression qui suit	-not ($a -eq $b) ou !($a -eq $b)

Opérateurs de types

Opérateur	Description	Exemple
-is	Retourne vrai si l'objet est une instance du type .NET spécifié	(Get-Date) -is [DateTime]
-isnot	Retourne vrai si l'objet n'est pas une instance du type .NET spécifié	(Get-Date) -isnot [String]
-as	Convertit l'objet en type .NET spécifié si possible, sinon retourne $null	"5/7/07" -as [DateTime]

Opérateurs unaires

Opérateur	Description	Exemple
+	Convertit la valeur en un nombre positif	+(-5) = 5
-	Convertit la valeur en un nombre négatif	-(5) = -5
!	Inverse la valeur booléenne	!($true) = False
-not	Inverse la valeur booléenne	-not ($true) = False
-bnot	Inverse les bits de la valeur	-bnot (0b1010) = 0b0101

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Structure conditionnelle

Définition

- Les scripts doivent souvent prendre des décisions et exécuter une logique différente en fonction de ces décisions
- C’est ce qu’on appelle une **exécution conditionnelle**
- Une instruction ou une valeur doit être évaluée, une autre section de code s'exécute en fonction de cette évaluation
- C’est là précisément qu’intervient l’instruction `if`

L'instruction if

- L’instruction if commence par évaluer l’expression entre parenthèses
- Si l’évaluation génère une valeur **$true**, elle exécute le code entre les accolades, sinon elle passe à l'instruction suivante

```
$condition = $true
if ($condition)
{
    Write-Output "The condition was true"
}
```

L'instruction else

- L’instruction else constitue toujours la dernière partie de l’instruction if lorsqu’elle est utilisée

```
if ( Test-Path -Path $Path -PathType Leaf )
{
    Move-Item -Path $Path -Destination $archivePath
}
else
{
    Write-Warning "$path n'existe pas."
}
```

Instruction elsif

- Il est possible de chaîner des instructions if et else au lieu de les imbriquer à l'aide de l'instruction `elseif`

```
if ( Test-Path -Path $Path -PathType Leaf )
{
  # code
}
elseif ( Test-Path -Path $Path )
{
    Write-Warning "A file was required but a directory was found instead."
}
else
{
}
```

Instruction switch

- L'instruction switch permet d'effectuer plusieurs comparaisons avec une valeur

```
switch ( $var )
{
    'Component'
    {
        'is a component'
    }
    'Role'
    {
        'is a role'
    }
}
```

Opérateur ternaire

- L'opérateur ternaire se comporte comme l'instruction simplifiée if-else
- `<condition> ? <if-true> : <if-false>`

```
$message = (Test-Path $path) ? "Path exists" : "Path not found"
```

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Structure itératives

Instruction While

- While permet d'exécuter un ensemble d'instructions de façon répétée sur la base d'une condition booléenne
- `while (<condition>){<statement list>}`

```
while($val -ne 3)
{
    $val++
    Write-Host $val
}
```

Instruction For

- For permet de répéter l'exécution d'une séquence d'instructions lorsque l'on connaît dès l'entrée le nombre d'itérations souhaitée
- `for (<Init>; <Condition>; <Repeat>) { <Statement list> }`

```
for ($i = 0; $i -lt 10; $i++)
{
  "$i"
}
```

Instruction ForEach

- Foreach permet d'itérer sur une série de valeurs dans une collection d’éléments (ex: tableau)
- `foreach ($<item> in $<collection>){<statement list>}`

```
$letterArray = "a","b","c","d"
foreach ($letter in $letterArray)
{
  Write-Host $letter
}
```

Break, Continue et Return

- L'instruction `Break` permet de casser une boucle
- L'instruction `Continue` permet de passer à l'itération suivante d'une boucle
- L'instruction `Return` permet de quitter l'étendue existante

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Les tableaux

Qu'est-ce qu'un tableau ?

- Un tableau est une structure de données qui sert de collection de plusieurs éléments
- Il est possible de boucler sur le tableau ou d’accéder à certains éléments individuellement à l’aide d’un index
- Le tableau est créé sous la forme d’un bloc séquentiel de mémoire dans lequel chaque valeur est stockée juste à côté de l’autre

Création d'un tableau

Il existe deux syntaxes pour créer des tableaux

1. Méthode `@()`

```
$data = @('Zero','One','Two','Three')
```

2. Liste séparée par des virgules

```
$data = 'Zero','One','Two','Three'
```

Accéder aux éléments

- Pour accéder à un élément d'un tableau on utilise les crochets: `[]`
- Le premier élément est indexé à partir de 0
- Les valeurs négatives permettent d'accéder aux éléments à partir de la fin

```
$data = 'Zero','One','Two','Three'

$data[0] # Zero

$date[-1] # Three
```

Taille d'un tableau

Les tableaux et autres collections possèdent une propriété count qui indique le nombre d’éléments qu’ils contiennent

```
PS> $data.count
4
```

Ajouter un élément

- Il est possible de créer un nouveau tableau sur place et d'y ajouter un élément avec `+=`
- ⚠️ cette opération duplique et créé un nouveau tableau, cela peut engendrer des problèmes de performances

```
$data = @('Zero', 'One', 'Two', 'Three')
$data += 'four'
```

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Les tables de hachages

Qu'est-ce qu'une table de hachage

- Une table de hachage est une structure de données, à l’instar d’un tableau, sauf que que l'on y stocke chaque valeur (objet) à l’aide d’une clé
- Il s’agit d’un magasin de clés/valeurs de base

Création d'une table de hachage

Une table de hachage se créé avec la notation `@{}`

```
$ageList = @{}

$key = 'Kevin'
$value = 36
$ageList.add( $key, $value )
```

Création et affectation

```
$ageList = @{
    Kevin = 36
    Alex  = 9
}
```

Parcourir une table de hache

- Le parcours des éléments se fait à l'aide de foreach en récupérant les clés de parcours via la table `$table.keys`

```
foreach($key in $ageList.keys)
{
    $message = '{0} is {1} years old' -f $key, $ageList[$key]
    Write-Output $message
}
```

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Manipulation de fichiers

Manipulation des dossiers et fichiers

Cmdlets	Description
Get-ChildItem	Liste les fichiers
Copy-Item	Copie un fichier
New-Item	Création de fichiers/dossiers
Remove-Item	Suppression de fichiers/dossiers
Rename-Item	Renomme un fichier
Get-Content	Lecture d'un fichier

📖 Exercice

1. Créer un dossier nommé "test" dans le répertoire courant
2. Copier tous les fichiers .txt du répertoire "C:\Documents" vers le dossier "test"
3. Afficher le contenu du dossier "test" et le nombre de fichiers qu'il contient
4. Supprimer le dossier "test" et tous ses fichiers

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Les fonctions

Qu'est-ce qu'une fonction ?

- Une fonction est une liste d'instructions PowerShell qui a un nom que vous attribuez
- Lorsque vous exécutez une fonction, vous tapez le nom de la fonction
- Les instructions de la liste s'exécutent comme si vous les aviez tapées à l'invite de commandes
- Par convention on les nomme `<ApprovedVerb>-<Prefix><SingularNoun>`
- Il existe une liste de verbes approuvés via `Get-Verb`

Déclarer une fonction

- Une fonction simple se déclare

```
function <function-name> {statements}
```

```
function Get-PSVersion {
    $PSVersionTable.PSVersion
}
```

Fonction paramétrée

- Il existe deux syntaxes pour utiliser des paramètres nommées dans une fonction

1. Avec le mot clé `param`

```
function Get-SmallFiles {
  param($Size)
  Get-ChildItem $HOME | Where-Object {
    $_.Length -lt $Size -and !$_.PSIsContainer
  }
}
```

2. Dans les parenthèses

```
function Add-Numbers([int]$one, [int]$two) {
    $one + $two
}
```

Exercice

1. Ecrire une fonction en powershell qui prend en paramètre un nom de fichier et qui renvoie le nombre de lignes de ce fichier
2. La fonction doit vérifier si le fichier existe et s'il est lisible avant de le traiter
3. Si le fichier n'existe pas ou s'il n'est pas lisible, la fonction doit afficher un message d'erreur et renvoyer une valeur nulle

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Transcription Bash vers PowerShell

Commandes Bash vers PowerShell

Bash	PowerShell	Description
ls	dir, Get-ChildItem	Lister les fichiers et dossier
tree	dir -Recurse, Get-ChildItem -Recurse	Lister les fichiers et dossier
cd	cd, Set-Location	Changer de répertoire
pwd	pwd, $pwd, Get-Location	Affiche le répertoire courant
clear, Ctrl+L, reset	cls, clear	Réinitialiser la console
mkdir	New-Item -ItemType Directory	Créer un dossier
touch test.txt	New-Item -Path test.txt	Créer un fichier vide
cat test1.txt test2.txt	Get-Content test1.txt, test2.txt	Afficher le contenu

Commandes Bash vers PowerShell

Bash	PowerShell	Description
cp ./source.txt ./dest/dest.txt	Copy-Item source.txt dest/dest.txt	Copier un fichier
cp -r ./source ./dest	Copy-Item ./source ./dest -Recurse	Copie de fichier de manière récursive
mv ./source.txt ./dest/dest.txt	Move-Item ./source.txt ./dest/dest.txt	Déplacer un fichier
rm test.txt	Remove-Item test.txt	Supprimer un fichier
rm -r <folderName>	Remove-Item <folderName> -Recurse	Supprimer un dossier

Commandes Bash vers PowerShell

Bash	PowerShell	Description
find -name build*	Get-ChildItem build* -Recurse	Trouver un fichier/répertoire commençant par 'build'
grep -Rin "sometext" --include="*.cs"	Get-ChildItem -Recurse -Filter *.cs | Select-String -Pattern "sometext"	Chercher un fichier de manière récursive
curl https://github.com	Invoke-RestMethod https://github.com	Télécharger depuis le web

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Merci pour votre attention

Des questions ?