

# Module 6 – Fonctions et commandes powershell personnelles

## Table des matières

Passer des paramètres aux scripts.....	2
La variable \$args.....	2
Récupération des paramètres dans un script.....	2
Paramètres nommés typés et gestion des erreurs.....	5
Paramètres typés.....	5
Gestion des erreurs.....	6
Écriture de fonctions.....	7
Fonctions avec paramètres.....	8
Paramètres typés et paramètres non optionnels avec throw.....	8
Paramètres optionnels.....	10
Paramètres obligatoires.....	10
Créer vos propres commande powershell.....	12
Étape 1: Ouvrir un fichier PS1 pour y écrire vos fonctions.....	12
Étape 2: Testez la fonction.....	12
Étape 3: Transformer le fichier sous forme de module.....	13
Étape 4: Copier le fichier de module dans le répertoire des modules du système.....	13
Étape 5: Tester.....	13
Ajouter l'aide à vos fonctions et commandes.....	14
Placement à l'intérieur d'un script.....	14
Placement à l'intérieur d'une fonction.....	14
Étiquettes principales pour les rubriques d'aide.....	15

## À NE PAS OUBLIER

Si vous faites des scripts (donc fichier avec extension .ps1), ne pas oublier de changer la politique d'exécution des scripts pour que Windows accepte que votre script puisse démarrer. Faire la commande:

```
Set-ExecutionPolicy unrestricted
```

## Passer des paramètres aux scripts

Vos scripts peuvent recevoir des paramètres qui sont passés par l'utilisateur à la ligne de commande. On augmente ainsi la flexibilité du script.

### *La variable \$args*

Entrez le script suivant:

```
write-host "Bonjour, $args!"
```

Exécutez-le:

```
.\script.ps1  
Bonjour, !
```

Il n'y a qu'un ! car nous n'avons pas entré de paramètres  
Si on réécrit:

```
.\script.ps1 Stéphane  
Bonjour, Stéphane!
```

Parce que la variable \$args est un tableau, vous avez donc accès à chaque élément du tableau grâce à la notation de crochet [] .

### *Récupération des paramètres dans un script*

Powershell dispose de variables pré-définies qui permettent de récupérer les arguments passés à la ligne de commande.

Il s'agit de la variable pré-définie "args". On peut y accéder en utilisant \$args.

Exemple:

Ainsi, pour aller chercher le premier argument, on peut faire:

```
write-host "Bonjour, $($args[0])!"
```

Essayons le script maintenant:

```
.\script.ps1 Stephane Chasse  
Bonjour, Stephane!
```

Que s'est-il passé avec "chasse" ? Utilisons args[1].

```
Write-host "Bonjour, $($args[0]) $($args[1])  
.\script.ps1 Stéphane Chassé
```

Bonjour, Stéphane Chassé

## Utilisation de paramètres nommés dans les scripts

Vous comprendrez que la variable args est utile mais n'est pas très flexible lorsqu'on veut utiliser des paramètres que nous définissons nous-mêmes. Viens alors les paramètres nommés. Les paramètres nommés sont utilisés avec l'instruction "param" qui définit l'ordre et le nom des paramètres.

La syntaxe est la suivante:

```
param ($NomParamètre1, $NomParamètre2, etc...)
```

Essayons tout ça avec un script qui demande le nom de l'utilisateur et son âge. Le nom du paramètre pour l'utilisateur se nommera "user" et le nom du paramètre pour l'âge se nommera "age".

Ouvrez votre application Powershell GUI et tapez:

```
param($User, $Age)
```

```
write-host "Bonjour $User. "  
if ( $Age -lt 12 )  
{  
    write-host "Vous avez l'age de prendre du lait"  
}  
else  
{  
    if ( $Age -lt 18 )  
    {  
        write-host "Vous avez l'age de prendre du jus"  
    }  
    else  
    {  
        write-host "Vous avez l'age de prendre de l'alcool"  
    }  
}
```

```
}
```

Sauvegardez le script et exécutez-le:

```
.\script2.ps1 Louis 35
```

Bonjour Louis  
Vous avez l'age de prendre de l'alcool

```
.\script2.ps1 -User Jean -Age 16
```

Bonjour Jean  
Vous avez l'age de prendre du jus

Votre script répond maintenant de façon beaucoup flexible à l'aide des paramètres nommés tout comme le ferait un cmdlet de powershell.

Remarques:

- 1) L'ordre des paramètres n'a pas d'importance:  
ex: `script2.ps1 -User Stef -Age 12` ou `script2.ps1 -Age 12 -User Stef`  
provoque le même affichage.
- 2) Le type des paramètres n'est pas vérifié par défaut.

## ***Paramètres nommés typés et gestion des erreurs***

### **Paramètres typés**

Reprenons le script précédent et entrez les valeurs suivantes:

```
./script2 -User 1 -Age Stef  
Bonjour 1.  
Vous avez l'age de prendre de l'alcool
```

Mais pourquoi donc le dernier paramètre nous a fait afficher de prendre de l'alcool ?

Parce que le chiffre 1 est allé dans le paramètre "User" comme chaîne de caractère alors que la chaîne "Stef" n'a pu être évaluée comme il se doit dans les conditions.

Nous allons remédier a cette situation en préfixant les paramètres des types voulues.

La syntaxe est la suivante:

```
param ( [type] $Nom, [type] $Nom2, etc...)
```

Reprenons l'exemple avec cette syntaxe:

```
param([string]$Name, [int] Age)
```

```
./Script3 -User Stef
```

Nous obtenons:

```
Bonjour Stef  
Vous avez l'age de prendre du lait.
```

En fait, l'affichage de la dernière phrase provient du fait que la variable n'a pas été entrée et donc la valeur vaut 0. Comme 0 est plus petit que 12, la phrase "Vous avez l'age de prendre du lait" est affichée.

Il serait donc intéressant de s'assurer que les paramètres sont bel et bien entrés correctement.

## Gestion des erreurs

Dans l'exemple précédent, l'erreur provenait du fait que l'utilisateur avait oublié d'entrer un paramètre (en occurrence, l'âge). Nous allons capter cette erreur en nous assurant qu'il y a bien un paramètre présent.

Nous allons utiliser l'instruction "throw" comme ceci:

```
param([string]$User=$(Throw "Paramètre manquant: -User Nom"),  
      [int]$age=$( Throw "Paramètre manquant: -Age x ou x est un nombre"))
```

Ainsi, si nous entrons:

```
./script3 -User Stef
```

Nous obtenons:

```
Paramètre manquant: -Age x ou x est un nombre  
à C:\Documents and Settings\Stephane\Mes documents\script3.ps1: ligne:2  
caractère:19  
+ [int]$age=$( Throw <<<< "Paramètre manquant: -Age x ou x est un nombre"))
```

```
./script3 -Age 17
```

```
Paramètre manquant: -User Nom  
à C:\Documents and Settings\Stephane\Mes documents\script3.ps1: ligne:1  
caractère:28  
param([string]$User=$(Throw <<<< "Paramètre manquant: -User Nom"),
```

## Écriture de fonctions

Une fonction est très similaire dans l'écriture aux passages de paramètres aux scripts. En effet, une fonction peut accepter des paramètres typés ou non, peut intercepter les erreurs et peut aussi s'assurer que certains paramètres soient obligatoires.

La fonction la plus simple est sans aucun doute, une fonction qui ne traite aucun paramètre. Écrivons une fonction qui fait afficher la phrase "Voici la fonction" à l'écran.

Syntaxe:

```
function Nom_Fonction ( paramètres possibles ici)
{
  corps de la fonction
}
```

Exemple:

Entrez le code suivant dans un script que vous sauvegarderez sous le nom "fonction1" :

```
Function Affiche()
{
  write-host "Voici la fonction"
}
```

Affiche

On l'exécute en tapant:

```
./fonction1
```

on obtient: Voici la fonction

Magnifique mais pas très pratique. En effet, il serait intéressant d'ajouter la possibilité de donner la phrase à faire afficher.

## ***Fonctions avec paramètres***

L'écriture est similaire à l'entrée de paramètres nommés que nous avons utilisé auparavant. Voici la syntaxe:

```
function Nom_Fonction ([type] $NomParamètre1, [type] $NomParamètre2, etc)
```

Il est à noter que le [type] est optionnel à moins que vous vouliez rendre obligatoire le type de la variable utilisée.

Exemple:

réécrivons le script en lui passant la phrase à afficher:

```
function Affiche($Phrase)
{
    write-host $Phrase
}
```

```
Affiche("Allo toi")
Affiche("Bonjour moi")
```

On exécute le script en tapant:

```
./fonction1
```

Nous obtenons:

```
Allo toi
Bonjour moi
```

On peut maintenant faire afficher la phrase que l'on désire avec notre fonction Affiche.

## ***Paramètres typés et paramètres non optionnels avec throw***

Si l'on veut absolument que le type du paramètre entré soit de type "string", nous le mentionnons alors dans l'énoncé de la fonction comme ci-dessous:

```
function Affiche ([string] $Phrase)
```

De même, si on veut être certain que ce paramètre n'a pas été oublié, on peut aussi écrire:

```
function Affiche([string]$Phrase=$(throw "parametre manquant: la phrase a
afficher" ))
{
    write-host $Phrase
}
```

```
Affiche "Allo toi"
Affiche
```

A l'exécution:

```
Allo toi
```

```
parametre manquant: la phrase a afficher
```

```
à C:\Documents and Settings\Stephane\Mes documents\affiche2.ps1: ligne:1 caractère:41
```

- function Affiche([string]\$Phrase=\$(throw <<<< "parametre manquant: la phrase a afficher" ))
-

## ***Paramètres optionnels***

Les fonctions peuvent aussi recevoir des paramètres optionnels. Autrement dit, si l'utilisateur n'utilise pas ce paramètre, la fonction lui donnera une valeur par défaut.

Syntaxe:

```
function Nom([string] $Valeur = "O" )
```

Écrivons le script avec la fonction qui reçoit une chaîne contenant le caractère "O" ou "N" avec une valeur par défaut de "O" :

```
function Reponse ([string] $Valeur = "O")
{
    write-host "Vous avez répondu $Valeur"
}
```

À l'exécution:

Reponse "N" donne :  
Vous avez répondu N

Reponse "O" donne:  
Vous avez répondu O

Reponse donne :  
Vous avez répondu O

## ***Paramètres obligatoires***

On peut également s'assurer que les paramètres entrés soient obligatoires comme ci-dessous:

```
function Nom ([type] [Parameter(Mandatory=$true)] $Montant, $Taxe = 0.05)
```

Écrivons une fonction qui accepte deux paramètres. Le premier de type "double" qui représente le montant payé. Ce paramètre sera de type obligatoire. Le deuxième paramètre de type double représentera le pourcentage de taxe à payer sur le montant. Il sera égal à 0.05 par défaut.

```
function Prix_Total ([double] [Parameter(Mandatory=$true)] $Montant, [double]$Taxe = 0.05)
{
    $Resultat = $Montant * $Taxe

    Write-Host $Resultat
}
```

Qu'obtient-on avec les appels de fonctions suivantes:

Prix\_Total

Prix\_Total 10

Prix\_Total 10 0.1

Autres paramètres possibles avec l'option "parameter"

### **ValidateLength**

Cet attribut spécifie l'intervalle de valeur qu'un paramètre peut accepter. La valeur se situe entre minimum et maximum.

Dans l'exemple suivant, le nom de l'utilisateur doit être entre 1 et 15 caractères de long.

```
[parameter(Mandatory=$true)] [ValidateLength(1, 15)] [String[]] $NomUsager
```

### **ValidatePattern**

Cet attribut permet d'utiliser une expression régulière qui sera comparée au contenu de la variable qui est passée en paramètre à la fonction.

Dans l'exemple suivant, le paramètre doit être une chaîne de caractères comportant 4 chiffres et chaque chiffre doit être entre 0 et 9 inclusivement.

```
[parameter(Mandatory=$true)] [ValidatePattern("[0-9][0-9][0-9][0-9]")] [String[]]  
$Valeur
```

Dans l'exemple suivant, le paramètre doit être une valeur entière comportant 4 chiffres compris entre 0 et 9 inclusivement.

```
[Int32][ValidatePattern("[0-9][0-9][0-9][0-9]")]$number = 1111
```

### **ValidateRange**

Cet attribut spécifie l'intervalle qu'une valeur numérique peut avoir.

Dans l'exemple suivant, le paramètre Chance doit avoir une valeur entre 0 et 10 inclusivement.

```
[parameter(Mandatory=$true)] [ValidateRange(0,10)] [Int] $Chance
```

Dans l'exemple suivant, le paramètre doit avoir une valeur entre 0 et 10 inclusivement et la valeur du paramètre est de 5 par défaut.

```
[Int32][ValidateRange(0,10)]$number = 5
```

# Créer vos propres commande powershell

Il est possible de mixer les deux façons de faire que nous avons vu pour passer les paramètres. On pourrait alors se créer un nouvel applet de commande.

Nous allons créer cette commande de façon à pouvoir l'utiliser de la même façon que n'importe quelle autre commande de powershell. Nous allons pour ce faire, créer une boîte à outil de fonctions que nous pourrons ensuite appeler de la même manière que les autres commandes de powershell.

En résumé, nous voulons que notre commande:

- Réagisse de la même façon que les autres commandes de powershell;
- Accepte des paramètres nommés que l'on pourra utiliser avec la commande. (Ex.: get-childitem [c:\script1](#) ou get-childitem -path [C:\windows](#) );
- Que la commande puisse réagir correctement en lui passant des valeurs par défaut;
- Que la commande donne l'aide et l'usage avec exemples en invoquant la commande get-help;
- Que l'on puisse activer ou désactiver nos commandes sur demandes à l'aide des modules.

Nous allons bâtir la fonction "get-taxe" qui est simple mais qui nous permettra de voir toutes les étapes pour arriver à ce que cette fonction réagisse comme une commande de powershell.

## Étape 1: Ouvrir un fichier PS1 pour y écrire vos fonctions

Créez un fichier que l'on nommera "mesfonctions.ps1

Entrez les lignes suivantes:

```
function get-taxe( [parameter(mandatory=$true)] [double] $Montant, [double] $Taxe = 0.05)
{
    $Resultat = $Montant * $Taxe
    $Resultat
}
```

## Étape 2: Testez la fonction

On peut ici rendre la fonction visible à la session en tapant la ligne suivante:

```
./mesfonctions.ps1
```

**Attention: Ce n'est pas une erreur, il y a vraiment un "." devant ".\mesfonctions.ps1"**

Testez la fonction en tapant comme une commande du système:

```
get-taxe 100
```

### **Étape 3: Transformer le fichier sous forme de module**

Lorsque votre script ou votre fonction fonctionne comme vous le souhaitez, transformez le fichier .PS1 en module en renommant l'extension du fichier PS1 en PSM1.

### **Étape 4: Copier le fichier de module dans le répertoire des modules du système**

- Déplacez-vous dans le répertoire des modules du système:  
C:\Windows\System32\WindowsPowerShell\v1.0\Modules
- Créez un dossier portant le même nom que le nom de votre fichier de module.
- Dans notre exemple, notre module se nomme mesfonctions.psm1. Créez alors un dossier qui porte le nom "mesfonctions" et copier le fichier mesfonctions.psm1 à l'intérieur.

### **Étape 5: Tester**

Ouvrez une nouvelle session powershell.

Vous constaterez que Powershell a automatiquement importé les modules qui se trouvent dans le répertoire des modules du système. Ceci est nouveau depuis la version 3 de powershell.

## Ajouter l'aide à vos fonctions et commandes

La forme la plus simple pour inclure l'aide dans vos scripts ou fonctions se fait par l'intermédiaire de commentaires. La syntaxe de l'aide basée sur des commentaires se présente comme cela :

```
# .< mot clé d'aide>  
# <contenu d'aide>
```

-OU -

```
<#  
.< mot clé d'aide>  
< contenu d'aide>  
#>
```

### Placement à l'intérieur d'un script

Vous pouvez utiliser les commentaires aux endroits suivants :

1. Au début du script
2. A la fin du script

### Placement à l'intérieur d'une fonction

Vous pouvez utiliser les commentaires aux endroits suivants :

1. Avant le mot clé Function
2. Au début du corps de la fonction, c'est-à-dire juste après l'accolade ouvrante
3. À la fin du corps de la fonction, c'est-à-dire juste avant l'accolade fermante

## Exemple :

```
<#  
.mot clé d'aide  
    contenu d'aide  
>  
function MyFunction { }
```

-OU -

```
function MyFunction  
{  
    <#  
    .mot clé d'aide  
        contenu d'aide  
>  
    les instructions de la fonction ici  
}
```

-OU -

```
function MyFunction  
{  
    Les instructions de la fonction ici  
<#  
.mot clé d'aide  
    Contenu d'aide  
>  
}
```

## ***Étiquettes principales pour les rubriques d'aide***

Il s'agit d'utiliser l'une des étiquettes suivantes dans un bloc de commentaire et d'ajouter le texte qui décrit l'élément en question. Les étiquettes principales sont:

```
.SYNOPSIS  
.DESCRIPTION  
.PARAMETER <Parameter-Name>  
.EXAMPLE  
.NOTES  
.LINK
```

Exemple complet :

Vous voulez vous faire une nouvelle commande qui se nommera "get-processovermem".

Celle-ci permettra de recevoir un paramètre (la quantité de mémoire) et de faire afficher les processus actuellement utilisant plus de cette quantité de mémoire vive que la valeur passée en paramètre.

1) Entrez le code suivant dans un fichier que vous nommerez mesfonctions.ps1

```
function get-processovermem ( $qteMemoire)
{
    get-process | where { $_.ws -gt $qteMemoire } | ft Name, ws
}
```

2) Ajoutons l'aide à la commande pour que get-help puisse fonctionner de la même façon sur notre nouvelle commande.

Ajoutez les lignes suivantes dans le fichier mesfonctions.ps1 juste avant la fonction "get-processovermem"

```
<#
```

```
.SYNOPSIS
```

```
Cette commande affiche les processus prenant plus d'une certaine quantité de mémoire.
```

```
.DESCRIPTION
```

```
Cette fonction permet de faire afficher les processus prenant plus d'une certaine quantité de mémoire vive. La quantité est passée en paramètre à la fonction. La fonction retourne les processus répondant au critère et fait afficher le nom et la mémoire utilisée par le processus.
```

```
.EXAMPLE
```

```
ProcessMem 10
```

```
Pour faire afficher les processus prenant plus de 10 Mo de mémoire vive.
```

```
Name      WS
```

```
-----
```

```
audiodg   14229504
```

```
explorer  43925504
```

```
iTunesHelper 11886592
```

```
powershell 47910912
```

SearchIndexer 27639808

svchost 13893632

svchost 31162368

svchost 11456512

svchost 31203328

#>

Voici donc le fichier mesfonctions.ps1 au complet:

<#

.SYNOPSIS

Cette commande affiche les processus prenant plus d'une certaine quantité de mémoire.

.DESCRIPTION

Cette fonction permet de faire afficher les processus prenant plus d'une certaine quantité de mémoire vive. La quantité est passée en paramètre à la fonction. La fonction retourne les processus répondant au critère et fait afficher le nom et la mémoire utilisée par le processus.

.EXAMPLE

get-processovermem 10

Pour faire afficher les processus prenant plus de 10 Mo de mémoire vive.

Name      WS

-----

audiodg    14229504

explorer   43925504

iTunesHelper 11886592

powershell 47910912

SearchIndexer 27639808

svchost    13893632

svchost    31162368

svchost    11456512

svchost    31203328

#>

```
function get-processovermem($memoire)
{
    $memoire = $memoire * 1MB
    get-process | ? {$_.ws -gt $memoire} | ft Name, WS
}
```

3) On teste la commande localement

```
. \mesfonctions.ps 1
```

```
get-processovermem 10
```

Au besoin, on modifie notre code si on s'aperçoit de certaines erreurs.

4) On teste l'aide avec get-help.

```
Ex.: get-help get-processovermem -detailed
```

```
get-help get-processovermem -full
```

```
get-help get-processovermem -examples
```

5) Lorsqu'on est satisfait de la façon que notre nouvelle commande se comporte, on peut alors copier le fichier dans le dossier de Windows comme mentionné à l'étape #4 de la page 13.