Basis Powershell

De doelstellingen

1. 3.1 Providers en aandrijvingen
2. 3.2 Werklocaties
3. 3.3 Artikelen
4. 3.4 Padnamen
5. 3.5 Toepassingsgebied
6. 3.6 ReadOnly en constante eigenschappen
7. 3.7 Overbelasting van de methode en oproepresolutie
8. 3.8 Naam opzoeken
9. 3.9 Typenaam opzoeken
10. 3.10 Automatisch geheugenbeheer
11. 3.11 Uitvoeringsbevel
12. 3.12 Foutafhandeling
13. 3.13 Pijpleidingen
14. 3.14 Modules
15. 3.15 Wildcard-expressies
16. 3.16 Reguliere expressies

3.1 Providers en aandrijvingen

Een *provider* biedt toegang tot gegevens en componenten die anders niet gemakkelijk toegankelijk zouden zijn via de opdrachtregel. De gegevens worden gepresenteerd in een consistente indeling die lijkt op een bestandssysteemstation.

De gegevens die een provider beschikbaar stelt, worden op een *station* weergegeven en de gegevens worden via een *pad* geopend, net als bij een schijfstation. Ingebouwde cmdlets voor elke provider beheren de gegevens op het provider-station.

PowerShell bevat de volgende set ingebouwde providers voor toegang tot de verschillende typen gegevensarchieven:

Aanbieder	Naam station	Beschrijving	Ref.
Alias	Alias:	PowerShell-aliassen	§3.1.1
Milieu	Env:	Omgevingsvariabelen	§3.1.2
Bestandssysteem	A:, B:, C:, ...	Schijfstations, mappen en bestanden	§3.1.3
Functie	Functie:	PowerShell-functies	§3.1.4

Aanbieder	Naam station	Beschrijving	Ref.
Veranderlijke	Veranderlijke	PowerShell-variabelen	§3.1.5

Windows PowerShell:

Aanbieder	Naam station	Beschrijving
Certificaat	Cert:	x509-certificaten voor digitale handtekeningen
Register	HKLM: (HKEY_LOCAL_MACHINE), HKCU: (HKEY_CURRENT_USER)	Windows-register
WSMan	WSMan:	Configuratie-informatie voor WS-Management

De volgende cmdlets hebben betrekking op providers en stations:

- Get-PSProvider: informatie ophalen over een of meer providers
- Get-PSDrive: hier wordt informatie opgehaald over een of meer stations

Het type object dat een provider vertegenwoordigt, wordt beschreven in §4.5.1. Het type object dat een station vertegenwoordigt, wordt beschreven in §4.5.2.

3.1.1 Aliassen

Een *alias* is een alternatieve naam voor een opdracht. Een opdracht kan meerdere aliassen hebben en de oorspronkelijke naam en alle bijbehorende aliassen kunnen door elkaar worden gebruikt. Een alias kan opnieuw worden toegewezen. Een alias is een item (§3.3).

Een alias kan aan een andere alias worden toegewezen; De nieuwe alias is echter geen alias van de oorspronkelijke opdracht.

De provideralias is een platte naamruimte die alleen objecten bevat die de aliassen vertegenwoordigen. De variabelen hebben geen onderliggende items.

Sommige aliassen zijn ingebouwd in PowerShell.

De volgende cmdlets hebben betrekking op aliassen:

- Nieuw-alias: maakt een alias
- Set-Alias: hiermee worden een of meer aliassen gemaakt of gewijzigd
- Get-Alias: hier wordt informatie opgehaald over een of meer aliassen
- Export-Alias: Exports one or more aliases to a file

When an alias is created for a command using , parameters to that command cannot be included in that alias. However, direct assignment to a variable in the Alias: namespace does permit parameters to be included.`New-Alias`

Note

It is a simple matter, however, to create a function that does nothing more than contain the invocation of that command with all desired parameters, and to assign an alias to that function.

The type of an object that represents an alias is described in §4.5.4.

Alias objects are stored on the drive Alias: (§3.1).

3.1.2 Environment variables

The PowerShell environment provider allows operating system environment variables to be retrieved, added, changed, cleared, and deleted.

The provider Environment is a flat namespace that contains only objects that represent the environment variables. The variables have no child items.

An environment variable's name cannot include the equal sign ().`=`

Changes to the environment variables affect the current session only.

An environment variable is an item (§3.3).

The type of an object that represents an environment variable is described in §4.5.6.

Environment variable objects are stored on the drive Env: (§3.1).

3.1.3 File system

The PowerShell file system provider allows directories and files to be created, opened, changed, and deleted.

The file system provider is a hierarchical namespace that contains objects that represent the underlying file system.

Files are stored on drives with names like A:, B:, C:, and so on (§3.1). Directories and files are accessed using path notation (§3.4).

A directory or file is an item (§3.3).

3.1.4 Functions

The PowerShell function provider allows functions (§8.10) and filters (§8.10.1) to be retrieved, added, changed, cleared, and deleted.

The provider Function is a flat namespace that contains only the function and filter objects. Neither functions nor filters have child items.

Changes to the functions affect the current session only.

A function is an item (§3.3).

The type of an object that represents a function is described in §4.5.10. The type of an object that represents a filter is described in §4.5.11.

Function objects are stored on drive Function: (§3.1).

3.1.5 Variables

Variables can be defined and manipulated directly in the PowerShell language.

The provider Variable is a flat namespace that contains only objects that represent the variables. The variables have no child items.

The following cmdlets also deal with variables:

- New-Variable: Creates a variable
- Set-Variable: Creates or changes the characteristics of one or more variables
- Get-Variable: Gets information about one or more variables
- Clear-Variable: Deletes the value of one or more variables
- Remove-Variable: Deletes one or more variables

As a variable is an item (§3.3), it can be manipulated by most Item-related cmdlets.

The type of an object that represents a variable is described in §4.5.3.

Variable objects are stored on drive Variable: (§3.1).

3.2 Working locations

The *current working location* is the default location to which commands point. This is the location used if an explicit path (§3.4) is not supplied when a command is invoked. This location includes the *current drive*.

A PowerShell host may have multiple drives, in which case, each drive has its own current location.

When a drive name is specified without a directory, the current location for that drive is implied.

The current working location can be saved on a stack, and then set to a new location. Later, that saved location can be restored from that stack and made the current working location. There are two kinds of location stacks: the *default working location stack*, and zero or more user-defined *named working location stacks*. When a session begins, the default working location stack is also the *current working location stack*. However, any named working location stack can be made the current working location stack.

The following cmdlets deal with locations:

- Set-Location: Establishes the current working location
- Get-Location: Determines the current working location for the specified drive(s), or the working locations for the specified stack(s)
- Push-Location: Saves the current working location on the top of a specified stack of locations
- Pop-Location: Restores the current working location from the top of a specified stack of locations

The object types that represents a working location and a stack of working locations are described in §4.5.5.

3.3 Items

An *item* is an alias (§3.1.1), a variable (§3.1.5), a function (§3.1.4), an environment variable (§3.1.2), or a file or directory in a file system (§3.1.3).

The following cmdlets deal with items:

- New-Item: Creates a new item
- Set-Item: Changes the value of one or more items
- Get-Item: Gets the items at the specified location
- Get-ChildItem: Gets the items and child items at the specified location
- Copy-Item: Copies one or more items from one location to another
- Move-Item: Moves one or more items from one location to another
- Rename-Item: Renames an item
- Invoke-Item: Performs the default action on one or more items
- Clear-Item: Deletes the contents of one or more items, but does not delete the items (see
- Remove-Item: Deletes the specified items

The following cmdlets deal with the content of items:

- Get-Content: Gets the content of the item
- Add-Content: Adds content to the specified items
- Set-Content: Writes or replaces the content in an item
- Clear-Content: Deletes the contents of an item

The type of an object that represents a directory is described in §4.5.17. The type of an object that represents a file is described in §4.5.18.

3.4 Path names

All items in a data store accessible through a PowerShell provider can be identified uniquely by their path names. A *path name* is a combination of the item name, the container and subcontainers in which the item is located, and the PowerShell drive through which the containers are accessed.

Path names are divided into one of two types: fully qualified and relative. A *fully qualified path name* consists of all elements that make up a path. The

following syntax shows the elements in a fully qualified path name:

Tip

The notation in the syntax definitions indicates that the lexical entity is optional in the syntax.`~opt~`

```
path:
    provider~opt~   drive~opt~   containers~opt~   item

provider:
    module~opt~   provider   ::

module:
    module-name   \

drive:
    drive-name   :

containers:
    container   \
    containers container   \
```

module-name refers to the parent module.

provider refers to the PowerShell provider through which the data store is accessed.

drive refers to the PowerShell drive that is supported by a particular PowerShell provider.

A *container* can contain other containers, which can contain other containers, and so on, with the final container holding an *item*. Containers must be specified in the hierarchical order in which they exist in the data store.

Here is an example of a path name:

```
E:\Accounting\InvoiceSystem\Production\MasterAccount\MasterFile.dat
```

If the final element in a path contains other elements, it is a *container element*; otherwise, it's a *leaf element*.

In some cases, a fully qualified path name is not needed; a relative path name will suffice. A *relative path name* is based on the current working location. PowerShell allows an item to be identified based on its location relative to the current working location. A relative path name involves the use of some special characters. The following table describes each of these characters and provides examples of relative path names and fully qualified path names. The examples in the table are based on the current working directory being set to C:\Windows:

Symbol	Description	Relative path	Fully qualified path
.	Current working location	.\System	C:\Windows\System
..	Parent of the current working location	..\Program Files	C:\Program Files
\	Drive root of the current working location	\Program Files	C:\Program Files
none	No special characters	System	C:\Windows\System

To use a path name in a command, enter that name as a fully qualified or relative path name.

The following cmdlets deal with paths:

- Convert-Path: Converts a path from a PowerShell path to a PowerShell provider path
- Join-Path: Combines a path and a child path into a single path
- Resolve-Path: Resolves the wildcard characters in a path
- Split-Path: Returns the specified part of a path
- Test-Path: Determines whether the elements of a path exist or if a path is well formed

Some cmdlets (such as Add-Content and use file filters. A *file filter* is a mechanism for specifying the criteria for selecting from a set of paths.`Copy-Item`

The object type that represents a resolved path is described in §4.5.5. Paths are often manipulated as strings.

3.5 Scopes

3.5.1 Introduction

A name can denote a variable, a function, an alias, an environment variable, or a drive. The same name may denote different items at different places in a script. For each different item that a name denotes, that name is visible only within the region of script text called its *scope*. Different items denoted by the same name either have different scopes, or are in different name spaces.

Scopes may nest, in which case, an outer scope is referred to as a *parent scope,* and any nested scopes are *child scopes* of that parent. The scope of a name is the scope in which it is defined and all child scopes, unless it is made private. Within a child scope, a name defined there hides any items defined with the same name in parent scopes.

Unless dot source notation (§3.5.5) is used, each of the following creates a new scope:

- A script file

- A script block
- A function or filter

Consider the following example:

```
# start of script
$x = 2; $y = 3
Get-Power $x $y

#function defined in script

function Get-Power([int]$x, [int]$y)
{
if ($y -gt 0) { return $x * (Get-Power $x (--$y)) }

else { return 1 }
}
# end of script
```

The scope of the variables and created in the script is the body of that script, including the function defined inside it. Function defines two parameters with those same names. As each function has its own scope, these variables are different from those defined in the parent scope, and they hide those from the parent scope. The function scope is nested inside the script scope.`$x``$y``Get-Power`

Note that the function calls itself recursively. Each time it does so, it creates yet another nested scope, each with its own variables and .`$x``$y`

Here is a more complex example, which also shows nested scopes and reuse of names:

```
# start of script scope
$x = 2              # top-level script-scope $x created
                    # $x is 2
F1                  # create nested scope with call to function F1
                    # $x is 2
F3                  # create nested scope with call to function F3
                    # $x is 2

function F1 {       # start of function scope
                    # $x is 2
    $x = $true      # function-scope $x created
                    # $x is $true

    & {             # create nested scope with script block
                    # $x is $true
        $x = 12.345 # scriptblock-scope $x created
                    # $x is 12.345
    }               # end of scriptblock scope, local $x goes away
```

```
                    # $x is $true
    F2              # create nested scope with call to function F2
                    # $x is $true
}                   # end of function scope, local $x goes away

function F2 {       # start of function scope
                    # $x is $true
    $x = "red"      # function-scope $x created
                    # $x is "red"
}                   # end of function scope, local $x goes away

function F3 {       # start of function scope
                    # $x is 2
    if ($x -gt 0) {
                    # $x is 2
        $x = "green"
                    # $x is "green"
    }               # end of block, but not end of any scope
                    # $x is still "green"
}                   # end of function scope, local $x goes away
# end of script scope
```

3.5.2 Scope names and numbers

PowerShell supports the following scopes:

- Global: This is the top-most level scope. All automatic and preference variables are defined in this scope. The global scope is the parent scope of all other scopes, and all other scopes are child scopes of the global scope.
- Local: This is the current scope at any execution point within a script, script block, or function. Any scope can be the local scope.
- Script: This scope exists for each script file that is executed. The script scope is the parent scope of all scopes created from within it. A script block does *not* have its own script scope; instead, its script scope is that of its nearest ancestor script file. Although there is no such thing as module scope, script scope provides the equivalent.

Names can be declared private, in which case, they are not visible outside of their parent scope, not even to child scopes. The concept of private is not a separate scope; it's an alias for local scope with the addition of hiding the name if used as a writable location.

Scopes can be referred to by a number, which describes the relative position of one scope to another. Scope 0 denotes the local scope, scope 1 denotes a 1-generation ancestor scope, scope 2 denotes a 2-generation ancestor scope, and

so on. (Scope numbers are used by cmdlets that manipulate variables.)

3.5.3 Variable name scope

As shown by the following production, a variable name can be specified with any one of six different scopes:

```
variable-scope:
    global:
    local:
    private:
    script:
    using:
    workflow:
    variable-namespace
```

The scope is optional. The following table shows the meaning of each in all possible contexts. It also shows the scope when no scope is specified explicitly:

Scope Modifier	**Within a Script File**	**Within a Script Block**	**Within a Function**
global	Global scope	Global scope	Global scope
script	Nearest ancestor script file's scope or Global if there is no nearest ancestor script file	Nearest ancestor script file's scope or Global if there is no nearest ancestor script file	Nearest ancestor script file's scope or Global if there is no nearest ancestor script file
private	Global/Script/Local scope	Local scope	Local scope
local	Global/Script/Local scope	Local scope	Local scope
using	Implementation defined	Implementation defined	Implementation defined
workflow	Implementation defined	Implementation defined	Implementation defined
none	Global/Script/Local scope	Local scope	Local scope

Variable scope information can also be specified when using the family of cmdlets listed in (§3.1.5). In particular, refer to the parameter , and the parameters and for more information.`Scope``Option Private``Option AllScope`

The scope is used to access variables defined in another scope while running scripts via cmdlets like , , or within an *inlinescript-statement*. For example:`using``Start-Job``Invoke-Command`

```
$a = 42
Invoke-Command --ComputerName RemoteServer { $using:a } # returns 42
workflow foo
{
    $b = "Hello"
    inlinescript { $using:b }
}
foo # returns "Hello"
```

The scope workflow is used with a *parallel-statement* or *sequence-statement* to access a variable defined in the workflow.

3.5.4 Function name scope

A function name may also have one of the four different scopes, and the visibility of that name is the same as for variables (§3.5.3).

3.5.5 Dot source notation

When a script file, script block, or function is executed from within another script file, script block, or function, the executed script file creates a new nested scope. For example,

```
Script1.ps1
& "Script1.ps1"
& { ... }
FunctionA
```

However, when *dot source notation* is used, no new scope is created before the command is executed, so additions/changes it would have made to its own local scope are made to the current scope instead. For example,

```
. Script2.ps1
. "Script2.ps1"
. { ... }
. FunctionA
```

3.5.6 Modules

Just like a top-level script file is at the root of a hierarchical nested scope tree, so too is each module (§3.14). However, by default, only those names exported by a module are available by name from within the importing context. The Global parameter of the cmdlet Import-Module allows exported names to have increased visibility.

3.6 ReadOnly and Constant Properties

Variables and aliases are described by objects that contain a number of properties. These properties are set and manipulated by two families of cmdlets

(§3.1.5, §3.1.1). One such property is Options, which can be set to ReadOnly or Constant (using the Option parameter). A variable or alias marked ReadOnly can be removed, and its properties can changed provided the Force parameter is specified. However, a variable or alias marked Constant cannot be removed nor have its properties changed.

3.7 Method overloads and call resolution

3.7.1 Introduction

As stated in §1, an external procedure made available by the execution environment (and written in some language other than PowerShell) is called a *method*.

The name of a method along with the number and types of its parameters are collectively called that method's *signature*. (Note that the signature does not include the method's return type.) The execution environment may allow a type to have multiple methods with the same name provided each has a different signature. When multiple versions of some method are defined, that method is said to be *overloaded*. For example, the type Math (§4.3.8) contains a set of methods called , which computes the absolute value of a specified number, where the specified number can have one of a number of types. The methods in that set have the following signatures:`Abs`

```
Abs(decimal)
Abs(float)
Abs(double)
Abs(int)
Abs(long)
Abs(SByte)
Abs(Int16)
```

In this case, all of the methods have the same number of arguments; their signatures differ by argument type only.

Another example involves the type Array (§4.3.2), which contains a set of methods called Copy that copies a range of elements from one array to another, starting at the beginning of each array (by default) or at some designated element. The methods in that set have the following signatures:

```
Copy(Array, Array, int)
Copy(Array, Array, long)
Copy(Array, int, Array, int, int)
Copy(Array, long, Array, long, long)
```

In this case, the signatures differ by argument type and, in some cases, by argument number as well.

In most calls to overloaded methods, the number and type of the arguments passed exactly match one of the overloads, and the method selected is obvious.

However, if that is not the case, there needs to be a way to resolve which overloaded version to call, if any. For example,

```
[Math]::Abs([byte]10) # no overload takes type byte
[Array]::Copy($source, 3, $dest, 5L, 4) # both int and long indexes
```

Other examples include the type **string** (i.e.; **System.String**), which has numerous overloaded methods.

Although PowerShell has rules for resolving method calls that do not match an overloaded signature exactly, PowerShell does not itself provide a way to define overloaded methods.

Note

Editor's Note: PowerShell 5.0 added the ability to define script-based classes. These classes can contain overloaded methods.

3.7.2 Method overload resolution

Given a method call (§7.1.3) having a list of argument expressions, and a set of *candidate method_s (i.e., those methods that could be called), the mechanism for selecting the_best method_is called_overload resolution.*

Given the set of applicable candidate methods (§3.7.3), the best method in that set is selected. If the set contains only one method, then that method is the best method. Otherwise, the best method is the one method that is better than all other methods with respect to the given argument list using the rules shown in §3.7.4. If there is not exactly one method that is better than all other methods, then the method invocation is ambiguous and an error is reported.

The best method must be accessible in the context in which it is called. For example, a PowerShell script cannot call a method that is private or protected.

The best method for a call to a static method must be a static method, and the best method for a call to an instance method must be an instance method.

3.7.3 Applicable method

A method is said to be *applicable* with respect to an argument list A when one of the following is true:

- The number of arguments in A is identical to the number of parameters that the method accepts.
- The method has M required parameters and N optional parameters, and the number of arguments in A is greater than or equal to M, but less than N.
- The method accepts a variable number of arguments and the number of arguments in A is greater than the number of parameters that the method accepts.

In addition to having an appropriate number of arguments, each argument in A must match the parameter-passing mode of the argument, and the argument type must match the parameter type, or there must be a conversion from the argument type to the parameter type.

If the argument type is ref (§4.3.6), the corresponding parameter must also be ref, and the argument type for conversion purposes is the type of the property Value from the ref argument.

If the argument type is , the corresponding parameter could be instead of `.ref``out``ref`

If the method accepts a variable number of arguments, the method may be applicable in either *normal form* or *expanded form*. If the number of arguments in A is identical to the number of parameters that the method accepts and the last parameter is an array, then the form depends on the rank of one of two possible conversions:

- The rank of the conversion from the type of the last argument in A to the array type for the last parameter.
- The rank of the conversion from the type of the last argument in A to the element type of the array type for the last parameter.

If the first conversion (to the array type) is better than the second conversion (to the element type of the array), then the method is applicable in normal form, otherwise it is applicable in expanded form.

If there are more arguments than parameters, the method may be applicable in expanded form only. To be applicable in expanded form, the last parameter must have array type. The method is replaced with an equivalent method that has the last parameter replaced with sufficient parameters to account for each unmatched argument in A. Each additional parameter type is the element type of the array type for the last parameter in the original method. The above rules for an applicable method are applied to this new method and argument list A.

3.7.4 Better method

Given an argument list A with a set of argument expressions and two application methods and with parameter types and , is defined to be a better method than if the *cumulative ranking of conversions* for is better than that for `.{ E~1~, E~2~, ..., E~N~ }``M~P~``M~Q~``{ P~1~, P~2~, ..., P~N~ }``{ Q~1~, Q~2~, ..., Q~N~ }``M~P~``M~Q~``M~P~``M~Q~`

The cumulative ranking of conversions is calculated as follows. Each conversion is worth a different value depending on the number of parameters, with the conversion of worth N, worth N-1, down to worth 1. If the conversion from to is better than that from to , the accumulates N-X+1; otherwise, accumulates N-X+1. If and have the same

value, then the following tie breaking rules are used, applied in order:E~1~``E~2~``E~N~``E~X~``P~X~``E~X~``Q~X~``M~P~``M~Q~``M~P~``M~Q~

- The cumulative ranking of conversions between parameter types (ignoring argument types) is computed in a manner similar to the previous ranking, so is compared against , against , …, and against . The comparison is skipped if the argument was , or if the parameter types are not numeric types. The comparison is also skipped if the argument conversion loses information when converted to but does not lose information when converted to , or vice versa. If the parameter conversion types are compared, then if the conversion from to is better than that from to , the accumulates N-X+1; otherwise, accumulates N-X+1. This tie breaking rule is intended to prefer the *most specific method* (i.e., the method with parameters having the smallest data types) if no information is lost in conversions, or to prefer the *most general method* (i.e., the method with the parameters with the largest data types) if conversions result in loss of information.P~1~``Q~1~``P~2~``Q~2~``P~N~``Q~N~``$null``E~X~``P~X~``Q~X~``P~X~``Q~X~``Q~X~``P~X~
- If both methods use their expanded form, the method with more parameters is the better method.
- If one method uses the expanded form and the other uses normal form, the method using normal form is the better method.

3.7.5 Better conversion

The text below marked like this is specific to Windows PowerShell.

Conversions are ranked in the following manner, from lowest to highest:

- T~1~[] to where no assignable conversion between and existsT~2~[]``T~1~``T~2~
- T to string where T is any type
- T~1~ to where or define a custom conversion in an implementation-defined mannerT~2~``T~1~``T~2~
- T~1~ to where implements IConvertibleT~2~``T~1~
- T~1~ to where or implements the method T~2~``T~1~``T~2~``T~2~ op_Implicit(T1)
- T~1~ to where or implements the method T~2~``T~1~``T~2~``T~2~ op_Explicit(T1)
- T~1~ to where implements a constructor taking a single argument of type T~2~``T~2~``T~1~
- Either of the following conversions:
  - string to where implements a static method or T``T``T Parse(string)``T Parse(string, IFormatProvider)
  - T~1~ to where is any enum and is either string or a collection of objects that can be converted to stringT~2~``T~2~``T~1~
- T to PSObject where is any typeT
- Any of the following conversions: Language
  - T to bool where is any numeric typeT

  - string to where is , , , , , , or T``T``regex``wmisearcher``wmi``wmiclass``adsi``adsisearcher`
  - T to bool
  - T_1 to where a conversion from to existsNullable[T~2~]``T~1~``T~2~
  - T to void
  - T~1~[] to where an assignable conversion between and existsT~2~[]``T~1~``T~2~
  - T~1~ to where is a collectionT~2~[]``T~1~
  - IDictionary to Hashtable
  - T to ref
  - T to xml
  - scriptblock to delegate
  - T~1~ to where is an integer type and is an enumT~2~``T~1~``T~2~
- $null to where is any value typeT``T
- $null to where is any reference typeT``T
- Any of the following conversions:
  - byte to where is T``T``SByte
  - UInt16 to where is , , or T``T``SByte``byte``Int16
  - Int16 to where is or T``T``SByte``byte
  - UInt32 to where is , , , , or T``T``SByte``byte``Int16``UInt16``int
  - int to where is , , , or T``T``SByte``byte``Int16``UInt16
  - UInt64 to where is , , , , , , or T``T``SByte``byte``Int16``UInt16``int``UInt32``long
  - long to where is , , , , , or T``T``SByte``byte``Int16``UInt16``int``UInt32
  - float to where is any integer type or T``T``decimal
  - double to where is any integer type or T``T``decimal
  - decimal to where is any integer typeT``T
- Any of the following conversions:
  - SByte to where is , , , or T``T``byte``uint6``UInt32``UInt64
  - Int16 to where is , , or T``T``UInt16``UInt32``UInt64
  - int to where is or T``T``UInt32``UInt64
  - long to UInt64
  - decimal to where is or T``T``float``double
- Any of the following conversions:
  - T to where is any numeric typestring``T
  - T to where is any numeric typechar``T
  - string to where is any numeric typeT``T
- Any of the following conversions, these conversion are considered an assignable conversions:
  - byte to where is , , , , , , , , or T``T``Int16``UInt16``int``UInt32``long``UInt64``single``dou
  - SByte to where is , , , , , , , , or T``T``Int16``UInt16``int``UInt32``long``UInt64``single``do
  - UInt16 to where is , , , or , , , or T``T``int``UInt32``long``UInt64``single``double``decimal
  - Int16 to where is , , , or , , , or T``T``int``UInt32``long``UInt64``single``double``decimal
  - UInt32 to where is , or , , , or T``T``long``UInt64``single``double``decimal
  - int to where is , , , , or T``T``long``UInt64``single``double``decimal
  - single to double
- T~1~ to where is a base class or interface of . This conversion is considered an assignable conversion.T~2~``T~2~``T~1~

- `string` to `char[]`
- `T` to – This conversion is considered an assignable conversion.`T`

For each conversion of the form to where is not an array and no other conversion applies, if there is a conversion from to , the rank of the conversion is worse than the conversion from to , but better than any conversion ranked less than the conversion from to `T~1~``T~2~[]``T~1~``T~1~``T~2~``T~1~``T~2~``T~1~``T~2~`

3.8 Name lookup

It is possible to have commands of different kinds all having the same name. The order in which name lookup is performed in such a case is alias, function, cmdlet, and external command.

3.9 Type name lookup

§7.1.10 contains the statement, "A *type-literal* is represented in an implementation by some unspecified *underlying type*. As a result, a type name is a synonym for its underlying type." Example of types are , , , and `.int``double``long[]``Hashtable`

Type names are matched as follows: Compare a given type name with the list of built-in *type accelerators*, such as int, long, double. If a match is found, that is the type. Otherwise, presume the type name is fully qualified and see if such a type exists on the host system. If a match is found, that is the type. Otherwise, add the namespace prefix . If a match is found, that is the type. Otherwise, the type name is in error. This algorithm is applied for each type argument for generic types. However, there is no need to specify the arity (the number of arguments or operands taken by a function or operator).`System.`

3.10 Automatic memory management

Various operators and cmdlets result in the allocation of memory for reference-type objects, such as strings and arrays. The allocation and freeing of this memory is managed by the PowerShell runtime system. That is, PowerShell provides automatic *garbage collection*.

3.11 Execution order

A *side effect* is a change in the state of a command's execution environment. A change to the value of a variable (via the assignment operators or the pre- and post-increment and decrement operators) is a side effect, as is a change to the contents of a file.

Unless specified otherwise, statements are executed in lexical order.

Except as specified for some operators, the order of evaluation of terms in an expression and the order in which side effects take place are both unspecified.

An expression that invokes a command involves the expression that designates the command, and zero or more expressions that designate the arguments whose values are to be passed to that command. The order in which these expressions are evaluated relative to each other is unspecified.

3.12 Error handling

When a command fails, this is considered an *error*, and information about that error is recorded in an *error record*, whose type is unspecified (§4.5.15); however, this type supports subscripting.

An error falls into one of two categories. Either it terminates the operation (a *terminating error*) or it doesn't (a *non-terminating error*). With a terminating error, the error is recorded and the operation stops. With a non-terminating error, the error is recorded and the operation continues.

Non-terminating errors are written to the error stream. Although that information can be redirected to a file, the error objects are first converted to strings and important information in those objects would not be captured making diagnosis difficult if not impossible. Instead, the error text can be redirected (§7.12) and the error object saved in a variable, as in .`$Error1 = command 2>&1`

The automatic variable contains a collection of error records that represent recent errors, and the most recent error is in . This collection is maintained in a buffer such that old records are discarded as new ones are added. The automatic variable controls the number of records that can be stored.`$Error``$Error[0]``$MaximumErrorCount`

`$Error` contains all of the errors from all commands mixed in together in one collection. To collect the errors from a specific command, use the common parameter ErrorVariable, which allows a user-defined variable to be specified to hold the collection.

3.13 Pipelines

A *pipeline* is a series of one or more commands each separated by the pipe operator (U+007C). Each command receives input from its predecessor and writes output to its successor. Unless the output at the end of the pipeline is discarded or redirected to a file, it is sent to the host environment, which may choose to write it to standard output. Commands in a pipeline may also receive input from arguments. For example, consider the following use of commands , , and , which create a list of file names in a given file system directory, sort a set of text records, and perform some processing on a text record, respectively:`|``Get-ChildItem``Sort-Object``Process-File`

```
Get-ChildItem
Get-ChildItem e:*.txt | Sort-Object -CaseSensitive | Process-File >results.txt
```

In the first case, creates a collection of names of the files in the current/default directory. That collection is sent to the host environment, which, by default, writes each element's value to standard output.`Get-ChildItem`

In the second case, creates a collection of names of the files in the directory specified, using the argument . That collection is written to the command , which, by default, sorts them in ascending order, sensitive to case (by virtue of the **CaseSensitive** argument). The resulting collection is then written to command , which performs some (unknown) processing. The output from that command is then redirected to the file .`Get-ChildItem``e:*.txt``Sort-Object``Process-File``results.txt`

If a command writes a single object, its successor receives that object and then terminates after writing its own object(s) to its successor. If, however, a command writes multiple objects, they are delivered one at a time to the successor command, which executes once per object. This behavior is called *streaming*. In stream processing, objects are written along the pipeline as soon as they become available, not when the entire collection has been produced.

When processing a collection, a command can be written such that it can do special processing before the initial element and after the final element.

3.14 Modules

A *module* is a self-contained reusable unit that allows PowerShell code to be partitioned, organized, and abstracted. A module can contain commands (such as cmdlets and functions) and items (such as variables and aliases) that can be used as a single unit.

Once a module has been created, it must be *imported* into a session before the commands and items within it can be used. Once imported, commands and items behave as if they were defined locally. A module is imported explicitly with the command. A module may also be imported automatically as determined in an implementation defined manner.`Import-Module`

The type of an object that represents a module is described in §4.5.12.

Modules are discussed in detail in §11.

3.15 Wildcard expressions

A wildcard expression may contain zero or more of the following elements:

Element	Description
Character other than *, ?, or [	Matches that one character
*	Matches zero or more characters. To match a * character, use [*].

Element	Description
?	Matches any one character. To match a ? character, use [?].
[*set*]	

Matches any one character from *set*, which cannot be empty.

If *set* begins with], that right square bracket is considered part of *set* and the next right square bracket terminates the set; otherwise, the first right square bracket terminates the set.

If *set* begins or ends with -, that hyphen-minus is considered part of *set*; otherwise, it indicates a range of consecutive Unicode code points with the characters either side of the hyphen-minus being the inclusive range delimiters. For example, A-Z indicates the 26 uppercase English letters, and 0-9 indicates the 10 decimal digits.

|

3.16 Regular expressions

A regular expression may contain zero or more of the following elements:

Element	Description
Character other than ., [, ^, *, $, or \	Matches that one character
.	Matches any one character. To match a . character, use \..
[*set*]	
[^*set*]	

The [*set*] form matches any one character from *set*. The [^*set*] form matches no characters from *set*. *set* cannot be empty.

If *set* begins with] or ^], that right square bracket is considered part of *set* and the next right square bracket terminates the set; otherwise, the first right square bracket terminates the set.

If *set* begins with - or ^-, or ends with -, that hyphen-minus is considered part of *set*; otherwise, it indicates a range of consecutive Unicode code points with the characters either side of the hyphen-minus being the inclusive range delimiters. For example, A-Z indicates the 26 uppercase English letters, and 0-9 indicates the 10 decimal digits.

| | * | Matches zero of more occurrences of the preceding element. | | + | Matches one of more occurrences of the preceding element. | | ? | Matches zero of one

occurrences of the preceding element. | | ^ | Matches at the start of the string. To match a ^ character, use \^. | | $ | Matches at the end of the string. To match a $ character, use $. | | \c | Escapes character *c*, so it isn't recognized as a regular expression element. |

Windows PowerShell: Character classes available in Microsoft .NET Framework regular expressions are supported, as follows:

Element	Description
`\p{name}`	Matches any character in the named character class specified by *name*. Supported names are Unicode groups and block ranges such as Ll, Nd, Z, IsGreek, and IsBoxDrawing.
`\P{name}`	Matches text not included in the groups and block ranges specified in *name*.
`\w`	Matches any word character. Equivalent to the Unicode character categories . If ECMAScript-compliant behavior is specified with the ECMAScript option, \w is equivalent to `.[\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}]``[a-zA-Z_0-9]`
`\W`	Matches any non-word character. Equivalent to the Unicode categories `.[\^\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}]`
`\s`	Matches any white space character. Equivalent to the Unicode character categories. `[\f\n\r\t\v\x85\p{Z}]`
`\S`	Matches any non-white-space character. Equivalent to the Unicode character categories `.[\^\f\n\r\t\v\x85\p{Z}]`
`\d`	Matches any decimal digit. Equivalent to for Unicode and for non-Unicode behavior.`\p{Nd}``[0-9]`
`\D`	Matches any non-digit. Equivalent to for Unicode and for non-Unicode behavior.`\P{Nd}``[\^0-9]`

Quantifiers available in Microsoft .NET Framework regular expressions are supported, as follows:

Element	Description
`*`	Specifies zero or more matches; for example, or Equivalent to `.\w*(abc)*.``{0,}`
`+`	Matches repeating instances of the preceding characters.
`?`	Specifies zero or one matches; for example, or . Equivalent to `.\w?``(abc)?``{0,1}`
`{n}`	Specifies exactly *n* matches; for example, `.(pizza){2}`
`{n,}`	Specifies at least *n* matches; for example, `.(abc){2,}`
`{n,m}`	Specifies at least *n*, but no more than *m*, matches.

Recommended content

- [

Flow control - PowerShell

```
](<https://docs.microsoft.com/en-us/powershell/scripting/learn/ps101/06-flow-control>)

PowerShell provides methods to create loops, make decisions, and logically control the flow
```

Feedback

Submit and view feedback for